Oracle® OLAP

DML Reference

10*g* Release 1 (10.1)

Part No. B10339-02

December 2003

Oracle OLAP DML Reference, 10*g* Release 1 (10.1)

Part No. B10339-02

Contents

Send Us Your Comments

Oracle OLAP DML Reference, 10*g* Release 1 (10.1)

Part No. B10339-02

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the title and part number of the documentation and the chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail: infodev@us.oracle.com
- FAX: 781-238-9850 Attn: Oracle OLAP
- Postal service:
  Oracle Corporation
  Oracle OLAP
  10 Van de Graaff Drive
  Burlington, MA 01803
  USA

If you would like a reply, please give your name, address, telephone number, and electronic mail address (optional).

If you have problems with the software, please contact your local Oracle Support Services.

Preface

The *Oracle OLAP DML Reference* provides a complete description of the OLAP Data Manipulation Language (OLAP DML) used to define and manipulate analytic workspace objects. Part I briefly describes how to use the OLAP DML. Part II consists of a reference topic for each of the OLAP DML statements.

Intended Audience

This manual is intended for programmers and database administrators who create and modify analytic workspaces and analytic workspace objects using the OLAP DML.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For additional information, visit the Oracle Accessibility Program Web site at

`http://www.oracle.com/accessibility/`

Accessibility of Code Examples in Documentation JAWS, a Windows screen reader, may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, JAWS may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Structure

This document is structured in the following three parts.

Part I, "Using the OLAP DML"

Introduces the basic concepts of analyzing data using the OLAP DML.

Chapter 1, "Introduction to the OLAP DML"
Provides an introduction to the OLAP DML and provides some basic information that you need to know to effectively use the OLAP DML.

Chapter 2, "Data Types and Operators"
Introduces the OLAP DML data types and operators.

Chapter 3, "Expressions"
Explains how to create and use OLAP DML expressions.

Chapter 4, "Formulas, Aggregations, Allocations, and Models"
Explains how to create and execute OLAP DML calculation specification objects.

Chapter 5, "OLAP DML Programs"
Explains how to create, test, and execute OLAP DML programs.

Part II, "Alphabetic Reference"

Consists of a topic for each of the OLAP DML statements, arranged alphabetically. Each topic provides a description, syntax, notes, and example for an OLAP DML statement.

Chapter 6, "$AGGMAP to AGGMAP"
Chapter 7, "AFFMAPINFO to ARCCOS"
Chapter 8, "ARCSIN to CHARLIST"
Chapter 9, "CHGDFN to DDOF"

Chapter 10, "DECIMALCHAR to DELETE"
Chapter 11, "DEPRDECL to EXISTS"
Chapter 12, "EXPORT to FILEMOVE"
Chapter 13, "FILENEXT to FULLDSC"
Chapter 14, "GET to IMPORT"
Chapter 15, "INF_STOP_ON_ERROR to LIKEESCAPE"
Chapter 16, "LIKENL to MAX"
Chapter 17, "MAXBYTES to MODTRACE"
Chapter 18, "MONITOR to NVL2"
Chapter 19, "OBJ to QUAL"
Chapter 20, "RANDOM to REPORT"
Chapter 21, "RESERVED to SPARSEINDEX"
Chapter 22, "SQL to STATVAL"
Chapter 23, "STDDEV to TRACKPRG"
Chapter 24, "TRAP to ZSPELL"

Part III, "Appendixes"

Provides summary information about OLAP DML statements.

Appendix A, "Functions and Commands by Functional Category"
This appendix lists the OLAP DML functions, commands, and programs by functional category.

Appendix B, "OLAP DML Statement Changes"
This appendix provides information about recent changes to the OLAP DML.

Related Documents

The following documents provide additional information about using Oracle OLAP:

- *Oracle OLAP Application Developer's Guide*

  Explains how SQL and Java applications can extend their analytic processing capabilities by using Oracle OLAP in the Enterprise Edition of Oracle Database.

- *Oracle OLAP Reference*

  Explains the syntax of PL/SQL packages and types and the column structure of views related to Oracle OLAP.

- *Oracle OLAP Java API Reference*

  Introduces the Oracle OLAP API, a Java application programming interface for Oracle OLAP, which is used to perform online analytical processing of the data stored in an Oracle database. Describes the API and how to discover metadata, create queries, and retrieve data.

- *Oracle OLAP Java API Reference*

  Describes the classes and methods in the Oracle OLAP Java API for querying analytic workspaces and relational data warehouses.

- *Oracle OLAP Analytic Workspace Java API Reference*

  Describes the classes and methods in the Oracle OLAP Java API for building and maintaining analytic workspaces.

Conventions

The following conventions are used in this manual.

Convention	Meaning
...	Horizontal ellipsis points in examples mean that information not directly related to the example has been omitted. Horizontal ellipsis points in syntax indicate a repeating argument or clause.
boldface text	Boldface type in text indicates a term defined in the text.
italic text	Italicized type in text indicates emphasis. Italicized type in syntax indicates or a user-supplied name.
[]	Brackets in syntax enclose optional clauses from which you can choose one or none.
{}	Braces in syntax enclose required clauses from which you must chose one.

Part I

Using the OLAP DML

Part I introduces the basic concepts of the OLAP DML and contains overview information about using the OLAP DML to perform multidimensional analysis.

This part contains the following chapters:

1 Introduction to the OLAP DML

This chapter provides an introduction to the OLAP DML. It includes the following topics:

- What is the OLAP DML?
- Basic Syntactical Units
- OLAP DML as a Data Definition Language
- OLAP DML as a Data Manipulation Language

What is the OLAP DML?

The OLAP DML is a language that defines and manipulates data in an analytic workspace.

- As a data definition language, you can use DML statements to create analytic workspaces and analytic workspace data objects. See "OLAP DML as a Data Definition Language" on page 1-5 for more information.
- As a data manipulation language, you can use DML statements to perform complex analysis of data. See "OLAP DML as a Data Manipulation Language" on page 1-10 for more information.

The purpose of the OLAP DML is to enable application developers to extend the analytical capabilities of querying languages such as SQL and the OLAP API for Java.

Basic Syntactical Units

The basic syntactic units of the OLAP DML are:

- Options to which you assign a value and that can influence the analytic workspace processing environment in various ways
- Properties that Oracle OLAP checks to determine processing
- Commands that initiate actions and functions that initiate actions and return a value.
- Programs that perform complicated analysis and reporting

OLAP DML commands, functions, options, programs, and properties are collectively referred to as OLAP DML statements. Part I of this manual introduces basic elements of the OLAP DML. The complete syntax of each statement, usage notes, and examples is provided in Part II of this manual. Lists of statements, arranged by functional category are presented in Appendix A

OLAP DML Options

An option is a special type of analytic workspace object that specifies the characteristic of some aspect of how Oracle OLAP calculates or formats data or what Oracle OLAP operations are activated. You cannot define an option as part of a workspace. However, you can use any of the options that are defined as part of the Oracle OLAP DML.

Some options are read-only, while others are read/write options for which you can specify values. Read/write options have default values.

Categories of Options

OLAP DML options fall into the following general categories:

- Aggregating data. See Table 6–1 on page 6-39.
- Allocating data. See Table 7–4 on page 7-52.
- Modeling data. See Table 17–1 on page 17-23.
- Compiling programs, models and aggregation specifications. See Table 9–1 on page 9-30.
- Handling errors. See Table 5–2 on page 5-15.
- Debugging programs and models. See Table 5–3 on page 5-15.
- Embedding SQL in OLAP DML programs. See Table 22–1 on page 22-5.
- File reading and writing. See Table 15–1 on page 15-6.
- Importing and exporting analytic workspaces. See Table 12–1 on page 12-7.

- OLAP DML reports. See Table 20–6 on page 20-65.
- Working with empty cells. See Table 3–5 on page 3-27.
- Using date and time values. See Table 2–6 on page 2-5.
- Calculating data. See Table 3–2 on page 3-11.
- Oracle Database options. See Table 18–2 on page 18-54.

Syntax for Specifying and Retrieving Option Values

The general syntax for specifying and retrieving option values is shown in Table 1–1, " Syntax for Specifying and Retrieving Option Values" on page 1-3.

Table 1–1 Syntax for Specifying and Retrieving Option Values

Action	Syntax
To specify an option value	*option-name* = *value*
To display an option value	SHOW *option-name*
To retrieve an option value into a predefined variable	*variable-name* = *option-name*

OLAP DML Properties

A property is a named value that is associated with a definition of an analytic workspace object. You name, create, and assign properties to an object using a PROPERTY statement.

Properties that begin with a $ (dollar sign) are recognized by Oracle OLAP as system properties. You assign system properties to objects the same way that you create other properties; however, you must give them the appropriate name in order for Oracle OLAP to recognize them. Part II, "Alphabetic Reference" includes tables that list two types of system properties:

- Table 6–2 on page 6-39 lists system properties that you can use to specify default behavior when aggregating or allocating data.
- Table 3–6 on page 3-28 lists system properties that you can use to specify behavior in regard to empty data cells.

OLAP has other system properties that are not as integral to the use of the OLAP DML. For example, properties are part of the object definitions for an analytic workspace that has database standard form.

OLAP DML Commands and Functions

Most OLAP DML statements are either OLAP DML commands and functions. OLAP DML commands and commands work in much the same way as commands and functions in other programming languages—the one exception is the "looping" nature of OLAP DML commands and functions discussed in "Looping Nature of OLAP DML Commands and Functions" on page 5-5.

OLAP DML Commands

Many OLAP DML statements are commands that perform complex actions. Some of these commands are data definition commands that you use to create an analytic workspace or define objects within an analytic workspace. Data definition commands are introduced in "OLAP DML as a Data Definition Language" on page 1-5.

Other OLAP DML commands are complex data manipulation commands. For example, you can use the OLAP DML SQL command to embed SQL statements in an OLAP DML program in order to copy data from relational tables into analytic workspace data objects, or you can use the AGGREGATE command to calculate summary data. You can also augment the functionality of the OLAP DML by writing an OLAP DML program for use as a command.

OLAP DML Functions

Most of the OLAP DML functions are simple text or calculation functions (that is, numeric, financial, statistical, date, time, time-series functions, and aggregation functions), or data type conversion functions. For tables listing these standard functions, see:

"Text Functions" on page A-4
"Date and Time Functions" on page A-8
"General Numeric Functions" on page A-10
"Financial Functions" on page A-12
"Statistical Functions" on page A-13
"Time-Series Functions" on page A-14
"Aggregation Functions" on page A-15
"Data Type Conversion" on page A-3

Other OLAP DML functions return more complex information. For example, the OLAP DML provides the AW function that you can use to retrieve many different types of information about an analytic workspace and the AGGREGATE function that you can use to calculate aggregate data on-the-fly at user request.

You can also augment the functionality of the OLAP DML by writing an OLAP DML program for use as a function.

OLAP DML Programs

Some OLAP DML statements are actually the names of OLAP DML programs provided as part of the OLAP DML. Some of these programs produce reports that you can print or see online. For example, the AWDESCRIBE program produces a report that consists of a summary page; an alphabetic list of analytic workspace objects showing name, type, and description; and a list of object definitions by object type.

Other programs provided as part of the OLAP DML perform standard calculations of use to programmers and database administrators. For example, VALSPERPAGE program calculates the maximum number of values for a variable of a given width that will fit on one analytic workspace page.

You execute programs provided as part of the OLAP DML the same way that you do any other OLAP DML statement following the syntax provided for that program in Part II, "Alphabetic Reference".

You can also write your own OLAP DML programs to augment the functionality of the OLAP DML as described in Chapter 5, "OLAP DML Programs".

OLAP DML as a Data Definition Language

The OLAP DML provides statements that you can use to create and manage analytic workspaces and the object definitions within them.

This section provides overview information about the statements that you use to:

- Define and manage analytic workspaces
- Define analytic workspace objects
- View analytic workspace definitions

Statements for Creating Analytic Workspaces

Table 1–2, " Statements for Creating and Managing Analytic Workspaces" on page 1-6 lists the OLAP DML statements that you use to create and manipulate analytic workspaces. Table 1–3, " Options Related to Creating or Attaching Analytic Workspaces" on page 1-6 lists the OLAP DML options that relate to these statements.

Table 1–2 Statements for Creating and Managing Analytic Workspaces

Statement	Description
AW command	Creates a new workspace; allocates space for a workspace; attaches a workspace to a session; deletes a workspace; detaches a workspace from a session; sets up a workspace for multiple segments; or sends to the current outfile a list of the active workspaces, along with their update status.
COMMIT	Executes a SQL COMMIT statement.
UPDATE	Moves analytic workspace changes from a temporary area to the database table in which the workspace is stored. The table is not saved until you execute a COMMIT command, either from Oracle OLAP or from SQL.

Table 1–3 Options Related to Creating or Attaching Analytic Workspaces

Statement	Description
AWWAITTIME	An option that contains the number of seconds that AW ATTACH with the if the WAIT keyword waits for an analytic workspace to become available for access.
DEFAULTAWSEGSIZE	An option that specifies the default maximum segment size for an analytic workspace created in your database session.

Defining Analytic Workspace Objects

An analytic workspace contains two types of objects:

- Data objects that contain the data that you want to analyze and the results of the analysis.
- Calculation specifications that contain OLAP DML statements that specify the analysis that you want performed.

Table 1–4, " Workspace Object Data Definition Statements" on page 1-6 lists the OLAP DML statements that relate to defining analytic workspace objects. For more specific information, see "Defining Data Objects Using the OLAP DML" on page 1-7 and ore information on calculation specification objects, see "Defining Calculation Specification Objects Using the OLAP DML" on page 1-8.

Table 1–4 Workspace Object Data Definition Statements

Statement	Description
CHGDFN	Changes certain aspects of the definitions of certain objects.

Table 1–4 (Cont.) Workspace Object Data Definition Statements

Statement	Description
CONSIDER	Identifies a definition as the current definition. This enables you to add a description, property, calculation specification, or trigger (event) to an object.
COPYDFN	Defines a new object in the analytical workspace and uses the same definition as a specified object in the current workspace or in an attached workspace.
DEFINE	Adds a new object to the analytic workspace.
DELETE	Deletes one or more objects from a workspace.
LD	Assigns a description to an object that has already been defined.
MOVE	Moves an object name to a new position in the NAME dimension of a workspace.
PERMITRESET	Causes the values of permission conditions to be reevaluated. Permission conditions consist of one or more Boolean expressions that designate the criteria used by PERMIT commands associated with an object.
PROPERTY	Assigns a property to an object. A property is a named value that is associated with a given object definition.
RENAME	Changes the name of an object in an analytical workspace and updates associated objects.
TRIGGER command	Associates a previously-created program to an object and identifies the object event that automatically executes the program; or a disassociates a trigger program from the object
VALSPERPAGE	Calculates the maximum number of values for a variable of a given width that will fit on one page. Pages are units of storage in the workspace.

Defining Data Objects Using the OLAP DML

Data objects contain the data that you want to analyze and the results of the analysis. Data objects are implemented as arrays and indexes.

Table 1–5, " OLAP Data Object Definition Statements" on page 1-8 briefly describes the data objects that you can define in an analytic workspace and the OLAP DML statements that you use to define these objects.

Table 1–5 OLAP Data Object Definition Statements

Object Name	Description	DEFINE command
Variable	An array of values that you want to analyze or an array of values that are the result of the analysis.	DEFINE VARIABLE
Dimension	A dimension or index to one or more variables or relations, or provide a list of values to an OLAP DML program.	DEFINE DIMENSION
Composite	A list of dimension value combinations that you use to dimension variables when you do not want the variable to have empty cells.	DEFINE COMPOSITE
Relation	A multidimensional array whose values specify correspondence between the values of one or more dimensions. For example, a parent relation for a hierarchical dimension describes the child-parent relationship of the values within the dimension.	DEFINE RELATION

Oracle OLAP also supports the definition of dimension surrogates and valuesets that you can use in calculations instead of dimensions. (You cannot use these objects to dimension variables or relations.) See DEFINE SURROGATE and DEFINE VALUESET for more information.

Defining Calculation Specification Objects Using the OLAP DML

Calculation specifications contain OLAP DML statements that specify analysis that you want performed.

Types of Calculation Specifications Using the OLAP DML you can define objects that are specifications for different types of OLAP calculation.

- Formulas—A formula is a saved expression.
- Aggregations—An aggregation is a specification for how data should be aggregated..
- Allocations—An allocation is a specification for how data should be allocated.
- Models—A model is a set of interrelated equations. The calculations in an equation can be based either on variables or on dimension values. You can assign the results of the calculations directly to a variable or you can specify a dimension value for which data is being calculated.

- Programs—An OLAP DML program is a collection of OLAP DML statements that helps you accomplish some workspace management or analysis task. You can use OLAP DML programs as user-defined commands and functions.

Creating Calculation Specification Objects The general process of creating a calculation specification object is the following two step process:

1. Define the calculation object using the appropriate DEFINE command.
2. Add the calculation specification to the object definition. You can add the calculation specification to the definition of a calculation object in the following ways:
   - At the command line level of the OLAP Worksheet, in an input file, or as an argument to a PL/SQL function. In this case, ensure that the object is the current object (issue a CONSIDER statement, if necessary), and, then, issue the appropriate command that includes the specification as a multiline text argument. To code the specification as a multiline text, you can use a JOINLINES function where each of the text arguments of JOINLINES is a statement that specifies the desired processing, and where the final statement is `END`.
   - In an Edit Window of the OLAP Worksheet. In this case, at the command line level of the OLAP Worksheet, issue an EDIT statement with the appropriate keyword. This opens an Edit Window for the specified object. You can then type each statement as an individual line in the Edit Window. Saving the specification and closing the Edit Window when you are finished.

Table 1–6 outlines the OLAP DML statements that you use to create each type of calculation specification. For more detailed information on creating calculation specifications, see the relevant DEFINE statement, Chapter 4, "Formulas, Aggregations, Allocations, and Models", and Chapter 5, "OLAP DML Programs".

Table 1–6 Commands for Defining Calculation Specifications

Specification Type	Definition Statement	Command for Entering Specification	Statement for Opening Edit Window
Aggregation	DEFINE AGGMAP	AGGMAP	EDIT AGGMAP *aggmap-name*
Allocation	DEFINE AGGMAP	ALLOCMAP	EDIT AGGMAP *aggmap-name*

Table 1–6 (Cont.) Commands for Defining Calculation Specifications

Specification Type	Definition Statement	Command for Entering Specification	Statement for Opening Edit Window
Formula	DEFINE FORMULA	EQ	EDIT FORMULA *formula-name*
Model	DEFINE MODEL	MODEL	EDIT MODEL *model-name*
Program	DEFINE PROGRAM	PROGRAM	EDIT [PROGRAM] *program-name*

Viewing Data Definitions

Table 1–7, " Statements for Viewing Definitions" on page 1-10 lists the OLAP DML statements that you can use to view definitions stored in an analytic workspace

Table 1–7 Statements for Viewing Definitions

Statement	Description
AW function	Returns information about currently attached workspaces.
EXISTS	Returns a value that indicates whether an object is defined in any attached workspace.
LISTBY	Lists all objects in a workspace that are dimensioned by or related to one or more specified dimensions or composites.
LISTNAMES	Lists the names of the objects in a workspace.
OBJ	Returns information about a workspace object.
OBJLIST	Lists the objects that in one or more workspaces that you specify.
AWDESCRIBE	Sends information about the current analytic workspace to the current outfile.
DESCRIBE	Lists the base definition of one or more workspace objects.
FULLDSC	Lists the definition of one or more workspace objects, including the properties and triggers of the object(s).

OLAP DML as a Data Manipulation Language

The real power of the OLAP DML is apparent when you begin using it to analyze your data. Using the OLAP DML you can:

- Define special OLAP DML objects (sometimes called calculation specification objects) that you can use to aggregate, allocate, or model data. (See "Defining Calculation Specification Objects Using the OLAP DML" on page 1-8 for a general discussion of this process.)
- Write programs to perform complex analysis.You can write a OLAP DML programs to perform almost any type of complicated multidimensional analysis.

This section provides overview information about the following types of programs:

- "Startup Programs" on page 1-11
- "Data Loading Programs" on page 1-13
- "Trigger Programs" on page 1-14
- "Aggregation, Allocation, and Modeling Programs" on page 1-15
- "Forecasting Programs" on page 1-16
- "Programs to Export and Import Workspace Objects" on page 1-16

For more information on creating an OLAP DML program, see Chapter 5, "OLAP DML Programs".

Startup Programs

Startup programs are programs that you write and that Oracle OLAP checks for by name when an AW ATTACH statement executes. Startup programs do not exist within an analytic workspace unless you define and write them. In a startup program you can execute any OLAP DML statements, or run any of your own programs. For example, a startup program might set options to values appropriate to your application.

The types of startup programs that are recognized by Oracle OLAP are discussed in this topic. The order in which these programs are executed is discussed in "Programs Executed When Attaching Analytic Workspaces" on page 8-39.

ONATTACH Programs

You can create an Onattach program in one of two ways:

- You can define a program named `ONATTACH`. Each time you attach the workspace, the ONATTACH program executes automatically unless you include a NOOTTACH keyword in the AW ATTACH statement.

- You can define a program and give it any name you want. When attaching the workspace using a AW ATTACH statement, you can run the program by specifying its name after the ONATTACH keyword. This is useful for application developers; an application can run a different startup program depending on the users' choices.

Permission Programs

The startup programs named PERMIT_READ and PERMIT_WRITE are also known as permission programs. Permission programs allow you to control two levels of access to the analytic workspace in which they reside.

- Access at the analytic workspace level—Depending on the return value of the permission program, the user is or is not granted access to the entire analytic workspace. You can use the return value to indicate to Oracle OLAP whether or not the user has the right to attach the workspace.
- Access at the object level—Depending on the statements in the permission program, the user is granted or denied access to specific objects or sets of object values. Within an ONATTACH program, you can use ACQUIRE statements to provide access to individual workspace objects. Within a permission program for read-only or read/write attachment, you can specify PERMIT commands that grant or restrict access to individual workspace objects.

Note: All of the objects referred to in a given permission program must exist in the same analytic workspace.

To create a permission program, define a user-defined function (as described in "Creating User-Defined Functions" on page 5-2) with one of the recognized names, then define the contents for the program as described in "Specifying Program Contents" on page 5-2.

AUTOGO Programs

You can create an Autogo program by defining a program with any name, and specifying that name in the AW ATTACH statement after the AUTOGO keyword.

TRIGGER_AW Program

When you create a program named TRIGGER_AW program, the execution of any AW command (including an AW ATTACH statement) becomes an event that triggers the execution of the TRIGGER_AW program.

Data Loading Programs

The OLAP DML provides support for loading data to and from relational tables, flat files, and spreadsheets.

Programs that Copy Data From Relational Tables to Workspace Objects

You can embed SQL statements in OLAP DML programs using the OLAP DML SQL command. Oracle OLAP provides statements that you can use in a program to copy relational data into analytic workspace objects using either an implicit cursor or an explicit cursor:

- To copy data from relational tables into analytic workspace objects using an implicit cursor, use the SQL SELECT command. You can use this OLAP DML command interactively in the OLAP Worksheet or within an OLAP DML program.
- To copy data from relational tables into analytic workspace objects using an implicit cursor, use the following statements in the order indicated. You can only use these commands within an OLAP DML program. You cannot use them interactively in the OLAP Worksheet.
  1. SQL DECLARE CURSOR defines a SQL cursor by associating it with a SELECT statement or procedure.
  2. SQL OPEN activates a SQL cursor.
  3. SQL FETCH and SQL IMPORT retrieve and process data specified by a cursor.
  4. SQL CLOSE closes a SQL cursor.
  5. SQL CLEANUP cancels a SQL cursor declaration and frees the memory resources of an SQL cursor.

For examples of programs that copy table data into workspace objects, see SQL FETCH and SQL IMPORT.

File-Reading Programs

Oracle OLAP provides a number of statements that you can use to read data from flat files. These statements (listed in "File Reading and Writing Statements" on page A-28) are frequently used together in a special program.

Spreadsheet Import Programs

Within an OLAP DML program you can use the IMPORT (from spreadsheet) command to imp;ort data from a spreadsheet into analytic workspace objects.

Trigger Programs

DEFINE, MAINTAIN, PROPERTY, SET (=) UPDATE, and AW commands are recognized by Oracle OLAP as events that can trigger the execution of OLAP DML programs.

- Programs triggered by AW ATTACH, are called startup programs and are discussed in "Startup Programs" on page 1-11 and in the topic for AW ATTACH.
- Programs triggered by AW CREATE, AW DELETE, AW DETACH, DEFINE, MAINTAIN, PROPERTY, UPDATE, and SET are called trigger programs and are discussed in this section and in the topic for the TRIGGER command.

Trigger programs are frequently written to maintain application-specific metadata. Trigger programs have certain characteristics depending on the statement that triggers them. Some trigger programs execute before the triggering statement executes; some after. Oracle OLAP passes arguments to programs triggered by some statements, but not others. Oracle OLAP does not change dimension status before most trigger programs execute, but does change dimension status before some MAINTAIN statements trigger program execution. In most cases, you can give a trigger program any name that you choose, but some events require a program with a specific name. "Characteristics of Trigger Programs" on page 24-11 discusses these characteristics.

Once an object is defined in an analytic workspace, you can create a trigger program for that object by following the following procedure:

1. Define the program as described in DEFINE PROGRAM.
2. Determine what to name the program and whether the program can be a user-defined program. (See Table 24–1, " Trigger Program Characteristics" on page 24-12.) If the program can be a user-defined program, decide whether or not you want to define the trigger program as a user-defined function.
3. Code the actual program as described in"Specifying Program Contents" on page 5-2.
4. Keep the following points in mind when coding trigger programs:
   - Use Table 24–1, " Trigger Program Characteristics" on page 24-12 to determine if Oracle OLAP will pass values to the program. If it will, use the

ARGUMENT command to declare these arguments in your program and the VARIABLE command to define program variables for the values. (See Table 24–2, " Arguments Passed to Trigger Programs" on page 24-13 for specific information about the arguments.)

- A program that is triggered by an Assign event is executed each time Oracle OLAP assigns a value to the object for which the event was defined. Thus, a program triggered by an Assign event is often executed over and over again as the assignment statements loops through a object assigning values. You can use TRIGGERASSIGN to assign a value that is different from the value specified by the assignment statement that triggered the execution of the program.
- In some cases, Oracle OLAP changes the status of the dimension being maintained when a Maintain event triggers the execution of a program. See Table 24–3, "How Programs Triggered by Maintain Events Effect Dimension Status" on page 24-14 for details
- Use the CALLTYPE function within a program to identify that the program was invoked as a trigger.

5. When the trigger program is *not* a TRIGGER_AFTER_UPDATE, TRIGGER_BEFORE_UPDATE, or TRIGGER_DEFINE program, associate the program with the desired object and event using the TRIGGER command.

   See also: The following statements:

   - TRIGGER function, DESCRIBE command, and OBJ function that retrieve information about triggers.
   - USETRIGGERS option that you can use to disable all triggers.

Aggregation, Allocation, and Modeling Programs

To aggregate, allocate, or model data using the OLAP DML, you first specify the calculation that you want performed by defining a calculation specification as outlined in "Defining Calculation Specification Objects Using the OLAP DML" on page 1-8. Later, if you want to populate variables with aggregated, allocated or modeled values as a database maintenance procedure, you write a program to execute the calculation object. For more information on the OLAP DML statements that you use in these programs, see "Executing the Aggregation" on page 4-4, "Allocating Data" on page 4-8, and "Running a Model" on page 4-15.

Forecasting Programs

Oracle OLAP provides statements that you can use to forecast data using a sophisticated forecast context. To forecast using this context, take the following steps:

1. Create the objects that you need to hold the results of the forecast.
2. Within the contents of a forecasting program, issue the following statements in the order indicated:
   a. FCOPEN function -- Creates a forecasting context.
   b. FCSET command -- Specifies the characteristics of a forecast.
   c. FCEXEC command -- Executes a forecast and populates Oracle OLAP variables with forecasting data.
   d. FCQUERY function -- Retrieves information about the characteristics of a forecast or a trial of a forecast.
   e. FCCLOSE command -- Closes a forecasting context.

For examples of forecasting programs, see Example 12–10, "A Forecasting Program" on page 12-43.

Programs to Export and Import Workspace Objects

You can export an entire workspace, several workspace objects, a single workspace object, or a portion of a workspace object to a specially formatted EIF file. Then you can import the information into a different workspace within the same Oracle database or a different one. The OLAP DML statements for importing and exporting data are listed in Table A–28, " Statements for Importing and Exporting Data" on page A-29.

One reason for exporting and importing is to move your data to a new location. Another purpose is to remove extra space from your analytic workspace after you have added and then deleted many objects or dimension values. To do this, issue an EXPORT statement to put all the data in an EIF file, create another workspace with a different name, and then use an IMPORT statement to import the EIF file into the new workspace. When you have imported into the same database, you can delete the old workspace and refer to the new one with the same workspace alias that you used for the original one.

The following statement copies all the data and definitions from the current analytic workspace to an EIF file called `reorg.eif` in a directory object called `mydir`.

```
EXPORT ALL TO EIF FILE 'mydir/reorg.eif'
```

2 Data Types and Operators

This chapter introduces the OLAP DML data types and operators. It includes the following topics:

- OLAP DML Data Types
- OLAP DML Operators

OLAP DML Data Types

Workspace data types fall into categories, which are referred to as basic data types. They are listed in Table 2–1, " OLAP DML Data Types".

Table 2–1 OLAP DML Data Types

Basic Type	Specific Types
Numeric	`INTEGER`, `SHORTINTEGER`, `LONGINTEGER`, `DECIMAL`, `SHORTDECIMAL`, `NUMBER`
Text	`TEXT`, `NTEXT`, `ID`
Boolean	`BOOLEAN`
Date	`DATETIME`, `DATE`

Different objects support the use of different data types for their values:

- For most values, including variable values, all of the data types are supported.
- For dimension values, only the `INTEGER`, `NUMBER`, `TEXT`, `ID`, and `NTEXT` data types are supported.

Also, when you want an OLAP DML program to be able to handle arguments without converting values to a specific data type, you can specify a data type of

WORKSHEET for the arguments and temporary variables in the program. Use the WKSDATA function to retrieve the data type of an argument with a WORKSHEET data type.

Numeric Data Types

The numeric data types described in Table 2–2, " OLAP DML Numeric Data Types" are supported.

Table 2–2 OLAP DML Numeric Data Types

Data Type	Data Value
INTEGER	A whole number in the range of (-2**31) to (2**31)-1.
SHORTINTEGER	A whole number in the range of (-2**15) to (2**15)-1.
LONGINTEGER	A whole number in the range of (-2**63) to (2**63)-1.
DECIMAL	A decimal number with up to 15 significant digits in the range of -(10**308) to +(10**308).
SHORTDECIMAL	A decimal number with up to 7 significant digits in the range of -(10**38) to +(10**38).
NUMBER	A decimal number with up to 38 significant digits in the range of -(10**125) to +(10**125).

For data entry, a value for any of these data types can begin with a plus (+) or minus (-) sign; it cannot contain commas. Note, however, that a comma is required *before* a negative number that follows another numeric expression, or the minus sign is interpreted as a subtraction operator. Additionally, a decimal value can contain a decimal point. For data display, thousands and decimal markers are controlled by the NLS_NUMERIC_CHARACTERS option.

Using LONGINTEGER Values

Most of the numerical data types return NA when a value is outside its range. However, the LONGINTEGER data type does not have overflow protection and will return an incorrect value when, for example, a calculation produces a number that exceeds its range. Use the NUMBER data type instead of LONGINTEGER when this is likely to be a problem.

Using NUMBER Values

When you define a NUMBER variable, you can specify its precision (*p*) and scale (*s*) so that it is sufficiently, but not unnecessarily, large. Precision is the number of significant digits. Scale can be positive or negative: Positive scale identifies the number of digits to the right of the decimal point; negative scale identifies the number of digits to the left of the decimal point that can be rounded up or down.

The NUMBER data type is supported by Oracle Database standard libraries and operates the same way as it does in SQL. It is used for dimensions and surrogates when a text or INTEGER data type is not appropriate. It is typically assigned to variables that are not used for calculations (like forecasts and aggregations), and it is used for variables that must match the rounding behavior of the database or require a high degree of precision. When deciding whether to assign the NUMBER data type to a variable, keep the following facts in mind in order to maximize performance:

- Analytic workspace calculations on NUMBER variables is slower than other numerical data types because NUMBER values are calculated in software (for accuracy) rather than in hardware (for speed).
- When data is fetched from an analytic workspace to a relational column that has the NUMBER data type, performance is best when the data already has the NUMBER data type in the analytic workspace because a conversion step is not required.

Text Data Types

The text data types described in Table 2–3, " OLAP DML Text Data Types" are supported by Oracle OLAP.

Table 2–3 OLAP DML Text Data Types

Data Type	Data Value
TEXT	Up to 4000 bytes for each line in the database character set. This data type is equivalent to the CHAR and VARCHAR2 data types in the database.
NTEXT	Up to 4000 bytes for each line in UTF-8 character encoding. This data type is equivalent to the NCHAR and NVARCHAR2 data types in the database.
ID	Up to 8 single-byte characters for each line in the database character set. (ID is valid only for values of simple dimensions, see DEFINE DIMENSION (simple).)

Literals

Enclose text literals in single quotes. Oracle OLAP recognizes unquoted alpha-numeric values as object names and double quotes as the beginning of a comment.

Escape Sequences

Table 2–4, " Recognized Escape Sequences" shows escape sequences that are recognized by Oracle OLAP.

Table 2–4 Recognized Escape Sequences

Sequence	Meaning
`\b`	Backspace
`\f`	Form feed
`\n`	Line feed
`\r`	Carriage return
`\t`	Horizontal tab
`\"`	Double quote
`\'`	Single quote
`\\`	Backslash
`\d`*nnn*	Character with ASCII code *nnn* decimal, where `\d` indicates a decimal escape and *nnn* is the decimal value for the character
`\x`*nn*	Character with ASCII code *nn* hexadecimal, where `\x` indicates a hexadecimal escape and *nn* is the hexadecimal value for the character
`\U`*nnnn*	Character with Unicode *nnnn*, where `\U` indicates a Unicode escape and *nnnn* is a four-digit hexadecimal integer that represents the Unicode codepoint with the value `U+`*nnnn*. The `U` must be a capital letter.

Boolean Data Type

A `BOOLEAN` data type enables you to represent logical values. In code, `BOOLEAN` values are represented by values for "no" and yes" (in any combination of uppercase and lowercase characters). The actual values that are recognized in your version of Oracle OLAP are determined by the language identified by the NLS_LANGUAGE option. You can use the read-only NOSPELL and YESSPELL options to obtain the values represent `BOOLEAN` values. In English language code, you can represent `BOOLEAN` values, using:

- YES, TRUE, ON
- NO, FALSE, OFF

Working with BOOLEAN expressions is discussed in "Boolean Expressions" on page 3-16.

Date Data Types

The date data types that are supported are listed in Table 2–5, " OLAP DML Date Data Types".

Table 2–5 OLAP DML Date Data Types

Data Type	Data Value
DATETIME	Dates between January 1, 4712 B.C. and December 31, 9999 A.D., and times in hours, minutes and seconds.
DATE	Dates between January 1, 1000 A.D. and December 31, 9999 A.D.

Date and Time Options

A number of options determine how date and time values are handled. These options are listed in Table 2–6, " Date and Time Options" on page 2-5.

Table 2–6 Date and Time Options

Statement	Description
CALENDARWEEK	Determines whether weeks should be aligned with the actual calendar year.
DATEFORMAT	Specifies the template used for displaying DATE values and converting DATE values to TEXT values.
DATEORDER	Contains three characters that indicate the intended order of the month, day, and year components of the DATE values in a workspace for those cases in which their interpretation is ambiguous.
DAYABBRLEN	Specifies the number of characters to use for abbreviations of day names that are stored in the DAYNAMES option.
DAYNAMES	A list of valid names for the days of the week. The names are used to display values of type DATE or to convert DATE values to text.

Table 2–6 (Cont.) Date and Time Options

Statement	Description
DSECONDS	(Read-only) The number of seconds since January 1, 1970.
MONTHABBRLEN	The number of characters to use for abbreviations of month names that are stored in the MONTHNAMES option.
MONTHNAMES	The list of valid names for months that is used in handling values with a `DATE` data type and values of dimensions of type DAY, WEEK, MONTH, QUARTER, and YEAR.
SECONDS	(Read-only) The number of seconds since January 1, 1970.
WEEKDAYSNEWYEAR	For a dimension of type WEEK, determines how many days of the new year there must be for a week to be identified as week 1 of the new year.
YRABSTART	The specific 100-year period associated with years that are read or displayed using a two-digit abbreviation.

DATE Values

`DATE` values have independent input and output formats. You can enter date values in one style and report them in a different style. To change the order of the month, day, and year components, see the DATEORDER option. When you show a date value in output, the format depends on the DATEFORMAT option. The default format is a 2-digit day, a 3-letter month, and a 2-digit year; for example, `03MAR97`. The text for the month names depends on the MONTHNAMES option.

DATETIME Values

The format and language of `DATETIME` values are controlled by the settings of the NLS_DATE_FORMAT and NLS_DATE_LANGUAGE options. The `DATETIME` data type is supported by Oracle Database standard libraries and operates the same way in the OLAP DML as it does in SQL. The DATEORDER, DATEFORMAT, and MONTHNAMES options, which control the formatting of `DATE` values, have no effect on `DATETIME` values. However, `DATETIME` and `DATE` values can be used interchangeably in most DML statements.

Calculating Dates

You can add numbers to a `DATE` or `DATETIME` value, or subtract numbers from them. Whole numbers are calculated as days, and decimal values are calculated as fractions of a day. For example, `SYSDATE+1.5` adds 1 day and 12 hours to the current date and time. You cannot divide or multiply `DATE` or `DATETIME` values, and you cannot subtract them from numbers. For example, `1-SYSDATE` and `1*SYSDATE` return errors.

Converting from One Data Type to Another

In many cases, Oracle OLAP performs automatic data type conversion for you.

- Oracle OLAP automatically converts `NTEXT` values to `TEXT` when they are specified as arguments to OLAP DML statements. This can result in data loss when the `NTEXT` values cannot be represented in the database character set.
- Oracle OLAP automatically converts `SHORTINTEGER` variables, as well as `INTEGER` variables with a fixed width of 1 byte, to `INTEGER` (with a width of 4 bytes) for calculations. When you calculate a total of `SHORTINTEGER` variables, then you can obtain and report a result greater than 32,767 or less than -32,768. When you calculate a total of 1-byte `INTEGER` variables, then you can obtain and report a result greater than 127 or less than -128. However, when you try to assign the result to a `SHORTINTEGER` variable or a 1-byte `INTEGER` variable respectively, then the variable is set to `NA`.

There are a number of OLAP DML functions that you can use to convert values from one data type to another. See Table A–2, " Data Type Conversion Functions" on page A-3 for a list of these functions.

OLAP DML Operators

An operator is a symbol that transforms a value or combines it in some way with another value. Table 2–7, " OLAP DML Operators" describes the categories of OLAP DML operators.

Table 2–7 OLAP DML Operators

Category	Description
Arithmetic	Operators that you can use in numeric expressions with numeric data to produce a numeric result. You can also use some arithmetic operators in date expressions with a mix of date and numeric data, which returns either a date or numeric result. For a list of arithmetic operators, see "Arithmetic Operators" on page 2-8. For more information on numeric expressions, see "Numeric Expressions" on page 3-11
Assignment	An operator that you use to assign the results of an expression into an object or to assign a value to an OLAP DML option. For more information on using assignment statements, see "Assignment Operator" on page 2-9 and SET.

Table 2–7 (Cont.) OLAP DML Operators

Category	Description
Comparison	Operators that you can use to compare two values of the same basic type (numeric, text, date, or, in rare cases, Boolean), which returns a `BOOLEAN` result. For a list of comparison operators, see "Comparison and Logical Operators" on page 2-9. For more information on `BOOLEAN` expressions, see "Boolean Expressions" on page 3-16.
Conditional	The IF...THEN...ELSE operators that you can use to select one of two values based on a `BOOLEAN` condition. For more information on the conditional operator, see Table 2–9, " Comparison and Logical Operators". For more information on conditional expressions, see "Conditional Expressions" on page 3-24.
Logical	Operators that you can use to transform `BOOLEAN` values using logical operations, which returns a `BOOLEAN` result. For a list of logical operators, see "Comparison and Logical Operators" on page 2-9. For more information on `BOOLEAN` expressions, see "Boolean Expressions" on page 3-16.
Substitution	The `&` (ampersand) operator that you can use to evaluate an expression and substitute the resulting value. For more information on the substitution operator, see "Substitution Expressions" on page 3-25.

Arithmetic Operators

Table 2–8, " Arithmetic Operators" shows the OLAP DML arithmetic operators, their operations, and priority where priority is the order in which that operator is evaluated. Operators of the same priority are evaluated from left to right. When you use two or more operators in a numeric expression, the expression is evaluated according to standard rules of arithmetic.

Table 2–8 Arithmetic Operators

Operator	Operation	Priority
-	Sign reversal	1
**	Exponentiation	2
*	Multiplication	3
/	Division	3
*	Addition	4
-	Subtraction	4

Note: A comma is required before a negative number that follows another numeric expression, or the minus sign is interpreted as a subtraction operator. For example, `intvar,-4`.

Comparison and Logical Operators

Table 2–9, " Comparison and Logical Operators" shows the OLAP DML comparison operators and logical operators, the operations, example, and priority where priority is the order in which that operator is evaluated. Operators of the same priority are evaluated from left to right. You use these operators to make expressions in much the same way as arithmetic operators. Each operator has a priority that determines its order of evaluation. Operators of equal priority are evaluated left to right, unless parentheses change the order of evaluation. However, the evaluation is halted when the truth value is already decided.

Table 2–9 Comparison and Logical Operators

Operator	Operation	Example	Priority
`NOT`	Returns opposite of `BOOLEAN` expression	`NOT(YES) = NO`	1
`EQ`	Equal to	`4 EQ 4 = YES`	2
`NE`	Not equal to	`5 NE 2 = YES`	2
`GT`	Greater than	`5 GT 7 = NO`	2
`LT`	Less than	`5 LT 7 = YES`	2
`GE`	Greater than or equal to	`8 GE 8 = YES`	2
`LE`	Less than or equal to	`8 LE 9 = YES`	2
`IN`	Is a date in a time period?	`'1Jan02' IN w1.02 = YES`	2
`LIKE`	Does a text value match a specified text pattern?	`'Finance' LIKE '%nan%' = YES`	2
`AND`	Both expressions are true	`8 GE 8 AND 5 LT 7 = YES`	3
`OR`	Either expression is true	`8 GE 8 OR 5 GT 7 = YES`	4

Assignment Operator

In the OLAP DML, as in many other programming languages, the = (equal) sign is used as an assignment operator.

An expression creates temporary data; you can display the resulting values, but these values are not automatically saved in your analytic workspace. When you want to save the result of an expression, then you store it in an object that has the same data type and dimensions as the expression. You use an assignment statement to store the value that is the result of the expression in the object.

Like other programming languages, an assignment statement in the OLAP DML sets the value of the target expression equal to the results of the source expression. However, an OLAP DML assignment statement does not work exactly as it does in other programming languages. Like many other OLAP DML statements it does not assign a value to a single cell, instead, when the target-expression is a multidimensional object, Oracle OLAP loops through the cells of the target object setting each one to the results of the source-expression. Additionally, you can use UNRAVEL to copy the values of an expression into the cells of a target object when the dimensions of the expression are not the same as the dimensions of the target object.

For more information on using assignment statements in the OLAP DML, see SET.

3 Expressions

Expressions represent data values in the syntax of the OLAP DML. This chapter explains how to create and use OLAP DML expressions. It includes the following topics:

Introducing OLAP DML Expressions

Expressions represent data values in the syntax of the OLAP DML. An expression has a data type and can also have dimensions. You can use expressions as arguments in statements. An expression often performs a mathematical or logical operation. It always evaluates to a result in one of the workspace data types.

An expression can be:

- A literal value. For example, `10` or `'East'`
- An analytic workspace object that contains multiple values. For example, the variable `sales`
- A function that returns one or more values. For example, TOTAL or JOINLINES
- Another expression that combines literal values, dimensions, variables, formulas, and functions with operators. For example, `inflation*1.02`

You can save an expression as a formula. See "Formulas" on page 4-1 for more information.

How the Data Type of an Expression is Determined

The data type of an expression is the data type of the resulting value. It might not be the same as the data type of the data objects that make up the expression; it depends on the data and on the operators and functions that are involved.

In addition, a conditional expression that is indicated by an `IF`... `THEN. . . ELSE` operator is supported. A conditional expression returns a value whose data type depends on the expressions in the `THEN` and `ELSE` clauses, not on the expression in the `IF` clause, which must be `BOOLEAN`.

Note: Do not confuse a conditional expression with the IF...THEN...ELSE statement in a program, which has similar syntax but a different purpose. The `IF` statement does not have a data type and is not evaluated like an expression.

Changing the Data Type of an Expression

You can use the CONVERT function to change data type of an expression. For example, you can convert a number to text, or you can convert a text string that consists of digits to a number.

However, there is no need to convert data to another type within the same basic category because those conversions are made automatically. In general, you can use `TEXT`, `NTEXT`, or `ID` data anywhere text is called for, and you can use integers and decimal numbers interchangeably.

Using Workspace Objects in Expressions

You can use an analytic workspace data object in an expression by specifying its name as described in "Syntax for Specifying an Object in an Expression" on page 3-3. When calculating the expression, Oracle OLAP uses the data in the specified object as described in "How Objects Behave in Expressions" on page 3-6.

Syntax for Specifying an Object in an Expression

You can specify an analytic workspace object in an expression using the following syntax.

[[*schema-name*.]*analytic-workspace-name*!]*object-name*

where:

- *schema-name* is the name of the schema in which the analytic workspace was defined when it was created. By default, a workspace is created in the schema for the database user ID of the user issuing the AW CREATE statement. In almost any DML statement, you can specify the full name of a workspace (for example, `Scott.demo`). When the workspace is in your schema, you can specify only the name (for example, `demo`) instead.
- *analytic-workspace-name* is the name of the workspace that contains the desired object. By specify the analytic workspace name along with the object name you create a **qualified object name** (QON) for the object. Using a QON for an object is recommended except in those sitatuations described in "When Not to Use Qualified Object Names" on page 3-5.

  You can specify the value for *analytic-workspace-name* in any of the following ways:

  - The name of an analytic workspace. A **workspace name** is assigned when a workspace is created with an AW CREATE statement.
  - The alias name of an analytic workspace. An **analytic workspace alias** is an alternative name for an attached analytic workspace. You can assign or delete an alias with an AW ALIASLIST statement. An alias is in effect from the time it is assigned to the time that the workspace is detached (or until the alias is deleted). Therefore, each time you attach an unattached workspace, you must reassign its aliases.

    One reason for assigning an alias is to have a short way to reference a workspace that belongs to a schema that is not yours. For example, you can use the alias in qualified object names and statements that reference such a workspace. Another reason for assigning an alias is to write generic code that includes a reference to a workspace but does not hard-code its name. With the alias providing a generic reference, you can assign the alias and run the code on different workspaces at different times.

- Within anm aggregation specification, model, or program, you can use **`THIS_AW`** to qualify an object name. When Oracle OLAP compiles an object, it interprets any occurrence of `THIS_AW` as the name of the workspace in which the object is being compiled. Thus if you have a workspace named `myworkspace` that contains a program named `myprog` and a variable named `myvar`, Oracle OLAP interprets a statement myvar=1 as though it was written `myworkspace!myvar=1`. Within a program, you can retrieve the value of `THIS_AW` using the THIS_AW option.

When you do not specify a value for *analytic-workspace-name*, Oracle OLAP assumes that the specified object is in the urrent analytic workspace. The **current analytic workspace** is the first analytic workspace in the list of the active analytic workspaces that you view with an AW LIST statement. You can retrieve the name of the current analytic workspace by using the AW function with the NAME keyword.

Note: Your session does not have to have a current analytic workspace. When you start Oracle OLAP without specifying an analytic workspace name, then the `EXPRESS` analytic workspace is first on the list. However, in this case, the `EXPRESS` analytic workspace is not current; there is no current analytic workspace until you specify one with the AW command.

- *object-name* is the name of the object.

Objects with the same name in different workspaces are treated as completely separate objects, and no similarity or relationship is assumed to exist between them. Any OLAP DML language restrictions that apply between objects in different workspaces apply even when the objects have the same name. For example, you cannot dimension an object in one workspace by a dimension that resides in another workspace, even when both workspaces have dimensions with the same name.

Considerations When Creating and Using Qualified Object Names

Although the use of qualified object names for objects is typical, there are a number of considerations to keep in mind:

- There are some situations where you cannot use a qualified object name or do not need to use a qualified object name. See "When Not to Use Qualified Object Names" on page 3-5 for more information
- Before you use ampersand substitution when creating a qualified object name you need to understand how and when the substitution occurs. See "Using Ampersand Substitution for Workspace and Object Names" on page 3-6 for more information.
- Special considerations apply when passing a qualified object name as an argument to a program. See "Passing Qualified Object Names to Programs" on page 3-6 for more information.

When Not to Use Qualified Object Names Generally it is good practice to use a qualified object name in an expression. However, there are some situations where you cannot use a qualified object name or when a qualified object name is not necessary:

- The following objects cannot have qualified object names:
  - An object that is local to a particular program because it was created by the ARGUMENT or VARIABLE command.
  - The NAME dimension of any given workspace. When you reference the NAME dimension, Oracle OLAP always uses the NAME dimension of the current workspace.
- You do not need to use a qualified object name in the following circumstances:
  - In the qualifiers of a qualified data reference (QDR). Only the object being qualified needs to be named with a qualified object name. Any unqualified names are assumed to apply to objects in the same workspace as the object being qualified.
  - In an unnamed composite, when you specify one base dimension as a qualified object name, then all the others are assumed to come from the same workspace.
  - In a named composite, when the name is a qualified object name then its base dimensions are assumed to come from the same workspace.

- In a model, when you specify the solution variable as a qualified object name, then all the dimensions named in DIMENSION (in models) statements are assumed to come from the same workspace.

Using Ampersand Substitution for Workspace and Object Names The workspace name, or the object name, or both can be supplied using ampersand substitution. However, take care when using a qualified object name with ampersand substitution because Oracle OLAP parses the qualified object name (with its exclamation point) before it resolves the ampersand reference. For example, in the expression `&awname!objname`, the ampersand (`&`) applies to the entire qualified object name, not just to the workspace name.

Passing Qualified Object Names to Programs When you pass a qualified object name as an argument to a program and you use the ARGUMENT command and the ARCTAN2, ARGFR, and ARGS functions, the entire qualified object name is considered to be a single argument. Its component parts are not passed separately.

How Objects Behave in Expressions

Table 3–1 summarizes how Oracle OLAP uses the data in an object used as an argument in an expression.

Table 3–1 Objects in Expressions

Object	Use in Expressions
Variables	As an array of data. For example, as the target or source expression in an assignment statement as outlined in "Using Objects in Assignment Statements" on page 21-58.
Relations	As an array of data. For example, as the target or source expression in an assignment statement as outlined in "Using Objects in Assignment Statements" on page 21-58. ■ When you use a relation in a text expression, the relation value is referenced as a text value. The values of the related dimension that is contained in the relation are converted into text, and you can use these values in an expression. You can also compare a text literal to a relation. ■ When you use a relation in a numeric expression, the relation value is referenced by its position (an INTEGER) in its related dimension array. You can use this numeric value in an expression. The position number is based on the default status list of the dimension, not the current status list of the dimension.

Table 3–1 (Cont.) Objects in Expressions

Object	Use in Expressions
Dimensions	As a one-dimensional array of data. When you use a `TEXT` dimensionvalue in a numeric expression or compare values in a non-numeric dimension, Oracle OLAP uses the `INTEGER` position number of the value in the array (as based on the default status list) rather than the value itself. **Note:** A dimension cannot be the target of an assignment statement; add values to dimensions using MAINTAIN.
Composites	You can use a composite wherever you can use a dimension. When you refer to an unnamed composite in an expression , specify `SPARSE <`*`dimensions`*`...>;` for example, `SPARSE <product month>`.
Valuesets	As a list of dimension values.
Dimension surrogates	As a one-dimensional array. When you use a `TEXT` surrogate value in a numeric expression or compare values in a non-numeric surrogate, Oracle OLAP uses the `INTEGER` position number of the value in the array (as based on the default status list) rather than the value itself. **Note:** A surrogate cannot be a participant object in any argument in a DEFINE statement that defines another object.
Formulas	As a sub-expression or as an expression in a statement.
Programs	For a program that does not return a value, use the program name as you would an OLAP DML command. For a program that returns a value, invoke the program the same way you invoke an OLAP DML command— use the program name in then expression and enclose the program arguments, if any, in parentheses.

Using Variables in Expressions

In expressions, a variable is referenced as an array containing values of the specified data type.

When you assign values to a variable or when you use REPORT or another statement that loops over the dimensions of a variable, the values of the fastest-varying dimension of the variable vary first. For example, for the `opcosts` variable that is dimensioned by month and city, when you view the variable as REPORT output, you see the data for all months for the first city before you see any data for the second city. In this case, `month` is the fastest-varying dimension because its values change before those of `city`. When you write programs that loop over a multidimensional variable in this way, try to maximize performance by matching the fastest-varying dimension with the inner loop.

Note: When you use a variable as the solution variable in a model, the model executes most efficiently when the order of the dimensions in the definition of the solution variable matches the order of the dimensions in the `DIMENSION` commands in the model.

You can uniquely and completely select any item of data within a multidimensional variable by using a qualified data reference (QDR) to specify one value from each of the dimensions of the variable.

For example, when the `opcosts` variable is dimensioned by `month` and `city`, specifying `Jan02` for the `month` dimension and `Boston` for the `city` dimension uniquely specifies a single cell in the variable.

Using Variables Defined with Composites in Expressions

In most cases, when you use OLAP DML statements with variables that are defined with composites, the statements treat those variables as if they were defined with base dimensions:

- You can access a variable that is dimensioned by a composite by requesting any of the base dimension values.
- The values of a composite that are in status are determined by the status of the base dimensions of the composite. Composites are not dimensions, and therefore, they do not have any independent status.

When you use the REPORT command or any other statement that loops over a variable that uses a composite, the default behavior is to evaluate all the combinations of the values of the base dimensions of the composite that are in status. Any combinations that do not exist in the composite display NA for their associated data.

For example, the following statements create a report for the East region that shows the number of coupons issued for sportswear from January through March 2002. Since no coupons were issued in March 2002, the report displays NA in that column.

```
LIMIT month TO 'Jan02' 'Feb02' 'Mar02'
LIMIT market TO 'East'
LIMIT product TO 'Sportswear'
REPORT coupons

MARKET: EAST
               ------------COUPONS-------------
               -------------MONTH--------------
PRODUCT          Jan02      Feb02      Mar02
-------------- ---------- ---------- ----------
Sportswear          1,000      1,000         NA
```

However, for performance reasons, you can change the default looping behavior for statements such as REPORT, ROW, and the assignment statement (SET) so that they loop over the values in the composite rather than all of the base dimension values.

Dimensionality of OLAP DML Expressions

An expression is dimensioned by a union of the dimensions of all of the variables, dimensions, relations, formulas, qualified data references, and functions in the expression:

- Variables, relations, and formulas are dimensioned by the dimensions listed in the definition of the object.

  Example 1: When the `price` variable is dimensioned by `month` and `product`, then the expression `price * 1.2` is also dimensioned by `month` and `product`.

  Example 2: When the `units` variable is dimensioned by `month`, `product`, and `district`, then the expression `units * price` is dimensioned by `month`, `product`, and `district` (even though the dimensions of the `price` variable are `month` and `product` only).

- Qualified data references (QDRs) are dimensioned by all of the dimensions of the associated object, expect for the dimensions being qualified. (For more information about qualified data references, see "Specifying a Single Data Value in an Expression" on page 3-32.)

- The return values of most OLAP DML functions are, in most cases, dimensioned by the union of the dimensions of the input arguments. However, some functions (such as aggregation functions) have fewer dimensions than the input arguments. In these cases, the dimensionality of the return value is documented in the topic for the function in Part II, "Alphabetic Reference".

Note: Unless otherwise noted this manual, when you specify breakout dimensions or relations in an aggregation function, you change the dimensionality of the expression. The first dimension that you specify as a breakout dimension is the slowest varying and the last dimension that you specify is the fastest varying.

Determining the Dimensions of an Expression

You can find out the dimensions of an expression with the PARSE command and the INFO function. PARSE evaluates the text of an expression; the INFO indicates how the expression is interpreted.

This example illustrates the use of the DIMENSION keyword with the INFO function to retrieve the dimensions of the expression just analyzed by the PARSE command. Assume that you issue the following statement.

```
PARSE 'TOTAL(sales region)'
```

The statement produces the following output.

```
SHOW INFO(PARSE DIMENSION)
REGION
```

How Dimension Status Affects the Results of Expressions

The number of values an expression yields depends on the dimensions of the expression and the status of those dimensions. An expression yields one data value for each combination of dimension values in the current status. For example, when three dimension values are in status for `month`, and two for `product`, then the expression `price gt 100` results in six values (3 times 2).

Thus, to get the desired results, you must ensure that the dimensions of an expression are limited to the range of data you want to consider. In addition, you must consider any PERMIT statements that might limit access to the dimensions of the data.

When you want to specify a single value without changing the current status you can use a qualified data reference (QDR). Using a QDR, you can qualify a dimension (which enables you to specify one dimension value in an expression) or one or more dimensions of a variable or relation. For more information on dimension status, see "Working with Subsets of Data" on page 3-30; for more information on QDRs, see "Specifying a Single Data Value in an Expression" on page 3-32.

Changing the Dimensionality of an Expression

You can change the dimensionality of an expression or subexpression using the CHGDIMS function.

Numeric Expressions

A numeric expression evaluates to data with any of the numeric data types (that is, `INTEGER`, `SHORTINTEGER`, `DECIMAL`, `SHORTDECIMAL, and` `NUMBER`). The data in a numeric expression can be any combination of the following:

- Numeric literals
- Numeric variables or formulas
- Dimensions
- Functions that yield numeric results
- Date literals, variables, formulas, or functions

In addition, you can join any of these three-part expressions with the arithmetic operators for a more complex numeric expression. You use arithmetic operators in numeric expressions with numeric data, which returns a numeric result. You can also use some arithmetic operators in date expressions with a mix of date and numeric data, to retrieve either a date or numeric result.

Numeric Options

A number of options determine how Oracle OLAP handles numeric expressions. These options are listed in Table 3–2, " Numeric Options" on page 3-11.

Table 3–2 Numeric Options

Option	Description
DECIMALOVERFLOW	Controls the result of arithmetic operations that produce out-of-range numbers. Decimal numbers are stored as a mantissa and an exponent. Decimal overflow occurs when the result of a calculation is very large and can no longer be represented by the exponent portion of the decimal representation.
DIVIDEBYZERO	Controls the result of division by zero.

Table 3–2 (Cont.) Numeric Options

Option	Description
RANDOM.SEED.1 and RANDOM.SEED.2	(Set only) Options that specify values used by RANDOM when computing random numbers. Typically, you only set values for these options when you are developing and debugging your application programs.
ROOTOFNEGATIVE	A flag that allows or disallows any attempt to obtain a root of a negative number.

Mixing Numeric Data Types

You can include any type of numeric data in the same numeric expression.

The data type of the result is determined according to the following rules:

- When all the data in the expression is `INTEGER` or `SHORTINTEGER`, and the only operations are addition, subtraction, and multiplication, then the result is `INTEGER`.
- When any of the data is `NUMBER`, then the result is `NUMBER`.
- When any of the data is `DECIMAL` or `SHORTDECIMAL`, and no data is `NUMBER`, then the result is `DECIMAL`.
- When you perform any division or exponentiation operations, then the result is `DECIMAL`.

Automatic Conversion of Numeric Data Types

Oracle OLAP automatically converts numeric data types according to the following rules:

- When you use a value with the `SHORTINTEGER` or `SHORTDECIMAL` data type in an expression, then the value is converted to its long counterpart before using it. See "Boolean Expressions" on page 3-16 for information about problems that can occur when you mix `SHORTDECIMAL` and `DECIMAL` data types in a comparison expression.
- When you save the results of a calculation as a value with the `SHORTINTEGER` data type, then NA is stored when the result is outside the range of a `SHORTINTEGER` (-32768 to 32767).

- When you assign the value of a `DECIMAL` expression to an object with the `INTEGER` data type, then the value is rounded before storing or using it.

> **Note:** When the decimal value is outside the range of an integer (approximately plus or minus 2 billion), then an NA is stored.

- When you use a decimal value where a value with the `INTEGER` data type is required, then the value is rounded before storing or using it.

> **Note:** When the decimal value is outside the range of an integer (approximately plus or minus 2 billion), then an NA is stored.

- When you assign the value of a decimal expression to a variable with the `SHORTDECIMAL` data type, then only the first 7 significant digits are stored.
- When you combine `NUMBER` values with other numeric data types, then all values are converted to `NUMBER`.

When these conversions are not what you want, then you can use the CONVERT, `TO_CHAR`, `TO_NCHAR`, `TO_NUMBER`, or `TO_DATE` functions to get different results.

Using Dimensions in Arithmetic Expressions

When you use a dimension with a data type of `TEXT` in a numeric expression, the dimension value is treated as a position (an `INTEGER`) and is used numerically. The position number is based on the default status list, not on current status.

Using Dates in Arithmetic Expressions

When you use dates in arithmetic expressions, the result can be numeric or it can be a date. The legal operations for dates and the data type of the result are outlined in Table 3–3, " Legal Operations for Dates" on page 3-13.

Table 3–3 Legal Operations for Dates

Operation	Result
Add or subtract a number from a date	Future or prior date
Subtract a date from a date	The number of days between the dates.

Table 3–3 (Cont.) Legal Operations for Dates

Operation	**Result**
Add or subtract a number from a time period.	The time period at the appropriate interval in the future or the past, similar to the return values of the LEAD or LAG function. The result is NA when there is no dimension value that corresponds to the result. The calculation is made based on the positions of the values in the default status list of the dimension.

Limitations of Floating Point Calculations

All decimal data is converted to floating point format, both for storing and for calculations. In floating point format, a number is represented by means of a mantissa and an exponent. The mantissa and the exponent are stored as binary numbers. The mantissa is a binary fraction which, when multiplied by a number equal to 2 raised to the exponent, produces a number that equals or closely approximates the original decimal number.

Because there is not always an exact binary representation for a fractional decimal number, just as there is not an exact representation for the decimal value of 1/3, fractional parts of decimal numbers cannot always be represented exactly as binary fractions. Arithmetic operations on floating point numbers can result in further approximations, and the inaccuracy gradually increases with the number of operations. In addition to the approximation factor, the available number of significant digits affects the exactness of the result.

For all of these reasons, a result computed by the TOTAL, AVERAGE, or other aggregation functions on a `DECIMAL` or `SHORTDECIMAL` variable can differ in the least significant digits from a result you compute by hand. Because the `SHORTDECIMAL` data type provides a maximum of only seven significant digits, you see more of these differences with `SHORTDECIMAL` data. Therefore, you might want to use the `NUMBER` data type when accuracy is more important than computational speed, such as variables that contain currency amounts.

Another result of the fact that some fractional decimal numbers cannot be exactly represented by binary fractions is that for such numbers, the `DECIMAL` data type offers a different and closer approximation than the `SHORTDECIMAL` data type, because it has more significant digits. This can lead to problems when `SHORTDECIMAL` and `DECIMAL` data types are mixed in a comparison expression. For information on how to handle such comparisons, see "Boolean Expressions" on page 3-16.

Controlling Errors During Calculations

You can control the following types of errors:

- Division by zero. When you divide an NA value by zero, then the result is NA; no error occurs. Dividing a non-NA value by zero normally produces an error. When a divide-by-zero error occurs when you are making a calculation on dimensioned data, then you can end up with partial results. When you use REPORT or an assignment statement (SET), values are reported or stored as they are calculated, so the division by zero halts the loop before it has gone through all the values.

  When you want to suppress the divide-by-zero error, then you can change the value of the DIVIDEBYZERO option to `YES`. This means that the result of any division by zero is NA and no error occurs. This allows the calculation of the other values of a dimensioned expression to continue.

- Root of negative numbers. It is normally an error to try to take the root of a negative number (which includes raising a number to a non-integer power). When you want to suppress the error message and allow the calculation of roots for non-negative values of the expression to continue, then set the ROOTOFNEGATIVE option to `YES`.
- Overflow errors. The DECIMALOVERFLOW option works in a similar manner to DIVIDEBYZERO. It lets you control whether an error is generated when a calculation produces a decimal result larger than it can handle.

Text Expressions

A text expression evaluates to data with the `TEXT`, `NTEXT`, or `ID` data type. Text expressions can be any combination of the following:

- Text literals. For example, `'Boston'` or `'Current Sales Report'`
- Text dimensions. For example, `district` or `month`
- Text variables or formulas. For example, `product.name`
- Functions that yield text results. For example, `JOINLINES('Product: ' product.name)`

Working with Dates in Text Expressions

When you use a DATETIME value where a text value (TEXT, NTEXT, or ID) is expected, or when you store a DATETIME value in a text variable, then the DATETIME value is automatically converted to a text value.

The format of a DATETIME value is controlled by the NLS_DATE_FORMAT option. Once a DATETIME value is stored in a text variable, the NLS_DATE_FORMAT setting has no impact.

Working with NTEXT Data

TEXT and NTEXT data are interchangeable in most cases. However, implicit conversion can occur, such as when an NTEXT value is assigned to a TEXT variable. When TEXT is converted to NTEXT, no data loss occurs because the UTF-8 character encoding of the NTEXT data type encompasses most other data types. However, when NTEXT is converted to TEXT, data loss occurs when NTEXT characters are not represented in the workspace character set.

When TEXT and NTEXT values are used together, for example in a call to the JOINCHARS function, the TEXT value is converted to NTEXT and an NTEXT value is returned.

Boolean Expressions

A Boolean expression is a logical statement that is either TRUE or FALSE. Boolean expressions can compare data of any type as long as both parts of the expression have the same basic data type. You can test data to see if it is equal to, greater than, or less than other data.

A Boolean expression can consist of Boolean data, such as the following:

- BOOLEAN values (YES and NO, and their synonyms, ON and OFF, and TRUE and FALSE)
- BOOLEAN variables or formulas
- Functions that yield BOOLEAN results
- BOOLEAN values calculated by comparison operators

For example, assume that your code contains the following Boolean expression.

```
actual GT 20000
```

When processing this expression, Oracle OLAP compares each value of the variable `actual` to the constant 20,000. When the value is greater than 20,000, then the statement is `TRUE`; when the value is less than or equal to 20,000, then the statement is `FALSE`.

When you are supplying a Boolean value, you can type either `YES`, `ON`, or `TRUE` for a true value, and `NO`, `OFF`, or `FALSE` for a false value. When the result of a Boolean calculation is produced, the defaults are `YES` and `NO` in the language specified by the NLS_LANGUAGE option. The read-only YESSPELL and NOSPELL options record the `YES` and `NO` values.

Table 2–9, " Comparison and Logical Operators" shows the comparison and logical operators. Each operator has a priority that determines its order of evaluation. Operators of equal priority are evaluated left to right, unless parentheses change the order of evaluation. However, the evaluation is halted when the truth value is already decided. For example, in the following expression, the TOTAL function is never executed because the first phrase determines that the whole expression is true.

```
yes EQ yes OR TOTAL(sales) GT 20000
```

Creating Boolean Expressions

A Boolean expression is a three-part clause that consists of two items to be compared, separated by a comparison operator. You can create a more complex Boolean expression by joining any of these three-part expressions with the `AND` and `OR` logical operators. Each expression that is connected by `AND` or `OR` must be a complete Boolean expression in itself, even when it means specifying the same variable several times.

For example, the following expression is not valid because the second part is incomplete.

```
sales GT 50000 AND LE 20000
```

In the next expression, both parts are complete so the expression is valid.

```
sales GT 50000 AND sales LE 20000
```

When you combine several Boolean expressions, the whole expression must be valid even when the truth value can be determined by the first part of the expression. The whole expression is compiled before it is evaluated, so when there are undefined variables in the second part of a Boolean expression, you get an error.

Use the NOT operator, with parentheses around the expression, to reverse the sense of a Boolean expression.

The following two expressions are equivalent.

```
district NE 'BOSTON'
NOT(district EQ 'BOSTON')
```

Example 3–1 Using Boolean Comparisons

The following example shows a report that displays whether sales in Boston for each product were greater than a literal amount.

```
LIMIT time TO FIRST 2
LIMIT geography TO 'BOSTON'
REPORT DOWN product ACROSS time: f.sales GT 7500
```

This REPORT statement returns the following data.

```
CHANNEL: TOTALCHANNEL
GEOGRAPHY: BOSTON
               ---F.SALES GT 7500---
               --------TIME---------
PRODUCT          Jan02      Feb02
-------------- ---------- ----------
Portaudio              NO         NO
Audiocomp             YES        YES
TV                     NO         NO
VCR                    NO         NO
Camcorder             YES        YES
Audiotape              NO         NO
Videotape             YES        YES
```

Comparing NA Values in Boolean Expressions

When the data you are comparing in a Boolean expression involves an NA value, a YES or NO result is returned when that makes sense. For example, when you test whether an NA value is equal to a non-NA value, then the result is NO. However, when the result would be misleading, then NA is returned. For example, testing whether an NA value is less than or greater than a non–NA value gives a result of NA.

Table 3–4, " Boolean Expressions with NA Values that Result in non-NA Values" shows the results of Boolean expressions involving NA values, which yield non-NA values.

Table 3–4 Boolean Expressions with NA Values that Result in non-NA Values

Expressions	Result
`NA EQ NA`	`YES`
`NA NE NA`	`NO`
`NA EQ non-NA`	`NO`
`NA NE non-NA`	`YES`
`NA AND NO`	`NO`
`NA OR YES`	`YES`

Controlling Errors When Comparing Numeric Data

When you get unexpected results when comparing numeric data, then there are several possible causes to consider:

- One of the numbers you are comparing might have a small decimal part that does not show in output because of the setting of the DECIMALS option.
- You are comparing two floating point numbers and at least one number is the result of an arithmetic operation.
- You have mixed `SHORTDECIMAL` and `DECIMAL` data types in a comparison.

Oracle recommends that you use the ABS and ROUND functions to do approximate tests for equality and avoid all three causes of unexpected comparison failure. When using ABS or ROUND, you can adjust the absolute difference or the rounding factor to values you feel are appropriate for your application. When speed of calculation is important, then you probably want to use the ABS rather than the ROUND function.

Controlling Errors Due to Numerical Precision

Suppose `expense` is a decimal variable whose value is set by a calculation. When the result of the calculation is 100.000001 and the number of decimal places is two, then the value appears in output as 100.00. However, the output of the following statement returns `NO`.

```
SHOW expense EQ 100.00
```

You can use the `ABS` or the `ROUND` function to ignore these slight differences when making comparisons.

Controlling Errors When Comparing Floating Point Numbers

A standard restriction on the use of floating point numbers in a computer language is that you cannot expect exact equality in a comparison of two floating point numbers when either number is the result of an arithmetic operation. For example, on some systems, the following statement returns a `NO` instead of the expected `YES`.

```
SHOW .1 + .2 EQ .3
```

When you deal with decimal data, you should not code direct comparisons. Instead, you can use the ABS or the ROUND function to allow a tolerance for approximate equality. For example, either of the following two statements produce the desired `YES`.

```
SHOW ABS((.1 + .2) - .3) LT .00001
SHOW ROUND(.1 + .2) EQ ROUND(.3, .00001)
```

Controlling Errors When Comparing Different Numeric Data Types

You cannot expect exact equality between `SHORTDECIMAL` and `DECIMAL` or `NUMBER` representations of a decimal number with a fractional component, because the `DECIMAL` and `NUMBER` data types have more significant digits to approximate fractional components that cannot be represented exactly.

Suppose you define a variable with a `SHORTDECIMAL` data type and set it to a fractional decimal number, then compare the `SHORTDECIMAL` number to the fractional decimal number, as shown here.

```
DEFINE sdvar SHORTDECIMAL
sdvar = 1.3
SHOW sdvar EQ 1.3
```

The comparison is likely to return `NO`. What happens in this situation is that the literal is automatically typed as `DECIMAL` and converts the `SHORTDECIMAL` variable `sdvar` to `DECIMAL`, which extends the decimal places with zeros. A bit-by-bit comparison is then performed, which fails. The same comparison using a variable with a `DECIMAL` or a `NUMBER` data type is likely to return `YES`.

There are several ways to avoid this type of comparison failure:

- Do not mix the `SHORTDECIMAL` with `DECIMAL` or `NUMBER` types in comparisons. To avoid mixing these two data types, you should generally avoid defining variables with decimal components as `SHORTDECIMAL`.
- Use the `ABS` or `ROUND` function to allow for approximate equality. The following statements both produce `YES`.

  ```
  SHOW ABS(sdvar - 1.3) LT .00001
  SHOW ROUND(sdvar, .00001) EQ ROUND(.3, .00001)
  ```

Comparing Dimension Values

Values are not compared in the same dimension based on their textual values. Instead, Oracle OLAP compares the positions of the values in the default status of the dimension. This enables you to specify statements like the following statement.

```
REPORT district LT 'Seattle'
```

Statements are interpreted such as these using the following process:

1. The text literal `'Seattle'` is converted to its position in the `district` default status list of the dimension.
2. That position is compared to the position of all other values in the `district` dimension.
3. As shown by the following report, the value `YES` is returned for districts that are positioned before `Seattle` in the `district` default status list of the dimension, and `NO` for `Seattle` itself.

   ```
   REPORT 22 WIDTH district LT 'Seattle'

   District       DISTRICT LT 'Seattle'
   -------------- ----------------------
   Boston                            YES
   Atlanta                           YES
   Chicago                           YES
   Dallas                            YES
   Denver                            YES
   Seattle                            NO
   ```

A more complex example assigns increasing values to the variable `quota` based on initial values assigned to the first six months. The comparison depends on the position of the values in the `month` dimension. Because it is a time dimension, the values are in chronological order.

```
quota = IF month LE 'Jun02' THEN 100 ELSE LAG(quota, 1, month)* 1.15
```

However, when you compare values from different dimensions, such as in the expression `region lt district`, then the only common denominator is `TEXT`, and text values are compared, not dimension positions.

Comparing Dates

You can compare two dates with any of the Boolean comparison operators. For dates, "less" means before and "greater" means after. The expressions being compared can include any of the date calculations discussed in " Comparison and Logical Operators" on page 2-9. For example, in a billing application, you can determine whether today is 60 or more days after the billing date in order to send out a more strongly worded bill.

```
bill.date + 60 LE SYSDATE
```

Dates also have a numeric value. You can use the TO_NUMBER and TO_DATE functions to change dates to integers and integers to dates for comparison.

Comparing Text Data

When you compare text data, you must specify the text exactly as it appears, with punctuation, spaces, and uppercase or lowercase letters. A text literal must be enclosed in single quotes. For example, this expression tests whether the first letter of each employee's name is greater than the letter "M."

```
EXTCHARS(employee.name, 1, 1) GT 'M'
```

You can compare `TEXT` and `ID` values, but they can only be equal when they are the same length. When you test whether a text value is greater or less than another, the ordering is based on the setting of the NLS_SORT option.

You can compare numbers with text by first converting the number to text. Ordering is based on the values of the characters. This can produce unexpected results because the text is evaluated from left to right. For example, the text literal `1234` is greater than `100,999.00` because `2`, the second character in the first text literal, is greater than `0`, the second character in the second text literal.

Suppose `name.label` is an `ID` variable whose value is `3-Person` and `name.desc` is a `TEXT` variable whose value is `3-Person Tents`.

The result of the following SHOW statement is NO.

```
SHOW name.desc EQ name.label
```

The result of the following statements is YES.

```
name.desc = '3-Person'
SHOW name.desc EQ name.label
```

Comparing a Text Value to a Text Pattern

The Boolean operator LIKE is designed for comparing a text value to a text pattern. A text value is *like* another text value or pattern when corresponding characters match.

Besides literal matching, LIKE lets you use wildcard characters to match more than one character in a string:

- An underscore (_) character in a pattern matches any single character.
- A percent (%) character in a pattern matches zero or more characters in the first string.

For example, a pattern of %AT_ matches any text that contains zero or more characters, followed by the characters AT, followed by any other single character. Both DATA and ERRATA return YES when LIKE is used to compare them with the pattern %AT_.

The results of expressions using the LIKE operator are affected by the settings of the LIKECASE and LIKENL options.

No negation operator exists for LIKE. To accomplish negation, you must negate the entire expression. For example, the result of the following statement is NO.

```
SHOW NOT ('Boston' LIKE 'Bo%')
```

Comparing Text Literals to Relations

You can also compare a text literal to a relation. A relation contains values of the related dimension and the text literal is compared to a value of that dimension. For example, region.district holds values of region, so you can do the following comparison.

```
region.district EQ 'West'
```

Conditional Expressions

A conditional expression is an expression you can use to select one of two values based on a Boolean condition. A conditional expression contains the conditional operators `IF...THEN...ELSE` and has the following format.

```
IF Boolean-expression THEN expression1 ELSE expression2
```

You can use a conditional expression as part of any other expression as long as the data type is appropriate.

Note: Do not confuse a conditional expression with the `IF` command, which has similar syntax but a different purpose. The `IF` command does not have a data type and is not evaluated like an expression.

A conditional expression is processed by first evaluating the Boolean expression; then:

- When the result of the Boolean expression is `TRUE`, then *expression1* is evaluated and returns that value.
- When the result of the Boolean expression is `FALSE`, then *expression2* is evaluated and returns that value.

The *`expression1`* and *`expression2`* arguments are any valid OLAP DML expressions that evaluate to the same basic data type. However, when the data type of either value is `DATE`, it is possible for the other value to have a numeric or text data type. Because both data types are expected to be `DATE`, Oracle OLAP converts the numeric or text value to a `DATE`. The data type of the whole expression is the same as the two expressions.

When the result of the Boolean expression is `NA`, then `NA` is returned.

Example 3–2 Report with Conditional Expression

This example shows a sales bonus report. The bonus is 5 percent of the amount that sales exceeded budget, but when sales in the district are below budget, then the bonus is zero.

```
LIMIT month TO 'Jan02' TO 'Jun02'
LIMIT product TO 'Tents'
REPORT DOWN district IF sales-sales.plan LT 0 THEN 0
       ELSE .05*(sales-sales.plan)
```

```
PRODUCT: TENTS
        ---IF SALES-SALES.PLAN LT 0 THEN 0 ELSE .05*(SALES-SALES.PLAN)---
          ---------------------MONTH-----------------------------
DISTRICT   Jan02    Feb02    Mar02     Apr02    May02    Jun02
--------- -------- -------- -------- ------- --------- ----------
Boston      229.53     0.00     0.00    0.00    584.51     749.13
Atlanta       0.00     0.00     0.00  190.34    837.62   1,154.87
Chicago       0.00     0.00     0.00   84.06    504.95     786.81
...
```

Substitution Expressions

To construct a substitution expression, use an ampersand character (&) at the beginning of an expression. Using an ampersand (that is, the substitution operator) this way is also called ampersand substitution. The ampersand specifies that Oracle OLAP should evaluate an expression containing a substitution expression as follows:

1. Evaluate the expression following the ampersand (the substitution expression).
2. Evaluate the rest of the expression using the result of step 1 (that is, the result of the substitution expression).

Ampersand substitution gives you a level of indirection when you are specifying an expression. For example, when you specify an ampersand followed by a variable that holds the name of another variable, the value of the expression becomes the data in the second variable. Ampersand substitution lets you write more general programs that can operate on data that is chosen when the program is run.

You cannot use ampersand substitution in model equations.

Note: Although ampersand substitution lets you write general programs that can handle different variables and data, program lines that use ampersand substitution are executed less efficiently. Lines with ampersand substitution are not compiled; instead these lines are interpreted when the program runs. To avoid ampersand substitution, you can often use the IF or SWITCH command instead.

Example 3–3 Using Ampersand Substitution

Suppose you have a variable called `curname` that holds the name of one of the dimensions in the analytic workspace (`product`). When you execute the following statement, then REPORT produces the single value, `product`, which is the actual value stored in the `curname` variable.

```
REPORT curname

CURNAME
----------
PRODUCT
```

However, when you execute the following statement, then REPORT produces the values of the dimension `product`.

```
REPORT &curname

PRODUCT
--------------
Tents
Canoes
Racquets
Sportswear
Footwear
```

Working with Empty Cells in Expressions

At any given time, some of the cells of an analytic workspace data object may be empty. An empty cell occurs when a specific data value has not been assigned to it or when a data value cannot be calculated for the cell. The value of any empty cell in an object is `NA`. An `NA` value has no specific data type. Certain functions (for example, the aggregation functions) return an `NA` values when the information that is requested with the function is not available or cannot be calculated. Similarly, an expression whose value cannot be calculated has `NA` as its value.

Specifying a Value of NA

There are cases in which you might specify an operation for which no data is available. For example, there might be no appropriate value for a given cell in a variable, for the return value of a function, or for the value of an expression that includes an arithmetic operator. In these cases, an `NA` (Not Available) value is automatically supplied.

- To set the values of a variable or relation to NA, you can use an assignment statement (SET), as shown in the following example.

```
sales = NA
```

Controlling how NA values are treated

A number of options and functions control how NA values are treated. For example:

- The options listed in Table 3–5, "NA Value Options".
- The NAFILL function returns the values of the source expression with any NA values appearing as the specified fill expression. You can include this function in an expression to control the format of its value.
- System properties listed in Table 3–6, " System Properties Used When Working with NA Values" on page 3-28.

Table 3–5 NA Value Options

Statement	Description
NASKIP	An option that controls whether NA values are considered as input to aggregation functions.
NASKIP2	An option that controls how NA values are treated in arithmetic operations with the + (plus) and - (minus) operators.
NASPELL	An option that controls the spelling that is used for NA values in output.
RECURSIVE	An option that controls the ability of a formula or $NATRIGGER expression to call itself.
TRIGGERMAXDEPTH	An option that specifies the maximum number of $NATRIGGER property expressions that Oracle OLAP can execute simultaneously.
TRIGGERSTOREOK	An option that determines whether Oracle OLAP permanently replaces NA values in the cells of a variable with the value of the $NATRIGGER property expression that is set for the variable.

Table 3–6 System Properties Used When Working with NA Values

Property	Description
$NATRIGGER	A property that specifies values to substitute for NA values that are in the object, but not in the session cache for the object (if any).
$STORETRIGGERVAL	A property that specifies that NA values in an object be permanently replaced by the values specified by the $NATRIGGER property.
$VARCACHE	A property that specifies whether Oracle OLAP stores or caches variable data that is the result of the execution of a AGGREGATE function or $NATRIGGER expression.

Working with the $NATRIGGER Property

An $NATRIGGER property expression is evaluated before applying the NAFILL function or the NASKIP, NASKIP2, or NASPELL options. When the $NATRIGGER expression is NA, then the NAFILL function and the NA options have an effect. Additionally, the $NATRIGGER property allows you a good deal of flexibility about handling NA values:

- You can make $NATRIGGERs recursive or mutually recursive by including triggered objects within the value expression. You must set the RECURSIVE option to `YES` before a formula, program, or other $NATRIGGER expression can invoke a trigger expression again while it is executing. For limiting the number of triggers that can execute simultaneously, see the TRIGGERMAXDEPTH option.
- You can replace the NA value in the cells of the variable with the $NATRIGGER expression value by setting the TRIGGERSTOREOK option to `YES` and setting the $STORETRIGGERVAL property on the variable to `YES`.

The ROLLUP and AGGREGATE commands and the AGGREGATE function ignore the $NATRIGGER property setting for a variable during a rollup or aggregation operation. Additionally, the $NATRIGGER property expression on a variable is not evaluated when the variable is simply exported with an EXPORT TO EIF file command. The $NATRIGGER property expression is only evaluated when the variable is part of an expression that is calculated during the export operation.

Using NASKIP

The NASKIP option controls how NA values are treated in aggregation functions:

- By default, the NASKIP option is set to YES, and NA values are ignored by aggregation functions. Only expressions with actual values are used in calculations.
- When you set the NASKIP option to NO, then NA values are considered as input to aggregation functions. When any of the values being considered are NA, then the function returns NA for that value.

Setting NASKIP to NO is useful for cases in which having NA values in the data makes the calculation itself invalid. For example, when you use the MOVINGMAX function, you specify a range from which to select the maximum value.

- When NASKIP is YES (the default), then MOVINGMAX returns NA only when all the values in the range are NA.
- When NASKIP is NO and any value in the range is NA, then MOVINGMAX returns NA.

Using NASKIP2

The NASKIP2 option controls how NA values are treated in arithmetic operations with the addition (+) and subtraction (-) operators.

- By default, the value of the NASKIP2 option is NO. NA values are treated as NAs in arithmetic operations using the addition (+) and subtraction (-) operators. When any of the operands being considered is NA, then the arithmetic operation evaluates to NA. For example, by default, 2+NA results in NA.
- When you set the value of the NASKIP2 option to YES, then zeroes are substituted for NA values in arithmetic operations using the addition (+) and subtraction (-) operators. The two special cases of NA+ NA and NA-NA both result in NA.

Using NAFILL

NASKIP and NASKIP2 do not change your data. They only affect the results of calculations on your data. When you would prefer a more targeted influence on any kind of expressions, and want the option of making an actual change in your data, then you can use the NAFILL function.

The effect of the NAFILL function is limited to the single expression you specify. It can be any kind of expression, not just a function or an addition (+) or subtraction (-) operation. In addition, you can use NAFILL to substitute anything for the NAs in the expression, not just zeroes. Moreover, using an assignment statement (SET), you can use NAFILL to make a permanent substitution for NAs in your data.

NAFILL returns the value of a specified expression unless its value is NA, in which case NAFILL returns the substitute value you specify.

The following command uses NAFILL to replace the NA values in the sales variable with the number 1 and then assign those values to the variable. This makes the substitution permanent in your data.

```
sales = NAFILL(sales, 1)
```

The following command illustrates the use of NAFILL for more specialized purposes. By substituting zeros for NA values, NAFILL in this example forces the AVERAGE function to include NA values when it counts the number of values it is averaging. The substitution is temporary, lasting only for the duration of this command.z

```
SHOW AVERAGE(NAFILL(sales 0.0) district)
```

Working with Subsets of Data

In the OLAP DML, when you want to calculate against a subset of data, you can specify the desired subset in one of the following ways:

- Specify what dimension values (and, therefore, what variable values) are currently accessible or "in status" to all OLAP DML statements and expressions. For more information, see "Working with Dimension Status" on page 3-30.
- Within an expression, specify a single value or a subset of values. For more information, see "Specifying a List of Dimension Values for an Expression or Subexpression" on page 3-32 or "Specifying a Single Data Value in an Expression" on page 3-32.

Working with Dimension Status

The **current status list** of a dimension is an ordered list of currently accessible values for the dimension. Values that are in the current status list of a dimension are said to be "in status." The current status list of a dimension determines the selection of the data from all of the objects that are dimensioned by it.

For dimensions, only those dimension values that are in the current status list are accessed. For dimensioned objects, only those data values that are indexed by dimension values in the current status list are accessed. As a loop is performed through a dimensioned object, the order of the dimension values in the current status list is used to determine the order in which the values of the object are accessed.

Important: Whether or not a dimension value is in status merely restricts your view of the value during a given session; it does not permanently affect the values that are stored in the analytic workspace.

A dimension and any surrogate for that dimension share the same status. Setting the status of a dimension surrogate sets the status of its dimension and setting the status of a dimension sets the status of any dimension surrogates for it. In Part II of this manual, references to dimensions apply equally to dimension surrogates, except where noted. Also the phrase "setting status" includes assigning values to a valueset as well as setting the current status of a dimension. Composites are not dimensions, and therefore they do not have any independent status. The values of a composite that are "in status" are determined by the status of the base dimensions of the composite. In general, when statements deal with objects defined with composites, the default behavior is to treat those objects as if no SPARSE keyword or named composite had been used when the object was defined.

When you first attach an analytic workspace, the current status list of each dimension consists of all of the values of the dimension that have read permission, in the order in which the values are stored. This list of values is called the **default status list** for the dimension.

Changing the Status List of a Dimension

You can change the current status list for a dimension by using:

- The LIMIT command to change the values and the order of the values in the current status list of a dimension.
- The SORT command to arrange the order of values in the current status list of a dimension.
- The PERMIT command to change the read permissions for dimension values.

You can change the default status list of a dimension in the following ways:

- You can add, delete, move, merge, and rename values in a dimension by using the MAINTAIN command or adding dimension values in other ways (for example, using a SQL FETCH statement).
- You can change the read permission of values that are associated with a dimension by using a PERMIT or PERMITRESET statement.

The OLAP DML provides a number of statements that you can use to identify and retrieve the status of dimension values These statements are listed in Table A–19, " Dimension and Composite Operation Statements".

Saving and Restoring Current Dimension Status

You can save the current status of a dimension in the following ways:

- When you want to save the current status for use in any session, then use a named valueset. Use a DEFINE VALUESET statement to define the valueset.
- When you want to save the current status for use in the current program, then use the PUSHLEVEL and PUSH statements. You can restore the current status values using the POPLEVEL and POP statements.
- When you want to save, access, or update the current status for use in the current session, then use a named context. Use the CONTEXT command to define the context.

Specifying a List of Dimension Values for an Expression or Subexpression

Using the CHGDIMS function, you can limit one element of an expression to only those values that are dimensioned by the specified dimension values. Using the CHGDIMS function in this manner limits the dimension to the specified values for the calculation without the current status of the dimension.

Specifying a Single Data Value in an Expression

A qualified data reference (QDR) is a way of limiting one or more dimensions of an expression to a single value. QDRs are useful when you want to specify a single value without changing the current status. Using a QDR, you can qualify a dimension (which enables you to specify one dimension value in an expression) or one or more dimensions of a variable or relation.

Form of a Qualified Data Reference

A qualified data reference takes the following form.

expression(*dimname1 dimexp1* [, *dimname2 dimexp2*. . .])

The *dimname* argument is the name of one of the dimensions, or a dimension surrogate of the dimension, of the expression and the *dimexp* argument is one of the following:

- A value of *dimname.*
- A text expression whose result is a value of *dimname.*
- A numeric expression whose result is the logical position of a value of *dimname.*
- A relation of *dimname.*

Note: To qualify a complex expression, use the QUAL function.

Qualifying a Variable

You can qualify any or all of a dimensions of a variable using either of the following techniques:

- The QDR can temporarily limit a dimension of the variable by selecting one specified value of the dimension. This value can be outside the current status.
- The QDR can replace a dimension of the variable with a less aggregate related dimension when you supply the name of an appropriate relation as the qualifier. The dimension is temporarily replaced by the dimension(s) of the relation.

For example, the variable `sales` has three dimensions, `month`, `product`, and `district`. You might want to compare total sales in Boston to the total sales in all cities. In a single statement, you want `district` to be limited to *two* different values:

- For the numerator of the expression, you want the status of `district` to be `Boston`.
- For the denominator of the expression, you want the status of `district` to be `ALL`.

The following statement lets you calculate this result by using a QDR.

```
SHOW sales(district 'Boston')/TOTAL(sales)
```

You can qualify more than one of the dimensions of a variable. For example, when you qualify all the dimensions of the `sales` variable by specifying one dimension value of each dimension, then you narrow `sales` down to a single–cell value.

To fetch sales for `Jun02`, `Tents`, and `Seattle`, use the following QDR.

```
SHOW sales(month 'Jun02', product 'Tents', district 'Seattle')
```

This statement fetches a single value.

You can use a qualified data reference with the target expression of an assignment (SET) statement. This lets you assign a value to a specific cell in a data object.

The following example assigns the value 10200 to the data cell of the `sales` composite that is specified in the qualified data reference. When the composite named `sales` does not already have a value for the combination `Boston` and `Tents`, then this value combination is added to the composite, thus adding the data cell.

```
sales(market 'Boston' product 'Tents' month 'Jan99')= 10200
```

Replacing a Dimension in a Variable

When you use a relation as the qualifier in the QDR, you replace a dimension of the variable with the dimension or dimensions of the relation. The relation must be related to the dimension that you are qualifying, and it must be dimensioned by the replacement dimension.

Example 3–4 Replacing a Dimension in a Variable

Suppose you have two variables, `sales` and `quota`, which are dimensioned by `month`, `product`, and `district`. A third variable, `division.mgr`, is dimensioned by `month` and `division`. You also have a relation between `division` and `product`, called `division.product`. These objects have the following definitions.

```
DEFINE sales VARIABLE DECIMAL <month product district>
LD Sales Revenue
DEFINE quota VARIABLE DECIMAL <month product district>
DEFINE division.mgr VARIABLE TEXT <month division>
DEFINE division.product RELATION division <product>
LD Division for each product
```

The following statement produces the report following it.

```
REPORT division.mgr
```

```
-------------------DIVISION.MGR----------------------
```

```
          ---------------------MONTH-------------------------
DIVISION  JAn02    Feb02    Mar02    Apr02    May02    Jun02
--------  -------- -------- -------- -------- -------- --------
Camping   Hawley   Hawley   Jones    Jones    Jones    Jones
Sporting  Carey    Carey    Carey    Carey    Carey    Musgrave
Clothing  Musgrave Musgrave Musgrave Musgrave Musgrave Wong
```

Suppose you want to obtain a report that shows the fraction by which sales have exceeded quota and you want to include the appropriate division manager for each product. You can show the division manager for each product by using the relation `division.product`, which is related to `division` and dimensioned by `product`, as the qualifier. The QDR replaces the `division` dimension with `product`, so that it has the same dimensions as the other expression in the report `sales / quota`. The following statement produces the report following it.

```
REPORT DOWN month sales W 6 sales/quota W 8 HEADING -
   'MANAGER' division.mgr(division division.product)

DISTRICT: BOSTON
        ----------------------------PRODUCT-----------------------------------
        ----TEnts---- ---canoes---- --racquets---  --sportswear-- ---footwear---
        Sales/        Sales/        Sales/         Sales/         Sales/
Month   Quota Manager Quota Manager Quota Manager  Quota Manager  Quota  Manager
------  ----- ------- ----- ------- ----- -------  ----- -------- ----- --------
Jan02   1.00  Hawley  0.82  Hawley  1.02  Carey    0.91  Musgrave 0.92  Musgrave
Feb02   0.84  Hawley  0.96  Hawley  1.00  Carey    0.80  Musgrave 1.07  Musgrave
Mar02   0.87  Jones   0.95  Jones   0.87  Carey    0.88  Musgrave 0.91  Musgrave
Apr02   0.91  Jones   0.93  Jones   0.99  Carey    0.94  Musgrave 0.95  Musgrave
...
```

Qualifying a Relation

You can also use a QDR to qualify a relation (which is really a special kind of variable).

Suppose the `region.district` relation is dimensioned by `district`. When you qualify `district` with the value `Seattle`, then the value of the expression is the value of the relation for `Seattle`. Because the QDR specifies one value of `district`, the expression has a single–cell result.

The definition of `region.district` is as follows.

```
DEFINE region.district RELATION region <district>
LD The region for each district
```

The following statement displays the value WEST.

```
SHOW region.district(district 'Seattle')
```

Qualifying a Dimension

You can use a QDR to qualify the dimension itself, which enables you to specify one dimension value in an expression. The following expression specifies one value of district, the one contained in the single-cell variable mydistrict.

```
district(district mydistrict)
```

For a concat dimension, you can use a QDR to qualify the dimension by specifying a value from one of the base dimensions of the concat dimension. The following expression specifies one value of reg.dist.ccdim, a concat dimension that has region and district as its base dimensions. The costs variable is dimensioned by the division and reg.dist.ccdim dimensions.

```
SHOW reg.dist.ccdim(district 'Boston')
```

The preceding expression produces the following result.

```
<DISTRICT: Boston>
```

Using Ampersand Substitution with QDRs

An ampersand character (&) at the beginning of an expression substitutes the value of the expression for the expression itself in a statement.When you use an ampersand with a QDR, you must enclose the whole expression in parentheses when you want the variable to be qualified before the substitution is made.

Suppose you have a text variable named myvar that is dimensioned by reptype and that contains the names of variables. Remember that it is myvar that is dimensioned by reptype, not the variables named by myvar. Therefore, you must use parentheses so that myvar is qualified and the resulting value is used in the REPORT command.

```
REPORT &(myvar(reptype 'actual'))
```

When you do not use parentheses and the variable that is specified in myvar is sales, then you get an error message that sales is not dimensioned by reptype.

Using the QUAL Function to Specify a QDR

Sometimes you the syntax of a QDR is ambiguous and could either be misinterpreted or cause a syntax error. In this case, you can use the QUAL function to explicitly specify a qualified data reference (QDR).

Example 3–5 Using the QUAL Function

The following example first shows how you might view your data by limiting its dimensions, and then how you might view it by using QUAL.

Assume that you want to create a report of Cogs line items in the Sporting division from January 1996 through June 1996 with columns for month, the maximum value of either actual costs or budgeted costs or `MAX(actual,budget)`, actual costs for the month, and budgeted amount for the month. To create this report you can issue three LIMIT statements (one each for `month`, `line`, and `division`) and a REPORT statement.

```
LIMIT month TO 'Jan96' to 'Jun96'
LIMIT line TO 'Cogs'
LIMIT division TO 'Sporting'
REPORT DOWN month W 11 MAX(actual,budget) W 11 actual W 11 budget

DIVISION: SPORTING
               ---------------LINE----------------
               ---------------COGS----------------
               MAX(ACTUAL,
MONTH            budget)     actual      budget
-------------- ----------- ----------- -----------
Jan96           287,557.87  287,557.87  279,773.01
Feb96           323,981.56  315,298.82  323,981.56
Mar96           326,184.87  326,184.87  302,177.88
Apr96           394,544.27  394,544.27  386,100.82
May96           449,862.25  449,862.25  433,997.89
Jun96           457,347.55  457,347.55  448,042.45
```

Now assume that you want a report on the same items and the same time period, but with only two columns: one for month and another for `MAX(actual,budget)`. In this case, you can issue merely one LIMIT statement for `month` and use the QUAL function in your REPORT statement to limit calculation to Cogs line items in the Sporting division.

```
LIMIT month TO 'Jan96' to 'Jun96'
REPORT HEADING 'For Cogs in Sporting Division' DOWN month -
   W 11 HEADING 'MAX(actual,budget)'-
   QUAL(MAX(actual,budget), line 'COGS', division 'SPORTING')

For Cogs in
Sporting       MAX(ACTUAL,
Division         BUDGET)
-------------- -----------
JAN96           287,557.87
FEB96           323,981.56
MAR96           326,184.87
APR96           394,544.27
MAY96           449,862.25
JUN96           457,347.55
```

When you attempt to produce the same report with standard QDR syntax, then an error is signalled.

```
REPORT HEADING 'For Cogs in Sporting Division' DOWN month -
   W 11 HEADING 'MAX(actual,budget)'-
   MAX(actual,budget) (line cogs, division sporting)
```

The following error message is produced.

```
ERROR: A right parenthesis or an operator is expected after LINE.
```

4 Formulas, Aggregations, Allocations, and Models

This chapter provides information about creating and executing OLAP DML calculation specification objects. It includes the following topics:

- Formulas
- Aggregations
- Allocations
- Models

Formulas

You can save an expression in a formula. Frequently, you define a formula for ease of use and to save storage space. Once you have defined a formula for an expression, you can use the name of the formula takes the place of the text of the expression. Oracle OLAP does not store the data for a formula in a variable; instead it is calculated at runtime each time it is requested.

Before you create a formula, decide whether you want to specify the expression when you first define the formula object or whether you want to specify the expression for the formula after you define the formula object:

- When you decide to specify the expression when you first define the formula object, then:
  1. Issue a DEFINE FORMULA statement to define the formula object. Include the expression in the definition. Do not specify values for the `datatype` or `dimensions` arguments.
  2. (Optional) Issue a COMPILE statement to compile the formula.

- When you decide to specify the expression for the formula after you define the formula object, then:
  1. Issue a DEFINE FORMULA statement to define the formula object. Specify values for the `datatype` or `dimensions` arguments, but do not specify a value for the expression, itself.
  2. Issue a CONSIDER statement to make the formula the current definition.
  3. Issue an EQ statement to specify the expression for the formula.
  4. (Optional) Issue a COMPILE statement to compile the formula.

For example, you can define a formula to calculate dollar sales, as follows.

```
DEFINE dollar.sales FORMULA units * price
```

Aggregations

Historically, aggregating data was summing detail data to provide subtotals and totals. However, using OLAP DML aggmap objects you can specify more complex aggregation calculation:

- The summary data dimensioned by hierarchical dimension can be calculated using many different types of methods (for example, first, last, average, or weighted average). For an example of this type of aggregation, see Example 6–27, "Aggregating Up a Hierarchy" on page 6-58.
- The summary data dimensioned by a nonhierarchical dimension can be calculated using a model. This functionality is useful to calculate values for dimensions, such as line items, that do not have a hierarchical structure. Instead, you create a model to calculate the values of individual line items from one or more other line items or workspace objects. For an example of this type of aggregation, see Example 6–26, "Solving a Model in an Aggregation" on page 6-57.
- The detail data used to calculate the summary data can be in the variable that contains the summary data or in one or more other variables. The variable that contains the summary data does not have to have exactly the same dimensions as the variables that contain the detail data. For an examples of this type of aggregation, see Example 6–24, "Aggregating into a Different Variable"Example 6–24, "Aggregating into a Different Variable" on page 6-54, and Example 7–7, "Capstone Aggregation" on page 7-17.

- The data can be aggregated as a database maintenance procedure, in response to user requests for summarized data, or you can combine these approaches. See "Executing the Aggregation" on page 4-4 for more information.
- Data that is aggregated in response to user requests can be calculated each time it is requested or stored or cached in the analytic workspace for future queries.
- The specification for the aggregation can be permanent or temporary as described in "Creating Custom Aggregates" on page 4-6.

Aggregating Data

To aggregate data using the OLAP DML, take the following steps:

1. Decide if you want to aggregate all of the data as a database maintenance procedure using the AGGREGATE command or on-the-fly at runtime using the AGGREGATE function, or if you want to combine these approaches and precalculate some values and calculate others at run time. For a discussion of the various approaches, see "Executing the Aggregation" on page 4-4.

   Note: When the variable that contains the data you want to aggregate is dimensioned by a compressed composite, you must use the AGGREGATE command to aggregated the data. See "Aggregating Variables Dimensioned by Compressed Composites" on page 6-38 for more information.

2. When the aggregation involves aggregating data up a variable dimensioned by a composite, ensure that the composite has a BTREE index.
3. Issue a DEFINE AGGMAP statement to define the aggmap object as type AGGMAP.
4. Write the aggregation specification as described in AGGMAP.
5. When aggregating a partitioned variable, run PARTITIONCHECK to check that the aggregation specification created in the previous step is compatible with the variable's partitioning. If it is not, either rewrite the aggregation specification or repartition the variable using CHGDFN.
6. When some or all of the data is to be aggregated at runtime:
   a. Compile the aggmap object as described in "Compiling Aggregation Specifications" on page 9-38.

   b. Save the aggmap object using an UPDATE command followed by COMMIT.
   c. (Optional) Add a $NATRIGGER property to the variable to trigger the AGGREGATE function in response to a runtime request for data.

7. (Optional) Add one or more of the following properties to variables that will use the aggmap object:
   - $AGGMAP to specify that the aggmap is the default aggmap for the variable.
   - $AGGREGATE_FROM or $AGGREGATE_FROMVAR to specify the location of the detail data when the detail data is not in the target variable.
8. For data that is to be precalculated:
   a. (Optional) Set the POUTFILEUNIT option so that you can monitor the progress of the aggregation.
   b. Use the AGGREGATE command with the aggmap to precalculate the data and store it in the database.

For brief descriptions of all of the OLAP DML statements that relate to aggregation, see "Aggregation Statements" on page A-17.

Executing the Aggregation

When variables are dimensioned with detailed, multilevel hierarchies, the number of cells of aggregate data can be many times greater than the number of cells of detail data. Users often query some levels of data heavily and other levels very infrequently. They tend to focus on top-level aggregates and only occasionally drill to middle-level aggregates, although the middle-level aggregates comprise the largest proportion of aggregate data.

For this reason, the OLAP DML provides two ways to aggregate data:

- As a data maintenance procedure using the AGGREGATE command.
- At run-time when needed using the AGGREGATE function.

The DBA can choose whatever method seems appropriate: by level, individual member, member attribute, time range, data value, or other criteria. You can also combine these approaches and precalculate some values and calculate others at run time. In this case, frequently, you use the same aggmap with the AGGREGATE command and the AGGREGATE function. However, in some cases you might use different aggmaps.

One step that you can take to achieve overall good performance is to balance the amount of the data that you aggregate and store in an analytic workspace with the amount of data that you specify for calculation on the fly. A technique called "skip level" aggregation pre-aggregates every other level in a dimension hierarchy. Good performance is a matter of trade-offs. (For more information about skip-level aggregation, see "Skip-Level Aggregation" on page 6-96.)

Aggregating Data as a Data Maintenance Procedure.

Using the AGGREGATE command, the DBA acquires detail data, calculates the aggregate values, and stores them in the analytic workspace for all users to share. This type of aggregate data is sometimes call precomputed or stored aggregation.

Precomputed aggregation supports the fastest querying time, but increases the size of the analytic workspace and therefore the size of the Oracle Database. The amount of precomputed data can also be limited by the amount of time available for the data task (often called a batch window).

Note: You must aggregate data in variables dimensioned using compressed composites using the AGGREGATE command. See "Aggregating Variables Dimensioned by Compressed Composites" on page 6-38 for more information

For an example of aggregating data as a batch job, see Example 7–2, "Precalculating Data in a Batch Job" on page 7-13. When an AGGREGATE command executes, Oracle OLAP *always* stores the results of the calculation directly in the variable in the same way it stores the results of an assignment statement. Additionally, if you issue another AGGREGATE command Oracle OLAP always recalculates the aggregation.

Aggregating Data at Run-Time When Needed.

You can use the AGGREGATE function in response to a runtime request for data. For example, an AGGREGATE function can be the expression of a $NATRIGGER property or a formula:

- As an expression of $NATRIGGER property, the AGGREGATE function is executed when a runtime requests data for NA or empty data cells.
- As the expression of a formula, the AGGREGATE function is executed whenever the formula is executed.

In either case, the aggregates are computed in response to the query. The results can be stored in a temporary cache for use throughout the session. When the session has write access to the analytic worksheet, the results can also be stored permanently. This type of aggregate data is referred to as on-the-fly or run-time aggregates. Calculating aggregate data at runtime slows querying time since the data must be calculated instead of just retrieved, but it does not require permanent storage for aggregate values.

There are a number of aggregation features that you can specify when you use the AGGREGATE function to aggregate data on the fly. For example, you can specify:

- Whether or not Oracle OLAP stores the results of the calculation directly in the variable or caches the data in the session cache. For a discussion of how to specify storage or caching, see "How Oracle OLAP Determines Whether to Store or Cache Aggregated Data" on page 6-23.
- Whether or not previously cached or stored data is recalculated by specifying or omitting FORCECALC keyword on the AGGREGATE function.
- Whether or not any `NA` values that result from the aggregation that are stored in the variable will cause an $NATRIGGER property to execute on future requests for `NA` variable values. For a discussion of caching the `NA` values which precludes $NATRIGGER execution for `NA` values that result from the execution of an aggregation, see "How Oracle OLAP Determines Whether to Store or Cache Results of $NATRIGGER" on page 6-21.

Creating Custom Aggregates

The definitions for most aggregations persist from one session to another. However, you might need to create session-only aggregates at runtime for forecasting or what-if analysis, or just because you want to view the data in an unforeseen way. Adding session-only aggregates is sometimes called creating custom aggregates. You can create non-persistent aggregated data without permanently changing the specification for the aggregation in the following ways:

- Using a MAINTAIN ADD SESSION statement, define temporary dimension members and include an aggregation specification as part of the definition of these members. The aggregation specification can either be a model or an aggmap. For an example of using this method to create a temporary aggregation, see "Creating Calculated Dimension Members with Aggregated Values" on page 16-84.
- Create a model that specifies the aggregation. Use an AGGMAP ADD statement to add the model to an aggmap at run time. At the end of a session, Oracle

OLAP automatically removes any models that you have added to an aggmap in this manner. See AGGMAP ADD or REMOVE model for more information.

Allocations

Allocating data involves creating lower-level data from summary data. Allocating data using the OLAP DML involves creating an ALLOCMAP type aggmap object that specifies how the data should be allocated, and executing that object using the ALLOCATE command to actually distribute the data from a source object to the cells of a target. The target is a variable that is dimensioned by one or more hierarchical dimensions. The source data is specified by dimension values at a higher level in a hierarchical dimension than the values that specify the target cells.

ALLOCATE uses an aggmap to specify the dimensions and the values of the hierarchies to use in the allocation, the method of operation to use for a dimension, and other aspects of the allocation.

Some of the allocation operations are based on existing data. The object containing that data is the basis object for the allocation. In those operations, ALLOCATE distributes the data from the source based on the values of the basis object.

ALLOCATE has operations that are the inverse of the operations of the AGGREGATE command. The allocation operation methods range from simple allocations, such as copying the source data to the cells of the target variable, to very complex allocations, such as a proportional distribution of data from a source that is a formula, with the amount distributed being based on another formula, with multiple variables as targets, and with an aggmap that specifies different methods of allocation for different dimensions.

The Oracle OLAP allocation system is very flexible and has many features, including the following:

- The source, basis, and target objects can be the same variable or they can be different objects.
- The source and basis objects can be formulas, so you can perform computations on existing data and use the result as the source or basis of the allocation.
- You can specify the method of operation of the allocation for a dimension. The operations range from simple to very complex.
- You can specify whether the allocated value is added to or replaces the existing value of the target cell.

- You can specify an amount to add to or multiply by the allocated value before the result is assigned to the target cell.
- You can lock individual values in a dimension hierarchy so that the data of the target cells for those dimension values is not changed by the allocation. When you lock a dimension value, then the allocation system normalizes the source data, which subtracts the locked data from the source before the allocation. You can choose to not normalize the source data.
- You can specify minimum, maximum, floor, or ceiling values for certain operations.
- You can copy the allocated data to a second variable so that you can have a record of individual allocations to a cell that is the target of multiple allocations.
- You can specify ways of handling allocations when the basis has a null value.
- You can use the same aggmap in different ALLOCATE commands to use the same set of dimension hierarchy values, operations, and arguments with different source, basis, or target objects.

Allocating Data

To allocate data using an aggmap object, use the following OLAP DML statements in the order indicated:

1. Issue a DEFINE AGGMAP statement to define the aggmap object and an ALLOCMAP statement to indicate that the aggmap object is of type ALLOCMAP and that it contains an allocation specification. For example:

   ```
   DEFINE myaggmap AGGMAP
   ALLOCMAP 'END'
   ```

2. Add a specification to the aggmap object that specifies the allocation that you want performed.
3. Save the aggmap object using an UPDATE command followed by COMMIT.
4. (Optional) Set the POUTFILEUNIT option so that you can monitor the progress of the allocation.
5. (Optional) Redesign the allocation error log by setting the ALLOCERRLOGFORMAT and ALLOCERRLOGHEADER options to nondefault values.
6. (Optional) Set the $ALLOCMAP on one or more variables to specify that the aggmap is the default allocation specification for the variables.

7. (Recommended, but optional) Limit the variable to the target cells (that is, the cells into which you want to allocate data).
8. Issue an ALLOCATE statement to allocate the data.

For brief descriptions of all of the OLAP DML statements that relate to allocation, see "Allocation Statements" on page A-18.

Handling NA Values

Sometimes you want to overwrite existing data when allocating values to a target variable and at other times you want to write allocated values to target cells that have an NA basis before the allocation. For example, when you create a new product in your product dimension, then no basis exists for the new product in your budget variable. You want to allocate advertising costs for the entire product line, including the new product.

You can handle NA values using formulas and hierarchical operators in a RELATION (for allocation) statement in the following ways:

- Handling NA data with formulas—One way to handle the NA values is to construct a basis that only describes the desired target cells. This is the preferred method. You can refine your choice of basis values by deriving the basis from a formula. The following statements define a formula that equates the values of the new product to twice the value of an existing product. You could use such a formula as the basis for allocating advertising costs to the new product.

  ```
  DEFINE formula_basis FORMULA DECIMAL <product>
  EQ IF product EQ 'NEWPRODUCT' -
     THEN 2 * product.budget(product 'EXISTINGPRODUCT') -
     ELSE product.budget
  ```

- Handling NA data with hierarchical operators—To allocate data to target cells that currently have NA values, use a hierarchical operator in a RELATION (for allocation) statement in the allocation specification. The hierarchical operators use the hierarchy of the dimension rather than existing data as the allocation basis. A danger in using hierarchical operators is the possibility of densely populating your detail level data, which can result in a much larger analytic workspace and require much more time to aggregate the data.

  To continue the example of allocating the advertising cost for the new product, you could use the hierarchical last operator HLAST to specify allocating the cost to the new (and presumably the last) product in the product dimension hierarchy.

Models

A **model** is a set of interrelated equations that can assign results either to a variable or to a dimension value. For example, in a financial model, you can assign values to specific line items, such as `gross.margin` or `net.income`.

```
gross.margin = revenue - cogs
```

When an assignment statement assigns data to a dimension value or refers to a dimension value in its calculations, then it is called a dimension-based equation. A dimension-based equation does not refer to the dimension itself, but only to the values of the dimension. Therefore, when the model contains any dimension-based equations, then you must specify the name of each of these dimensions in a `DIMENSION` statement at the beginning of the model.

When a model contains any dimension-based equations, then you must supply the name of a **solution variable** when you run the model. The solution variable is both a source of data and the assignment target of model equations. It holds the input data used in dimension-based equations, and the calculated results are stored in designated values of the solution variable. For example, when you run a financial model based on the `line` dimension, you might specify `actual` as the solution variable.

Dimension-based equations provide flexibility in financial modeling. Since you do not need to specify the modeling variable until you solve a model, you can run the same model with the `actual` variable, the `budget` variable, or any other variable that is dimensioned by `line`.

Models can be quite complex. You can:

- Include one model within another model as discussed in "Nesting Models" on page 4-11
- Use data from different time periods as discussed in "Using Data from Past and Future Time Periods" on page 4-12
- Solve simultaneous equations as discussed in "Solving Simultaneous Equations" on page 4-13
- Create models for different scenarios as described in "Modeling for Multiple Scenarios" on page 4-14

Creating Models

To create an OLAP DML model, take the following steps:

1. Issue a DEFINE MODEL statement to define the program object.
2. Add a specification to the model to specify the processing that you want performed as described in MODEL.
3. Compile the model as described in "Compiling a Model" on page 4-14.
4. (Optional) If necessary, change the settings of model options listed in Table 17–1, "Model Options" on page 17-23.
5. Execute the model as described in "Running a Model" on page 4-15.
6. Debug the model as described in "Debugging a Model" on page 4-18.

For an example of creating a model, see "Creating a Model" on page 17-25.

Nesting Models

You can include one model within another model by using an INCLUDE statement. The model that contains the INCLUDE statement is referred to as the **parent model**. The included model is referred to as the **base model**. You can nest models by placing an INCLUDE statement in a base model. For example, model `myModel1` can include model `myModel2`, and model `myModel2` can include model `myModel3`. The nested models form a hierarchy. In this example, `myModel1` is at the top of the hierarchy, and `myModel3` is at the root.

When a model contains an INCLUDE statement, then it cannot contain any DIMENSION (in models) statements. A parent model inherits its dimensions, if any, from the DIMENSION statements in the root model of the included hierarchy. In the example just given, models `myModel1` and `myModel2` both inherit their dimensions from the DIMENSION statements in model `myModel3`.

The INCLUDE statement enables you to create modular models. When certain equations are common to several models, then you can place these equations in a separate model and include that model in other models as needed.

The INCLUDE statement also facilitates what-if analyses. An experimental model can draw equations from a base model and selectively replace them with new equations. To support what-if analysis, you can use equations in a model to mask previous equations. The previous equations can come from the same model or from included models. A masked equation is not executed or shown in the MODEL.COMPRPT report for a model

Dimension Status and Model Equations

When a model contains an assignment statement to assign data to a dimension value, then the dimension is limited temporarily to that value, performs the calculation, and restores the initial status of the dimension.

For example, a model might have the following statements.

```
DIMENSION line
gross.margin = revenue - cogs
```

If you specify `actual` as the solution variable when you run the model, then the following code is constructed and executed.

```
PUSH line
LIMIT line TO gross.margin
actual = actual(line revenue) - actual(line cogs)
POP line
```

This behind-the-scenes construction lets you perform complex calculations with simple model equations. For example, line item data might be stored in the `actual` variable, which is dimensioned by `line`. However, detail line item data might be stored in a variable named `detail.data`, with a dimension named `detail.line`.

When your analytic workspace contains a relation between `line` and `detail.line`, which specifies the line item to which each detail item pertains, then you might write model equations such as the following ones.

```
revenue = total(detail.data line)
expenses = total(detail.data line)
```

The relation between `detail.line` and `line` is used automatically to aggregate the detail data into the appropriate line items. The code that is constructed when the model is run ensures that the appropriate total is assigned to each value of the `line` dimension. For example, while the equation for the `revenue` item is calculated, `line` is temporarily limited to `revenue`, and the `TOTAL` function returns the total of detail items for the `revenue` value of `line`.

Using Data from Past and Future Time Periods

Several OLAP DML functions make it easy for you to use data from past or future time periods. For example, the `LAG` function returns data from a specified previous time period, and the `LEAD` function returns data from a specified future period.

When you run a model that uses past or future data in its calculations, you must make sure that your solution variable contains the necessary past or future data. For example, a model might contain an assignment statement that bases an estimate of the `revenue` line item for the current month on the `revenue` line item for the previous month.

```
DIMENSION line month
...
revenue = LAG(revenue, 1, month) * 1.05
```

When the `month` dimension is limited to `Apr2004` to `Jun2004` when you run the model, then you must be sure that the solution variable contains `revenue` data for `Mar96`.

When your model contains a `LEAD` function, then your solution variable must contain the necessary future data. For example, when you want to calculate data for the months of April through June of 2004, and when the model retrieves data from one month in the future, then the solution variable must contain data for July 2004 when you run the model.

Handling NA Values

Oracle OLAP observes the NASKIP2 option when it evaluates equations in a model. NASKIP2 controls how `NA` values are handled when `+` (plus) and `-` (minus) operations are performed. The setting of NASKIP2 is important when the solution variable contains `NA` values.

The results of a calculation may be `NA` not only when the solution variable contains an `NA` value that is used as input, but also when the target of a simultaneous equation is `NA`. Values in the solution variable are used as the initial values of the targets in the first iteration over a simultaneous block. Therefore, when the solution variable contains `NA` as the initial value of a target, an `NA` result may be produced in the first iteration, and the `NA` result may be perpetuated through subsequent iterations.

To avoid obtaining `NA` for the results, you can make sure that the solution variable does not contain `NA` values or you can set NASKIP2 to `YES` before running the model.

Solving Simultaneous Equations

An iterative method is used to solve the equations in a simultaneous block. In each iteration, a value is calculated for each equation, and compares the new value to the value from the previous iteration. When the comparison falls within a specified tolerance, then the equation is considered to have converged to a solution. When the comparison exceeds a specified limit, then the equation is considered to have diverged.

When all the equations in the block converge, then the block is considered solved. When any equation diverges or fails to converge within a specified number of iterations, then the solution of the block (and the model) fails and an error occurs.

You can exercise control over the solution of simultaneous equations, use the OLAP DML options described in Table 17–1, "Model Options" on page 17-23. For example, using these options, you can specify the solution method to use, the factors to use in testing for convergence and divergence, the maximum number of iterations to perform, and the action to take when the assignment statement diverges or fails to converge.

Modeling for Multiple Scenarios

Instead of calculating a single set of figures for a month and division, you might want to calculate several sets of figures, each based on different assumptions.

You can define a **scenario** model that calculates and stores forecast or budget figures based on different sets of input figures. For example, you might want to calculate profit based on optimistic, pessimistic, and best-guess figures.

To build a scenario model, follow these steps.

1. Define a scenario dimension.
2. Define a solution variable dimensioned by the scenario dimension.
3. Enter input data into the solution variable.
4. Write a model to calculate results based on the input data.

For an example of building a scenario model see, Example 17–12, "Building a Scenario Model" on page 17-26.

Compiling a Model

When you finish writing the statements in a model, you can use COMPILE to compile the model. During compilation, COMPILE checks for format errors, so you can use COMPILE to help debug your code before running a model. When you do not use COMPILE before you run the model, then the model is compiled automatically before it is solved.

When you compile a model, either by using a COMPILE statement or by running the model, the model compiler examines each equation to determine whether the assignment target and each data source is a variable or a dimension value.

Understanding Dependencies

After resolving each name reference, the model compiler analyzes dependencies between the equations in the model. A dependence exists when the expression on the right-hand side of the equal sign in one equation refers to the assignment target of another equation. When an assignment statement indirectly depends on itself as the result of the dependencies among equations, then a cyclic dependence exists between the equations.

The model compiler structures the equations into blocks and orders the equations within each block, and the blocks themselves, to reflect dependencies. The compiler can produce three types of solution blocks: simple blocks, step blocks, and simultaneous blocks as described in "Dependencies Between Equations" on page 9-34.

Checking for Additional Problems

The compiler does not analyze the contents of any programs or formulas that are used in model equations. Therefore, you must check the programs and formulas yourself to make sure they do *not* do any of the following:

- Refer to the value of any variable used in the model.
- Refer to the solution variable.
- Limit any of the dimensions used in the model.
- Invoke other models.

When a model or program violates any of these restrictions, the results of the model may be incorrect.

Running a Model

To run or solve a model, use the following syntax.

model-name [*solution-variable*] [NOWARN]

where:

- *model-name* is the name of the model.

- *solution-variable* is the name of a numeric variable that serves as both the source and the target of data in a model that contains dimension-based equations. The solution variable is usually dimensioned by all the dimensions on which model equations are based (as specified in explicit or included DIMENSION commands). The *solution-variable* argument is required when the model contains any dimension-based equations. When all the model equations are based only on variables, a solution variable is not needed and an error occurs when you supply this argument.
- NOWARN is an optional argument that specifies that you do not want to be warned when the model contains a block of simultaneous equations.

When you run a model, you should keep these points in mind:

- Before you run a model, the input data must be available in the solution variable.
- Before running a model that contains a block of simultaneous equations, you might want to check or modify the values of some OLAP DML options that control the solution of simultaneous blocks. These options are described briefly in Table 17–1, "Model Options" on page 17-23.
- When your model contains any dimension-based equations, then you must provide a numeric solution variable that serves both as a source of data and as the assignment target for equation results. The solution variable is usually dimensioned by all of the dimensions on which model equations are based and also by the other dimensions of the solution variable on which you are not basing equations.
- When you run a model, a loop is performed automatically over the values in the current status list of each of the dimensions of the solution variable on which you have not based equations.
- When a model equation bases its calculations on data from previous time periods, then the solution variable must contain data for these previous periods. When it does not, or when the first value of the dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR is in status, then the results of the calculation are `NA`.

Dimensions of Solution Variables

In a model with dimension-based equations, the solution variable is usually dimensioned by the dimensions on which model equations are based. Or, when a solution variable is dimensioned by a composite, the model equations can be based on base dimensions of the composite. The dimensions on which model equations are based are listed in explicit or inherited DIMENSION (in models) commands.

The following special cases regarding the dimensions of the solution variable can occur:

- The solution variable can have dimensions that are not listed in DIMENSION commands. Oracle OLAP automatically loops over the values in the status of the extra dimensions. For example, the model might contain a DIMENSION command that lists the `line` and `month` dimensions, but you might specify a solution variable dimensioned by `line`, `month`, and `division`. Oracle OLAP automatically loops over the `division` dimension when you run the model. The solution variable can also be dimensioned by a composite that has one or more base dimensions that are not listed in DIMENSION commands. See "Solution Variables Dimensioned by a Composite" on page 4-17.
- When the solution variable has dimensions that are not listed in DIMENSION commands *and* when any of these other dimensions are the dimension of a step or simultaneous block, an error occurs.
- Oracle OLAP loops over the values in the status of all the dimensions listed in DIMENSION commands, regardless of whether the solution variable is dimensioned by them. Therefore, Oracle OLAP will be doing extra, unnecessary work when the solution variable is not dimensioned by all the listed dimensions. Oracle OLAP warns you of this situation before it starts solving the model.
- The inclusion of an unneeded dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR in a DIMENSION command causes incorrect results when you use a loan, depreciation, or aggregation function in a model equation. This happens because any component of a model equation that refers to the values of a model dimension behaves *as if* that component has all the dimensions of the model.

Solution Variables Dimensioned by a Composite

When a solution variable contains a composite in its dimension list, Oracle OLAP observes the sparsity of the composite whenever possible. As it solves the model, Oracle OLAP confines its loop over the composite to the values that exist in the composite. It observes the current status of the composite's base dimensions as it loops.

However, for proper solution of the model, Oracle OLAP must treat the following base dimensions of the composite as regular dimensions:

- A base dimension that is listed in a DIMENSION (in models) command.
- A base dimension that is implicated in a model equation created using SET (for example, an equation that assigns data to a variable dimensioned by the base dimension).
- A base dimension that is also a base dimension of a different composite that is specified in the ACROSS phrase of an equation. (See SET for more information on assignment statements and the use of ACROSS phrase.)

When a base dimension of a solution variable's composite falls in any of the preceding three categories, Oracle OLAP treats that dimension as a regular dimension and loops over all the values that are in the current status.

When the solution variable's composite has other base dimensions that do not fall in the special three categories, Oracle OLAP creates a temporary composite of these extra base dimensions. The values of the temporary composite are the combinations that existed in the original composite. Oracle OLAP loops over the temporary composite as it solves the model.

Debugging a Model

The following tools are available for debugging models:

- To see the order in which the equations in a model are solved, you can set the MODTRACE option to `YES` before you run the model.When you set MODTRACE to `YES`, you can use the DBGOUTFILE command to send debugging information to a file. The file produced by DBGOUTFILE interweaves each line of your model with its corresponding output.
- You can use the MODEL.COMPRPT, MODEL.DEPRT, and MODEL.XEQRPT programs and the INFO function to obtain information about the structure of a compiled model and the solution status of a model you have run.

5 OLAP DML Programs

This chapter provides an overview of create OLAP DML programs. It includes the following topics:

- Creating OLAP DML Programs
- Compiling Programs
- Testing and Debugging Programs
- Executing Programs

Creating OLAP DML Programs

An OLAP DML program is written in the OLAP DML. It acts on data in the analytic workspace and helps you accomplish some workspace management or analysis task. You can write OLAP DML programs to perform tasks that you must do repeatedly in the analytic workspace, or you can write them as part of an application that you are developing.

To create an OLAP DML program, take the following steps:

1. Issue a DEFINE PROGRAM statement to define the program object. When the program that you are defining will be used is a function, include the *datatype* and *dimension* arguments.
2. Add contents to the program that specify the processing that you want performed as described in "Specifying Program Contents" on page 5-2.
3. Compile the program as described in "Compiling Programs" on page 5-13.
4. Test and debug the program as described in "Testing and Debugging Programs" on page 5-14.
5. Execute the program as described in "Executing Programs" on page 5-17.

Specifying Program Contents

The content of a program consists of the following OLAP DML statements:

1. A PROGRAM statement that indicates the beginning of the program contents. (Omit when coding the specification in an Edit window of the OLAP Worksheet.)
2. (Optional) VARIABLE statements that define any local variables.
3. (Optional) ARGUMENT statements that declare arguments. (See "Passing Arguments" on page 5-3 for more information.)
4. Additional OLAP DML statements that specify the processing you want performed. You can use almost any of the OLAP DML statements in a program. There are also some OLAP DML statements, such as flow-of-control statements, that are only used in programs. For brief descriptions, see "Programming Statements" on page A-22.

   Use the following formatting guidelines as you add lines to your program:

   - Each line of code can have a maximum of 4000 bytes.
   - To continue a single command on the next line, place a hyphen (`-`) at the end of the line to be broken. The hyphen is called a continuation character.
   - You cannot use a continuation character in the middle of a text literal.
   - To write more than one command on a single line, separate the commands with semicolon (`;`).
   - Enclose literal text in single quotation marks (`'`). To include a single quotation mark within literal text, precede it with a backslash (\). To specify escape sequences, see "Escape Sequences" on page 2-4.
   - Precede comments with double quotation marks (`"`). You can place a comment, preceded by double quotation marks, either at the beginning of a line or at the end of a line, after some commands.
5. A final END statement that indicates the end of the contents of the program. (Omit when coding the specification in an Edit window of the OLAP Worksheet.)

Creating User-Defined Functions

One type of program that is commonly written is a user-define function that you can use in OLAP DML statements in much the same way as you use an OLAP DML function. A user-defined function is simply an OLAP DML program that returns a value. For an example of a user-defined function, see Example 8–7, "Passing an Argument to a User-Defined Function" on page 8-22.

When you create a user-defined function, you use a DEFINE PROGRAM statement that includes the *datatype* and *dimension* arguments. Within the program, you include a RETURN statement that returns a value. The return expression in the program should match the data type that is specified in its definition. When the data type of the return value does not match the data type that is specified in its definition, then the value is converted to the data type in the definition.

User-defined functions can accept arguments. A user-defined function returns only a single value. However, when you supply an argument to a user-defined function in a context that loops over a dimension (for example, in a REPORT command), then the function returns results with the same dimensions as its argument.

You must declare the arguments using the ARGUMENT command within the program, and you must specify the arguments in parentheses following the name of the program.

> **See Also:** "Passing Arguments" on page 5-3 for more information about using arguments with programs.

Passing Arguments

Use ARGUMENT statements to declare both simple and complex arguments (such as expressions). The ARGUMENT command also makes it convenient to pass arguments from one program to another, or to create your own user-defined functions. The ARGUMENT command lets you declare an argument of any data type, dimension, or valueset. Any ARGUMENT commands must precede the first executable line in the program. When you run the program, these declared arguments are initialized with the values you provided as arguments to the program. The program can then use these arguments in the same way it would use local variables.

Using Multiple Arguments A program can declare as many arguments as needed. When the program is executed with arguments specified, the arguments are matched positionally with the declared arguments in the program. When you run the program, you must separate arguments with spaces rather than with commas or other punctuation. Punctuation is treated as part of the arguments. For an example of passing multiple arguments, see Example 8–8, "Passing Multiple Arguments" on page 8-23.

Handling Arguments Without Converting Values to a Specific Data Type Sometimes you want your OLAP DML program to be able to handle arguments without converting values to a specific data type. In this case, you can specify a data type of `WORKSHEET` in the ARGUMENT and VARIABLE statements that define the arguments and temporary variables for the program. You can use WKSDATA to determine the actual data type of the argument or variable.

Passing Arguments as Text with Ampersand Substitution It is very common to pass a simple text argument to a program. However, there are some situations in which you might want to write more general programs or pass a more complicated text argument, such as an argument that is all of the data in one of the analytic workspace objects or the results of an expression. In these cases, you can pass the argument using a substitution expression. Passing an argument in this way is called **ampersand substitution**.

For the following types of arguments, you must *always* use an ampersand to make the appropriate substitution:

- Names of workspace objects, such as `units` or `product`
- Command keywords, such as `COMMA` or `NOCOMMA` in the REPORT command, or `A` or `D` in the SORT command

When you use ampersand substitution to pass the names of workspace objects to a program (rather than their values), the program has access to the objects themselves because the names are known to the program. This is useful when the program must manipulate the objects in several operations.

Note: You cannot compile and save any program line that contains an ampersand. Instead, the line is evaluated at run time, which can reduce the speed of your programs. Therefore, to maximize performance, avoid using ampersand substitution when another technique is available.

For an example of using ampersand substitution to pass multiple dimension values, see Example 16–6, "Using Ampersand Substitution with LIMIT" on page 16-17. For an example of using ampersand substitution to pass the text of an expression, see Example 8–10, "Passing the Text of an Expression" on page 8-25. For an example of using ampersand substitution to pass object names and keywords, see Example 8–11, "Passing Workspace Object Names and Keywords" on page 8-25.

See Also: "Substitution Expressions" on page 3-25 for more information about ampersand substitution.

Program Flow-of-Control

Like most programming languages, the OLAP DML has a number of commands that you can use to determine the flow-of-control within a program. However, you need to code explicit loops less frequently in an OLAP DML program because of the intrinsic looping nature of many OLAP DML statements.

Table 5–1, " Statements For Determining Flow-of-Control" on page 5-7 lists OLAP DML flow-of-control commands. The looping characteristic of OLAP DML commands is discussed in "Looping Nature of OLAP DML Commands and Functions" on page 5-5.

Looping Nature of OLAP DML Commands and Functions Unlike SQL statements that operate against a single row in a table, OLAP DML commands and functions usually operate against the entire array of data represented by an analytic workspace data object:

- When you issue a statement against an object that has one or more dimensions, the statement loops over the values in status for each dimension of the object and performs the requested operation.
- When you use an OLAP assignment statement (that is, SET) to assign values to a variable, Oracle OLAP loops through all of the cells assigning values in sequence.

  Assume for example, that there is a dimension named `prodid` that has three values, `Prod01`, `Prod02`, and `Prod03`, and you have a variable named `quantity` that is dimensioned by `prodid`. As the following code snippet illustrates, when you assign the value 3 to quantity, Oracle OLAP loops over the values in status for each dimension of the target and assigns the value 3 to all the cells in `quantity`.

```
quantity = 3
REPORT quantity

PRODID          QUANTITY
-------------- ----------
PROD01              3.00
PROD02              3.00
PROD03              3.00
```

- Other OLAP DML statements (for example, REPORT, ROW, and FOR) also loop through all of the values of a dimensioned object when they execute.

By default, looping statements loop through the values of a dimensioned object using the order in which the dimensions of the object are listed in the definition of the object. Also, when a variable is dimensioned by a composite, most looping statements loop through the variable as though it was *not* dimensioned by a composite, but was, instead, dimensioned by the base dimensions of the composite.

The OLAP DML provides ways for you to change the default looping behavior or to explicitly request looping:

- ACROSS phrase—Some looping statements (such as SET that you use to assign values) have an ACROSS phrase that you can use to specify nondefault looping behavior. Using the ACROSS phrase, you can specify:
  - The specific dimensions (and order) in which you want the statement to loop. In this case, the statement will loop over the dimensions in the order that you specify them in the ACROSS phrase, not in the order in which they appear in the variable's definition.
  - A composite over which you want the statement to loop. When a variable is dimensioned by a composite, specifying the name of a composite improves performance. When you specify the name of a composite in the ACROSS phrase of a looping statement, the statement only loops over the existing cells of a variable.

  For more complete documentation of the ACROSS phrase, see SET.
- ACROSS command—When an OLAP DML statement is not a looping statement or does not include an ACROSS phrase, you can request looping behavior by coding the DML statement as an argument of the ACROSS command.

Flow-of Control Commands The OLAP DML contains the flow-of-control statements typically found in a programming language. Table 5–1, " Statements For Determining Flow-of-Control" on page 5-7 lists these statements.

Table 5–1 Statements For Determining Flow-of-Control

Statement	Description
BREAK statement	Transfers program control from within a SWITCH, FOR, or WHILE statement to the statement immediately following the DOEND associated with SWITCH, FOR, or WHILE.
CONTINUE statement	Transfers program control to the end of a FOR or WHILE loop (just before the DO/DOEND statement), allowing the loop to repeat. You can use CONTINUE only within programs and only with FOR or WHILE.
DO ... DOEND statements	Brackets a group of one or more statements. DO and DOEND are normally used to bracket a group of statements that are to be executed under a condition specified by an IF statement, a group of statements in a repeating loop introduced by FOR or WHILE, or the CASE labels for a SWITCH statement.
FOR statement	Specifies one or more dimensions whose status will control the repetition of one or more statements.
GOTO statement	Alters the sequence of statement execution within the program by indicating the next program statement to execute.
IF...THEN...ELSE statement	Executes one or more statements in a program if a specified condition is met. Optionally, it also executes an alternative statement or group of statements when the condition is not met.
OKFORLIMIT	An option that determines whether you can limit the dimension you are looping over within an explicit FOR loop.
RETURN statement	Terminates execution of a program prior to its last line. You can optionally specify a value that the program will return.
SIGNAL statement	Produces an error message and halts normal execution of the program. When the program contains an active trap label, execution branches to the label. Without a trap label, execution of the program terminates and, if the program was called by another program, execution control returns to the calling program.
SWITCH statement	Provides a multipath branch in a program. The specific path taken during program execution depends on the value of the control expression that is specified with SWITCH.

Table 5–1 (Cont.) Statements For Determining Flow-of-Control

Statement	Description
TEMPSTAT statement	Limits the dimension you are looping over, inside a FOR loop or inside a loop that is generated by the REPORT command. Status is restored after the statement following TEMPSTAT. If a DO ... DOEND phrase follows TEMPSTAT, status is restored when the matched DOEND or a BREAK or GOTO statement is encountered.
TRAP statement	Causes program execution to branch to a label when an error occurs in a program or when the user interrupts the program. When execution branches to the trap label, that label is deactivated.
WHILE statement	Repeatedly executes a statement while the value of a Boolean expression remains TRUE.

Preserving the Environment Settings

There are two types of environments:

- Session environment. The dimension status, option values, and output destination that are in effect before a program is run make up the session environment.
- Program environment. The dimension status, option values, and output destination that you use in a program make up the program environment.

Changing the Program Environment

To perform a task within a program, you often need to change the output destination or some dimension and option values. For example, you might run a monthly sales report that always shows the last six months of sales data. You might want to show the data without decimal places, include the text "No Sales" where the sales figure is zero, and send the report to a file. To set up this program environment, you can use the following commands in your program.

```
LIMIT month TO LAST 6
DECIMALS = 0
ZSPELL = 'No Sales'
OUTFILE monsales.txt
```

To avoid disrupting the session environment, the initialization section of a program should save the values of the dimensions and options that will be set in the program. At the end of the program, you can restore the saved environment, so that other programs do not need to be concerned about whether any values have been changed. In addition, when you have sent output to a file, then the exit sections should return the output destination to the default outfile.

Ways to Save and Restore Environments

The following suggestions let you save the environment of a program or a session:

- When you want to save the current status or value of a dimension, a valueset, an option, or a single-cell variable that will be changed in the current program, then use the PUSHLEVEL and PUSH commands. You can restore the current status values using the POPLEVEL and POP commands.
- When you want to save, access, or update the current status or value of a dimension, a valueset, an option, a single-cell variable, or a single-cell relation for use in the current session, then use a named context. Use the CONTEXT command to define the context.

Contexts are the most sophisticated way to save object values for use during a session. With contexts, you can access, update, and commit the saved object values. In contrast, PUSH and POP simply allow you to save and restore values. Typically, you use the PUSH and POP commands within a program to make changes that apply only during the execution of the program.

Saving the Status of a Dimension or the Value of an Option

The PUSH command saves the current status of a dimension, the value of an option, or the value of a single-cell variable. For example, to save the current value of the `DECIMALS` option so you can set it to a different value for the duration of the program, use the following command in the initialization section.

```
PUSH DECIMALS
```

You do not need to know the original value of the option to save it or to restore it later. You can restore the saved value with the POP command.

```
POP DECIMALS
```

You must make sure the POP command is executed when errors cause abnormal termination of the program, as well as when the program ends normally. Therefore, you should place the POP command in the normal and abnormal exit sections of the program.

Saving Several Values at Once

You can save the status of one or more dimensions and the values of any number of options and variables in a single PUSH command, and you can restore the values with a single POP command, as shown in the following example.

```
PUSH month DECIMALS ZSPELL
       ...
POP month DECIMALS ZSPELL
```

Using Level Markers

When you are saving the values of several dimensions and options, then the PUSHLEVEL and POPLEVEL commands provide a convenient way to save and restore the session environment.

You first use the PUSHLEVEL command to establish a level marker. Once the level marker is established, you use the PUSH command to save the status of dimensions and the values of options or single-cell variables.

When you place more than one PUSH command between the PUSHLEVEL and POPLEVEL commands, then all the objects that are specified in those PUSH commands are restored with a single POPLEVEL command.

By using PUSHLEVEL and POPLEVEL, you save some typing as you write your program because you only need to type the list of objects once. You also reduce the risk of omitting an object from the list or misspelling the name of an object.

For an example of creating level markers, see Example 19–45, "Creating Level Markers" on page 19-103.Example 19–46, "Nesting PUSHLEVEL and POPLEVEL Commands" on page 19-103 illustrates nesting PUSHLEVEL and POPLEVEL commands.

Using CONTEXT to Save Several Values at Once

As an alternative to using PUSHLEVEL and POPLEVEL, you can use the CONTEXT command. After you create a context, you can save the current status of dimensions and the values of options, single-cell variables, valuesets, and single-cell relations in the context. You can then restore some or all of the object values from the context. The CONTEXT function returns information about objects in a context.

Handling Errors

When an error occurs anywhere in a program, Oracle OLAP performs the following actions:

1. Stores the name of the error in the ERRORNAME option, and the text of the error message in the ERRORTEXT option.

   > **Note:** When the ERRNAMES option is set to the default value of `YES`, the ERRORTEXT option contains the name of the error (that is, the value of the ERRORNAME option) as well as the text of the error message.

2. When ECHOPROMPT is `YES`, then Oracle OLAP echoes input lines, error messages, and output lines, to the current outfile. When you use the OUTFILE or DBGOUTFILE command, you can capture the error messages in a file. See Example 19–17, "Directing Output to a File" on page 19-38 for an example of directing output to a file.
3. When error trapping is off, then the execution of the program is halted. When error trapping is on, then the error is trapped.

Trapping an Error

To make sure the program works correctly, you should anticipate errors and set up a system for handling them. You can use the TRAP command to turn on an error-trapping mechanism in a program. When error trapping is on and an error is signaled, then the execution of the program is not halted. Instead, error trapping does the following:

1. Turns off the error-trapping mechanism to prevent endless looping in case additional errors occur during the error-handling process
2. Branches to the label that is specified in the `TRAP` command
3. Executes the commands following the label

Suppressing Error Messages

When you do not want to produce the error message that is normally provided for a given error, then you can use the `NOPRINT` keyword with the `TRAP` command.

```
TRAP ON error NOPRINT
```

When you use the `NOPRINT` keyword with `TRAP`, control branches to the `error` label, and an error message is not issued when an error occurs. The commands following the `error` label are then executed.

When you suppress the error message, you might want to produce your own message in the abnormal exit section. The SHOW command produces the text you specify but does not signal an error.

```
TRAP ON error NOPRINT
        ...
error:
        ...
SHOW 'The report will not be produced.'
```

The program continues with the next command after producing the message.

Creating Your Own Error Messages

All errors that occur when a command or command sequence does not conform to its requirements are signaled automatically. In your program, you can establish additional requirements for your own application. When a requirement is not met, you can execute the SIGNAL command to signal an error.

You can give the error any name. When the SIGNAL command is executed, the error name you specify is stored in the ERRORNAME option, just as an OLAP DML error name is automatically stored. When you specify your own error message in the SIGNAL command, then your message is produced just as an OLAP DML error message is produced. When you are using a TRAP command to trap errors, a SIGNAL command branches to the TRAP label after the error message is produced.

For an example of signaling an error, see Example 21–32, "Signaling an Error" on page 21-77.

When you want to produce a warning message without branching to an error label, then you can use the SHOW command as illustrated in Example 21–31, "Creating Error Messages Using SHOW" on page 21-74.

Handling Errors in Nested Programs

When handling errors in nested programs, the error-handling section in each program should restore the environment. It can also handle any special error conditions that are particular to that program. For example, when your program signals its own error, then you can include commands that test for that error.

Any other errors that occur in a nested program should be passed up through the chain of programs and handled in each program. To pass errors through a chain of nested programs, you can use one of two methods, depending on when you want the error message to be produced:

- The error message is produced immediately, and the error condition is then passed through the chain of programs. This approach is illustrated in Example 24–2, "Producing a Program Error Message Immediately" on page 24-6.
- The error is passed through the chain of programs first, and the error message is produced at the end of the chain. This approach is illustrated in Example 24–3, "Producing a Program Error Message at the End of the Chain" on page 24-7.

The `SIGNAL` command is used in both methods.

Handling Errors While Saving the Session Environment

To correctly handle errors that might occur while you are saving the session environment, place your `PUSHLEVEL` command before the `TRAP` command and your `PUSH` commands after the `TRAP` command.

```
PUSHLEVEL 'firstlevel'
TRAP ON error
PUSH
 ...
```

In the abnormal exit section of your program, place the error label (followed by a colon) and the commands that restore the session environment and handle errors. The abnormal exit section might look like this.

```
error:
POPLEVEL 'firstlevel'
OUTFILE EOF
```

These commands restore saved dimension status and option values and reroute output to the default outfile.

Compiling Programs

You can explicitly compile a program by using the `COMPILE` command. If you do not explicitly compile a program, then it is compiled when you run the program for the first time.

When a program is compiled, it translates the program commands into efficient processed code that executes much more rapidly than the original text of the program. When errors are encountered in the program, then the compilation is not completed, and the program is considered to be uncompiled.

After you compile a program, the compiled code is used each time you run the program in the current session. When you update and commit your analytic workspace after compiling a program, the compiled code is saved in your analytic workspace and used to run the program in future sessions. Therefore, you should be sure to update and commit after compiling a program. This is particularly critical when the program is part of an application that is run by many users. Unless the compiled version of the program is saved in the analytic workspace, the program is recompiled individually in each user session.

Example 9–11, "Compiling a Program" on page 9-39 illustrates using COMPILE to compile a program

Finding Out If a Program Has Been Compiled

You can use the `ISCOMPILED` choice of the `OBJ` function to determine whether a specific program in your analytic workspace has been compiled since the last time it was modified. The function returns a Boolean value.

```
SHOW OBJ(ISCOMPILED 'myprogram')
```

Programming Methods That Prevent Compilation

Program lines that include ampersand substitution are *not* compiled. Any syntax errors are not caught until the program is run. A program whose other lines compiled correctly is considered to be a compiled program.

When your program defines an object and then uses the object in the program, the program cannot be compiled. `COMPILE` treats the reference to the object as a misspelling because the object does not yet exist in the analytic workspace.

Testing and Debugging Programs

Even when your program compiles cleanly, you must also test the program by running it. Running a program helps you detect errors in commands with ampersand substitution, errors in logic, and errors in any nested programs.

To test a program by running it, use a full set of test data that is typical of the data that the program processes. To confirm that you test all the features of the program, including error-handling mechanisms, run the program several times, using different data and responses. Use test data that:

- Falls within the expected range
- Falls outside the expected range

- Causes each section of a program to execute

Error and Debugging Options

A number of options determine how errors are handled and what happens during debugging. These options are listed in Table 5–2, " Error Handling Options" on page 5-15 and Table 5–3, "Debugging Options" on page 5-15.

Table 5–2 Error Handling Options

Statement	Description
ERRNAMES	An option that controls whether the value of the ERRORTEXT option contains the name of the error (that is, the value of the ERRORNAME option) as well as the text of the error message.
ERRORNAME	An option that contains the name of the first error that occurs when you execute a program or when you execute an OLAP DML statement.
ERRORTEXT	An option that contains the text of the first error message that occurs when you execute a program or a statement.
PERMITERROR	An option that determines whether or not an error is signaled on attempted access of a variable for which read or write permission is denied by a PERMIT command.
MODERROR	Specifies the action to be taken when a model equation diverges or a block fails to converge. The possible values are STOP, CONTINUE, and DEBUG.
BADLINE	When a program, model, or input file is executing, an option that controls whether Oracle OLAP records, in the current outfile, the line that caused an error.
INF_STOP_ON_ERROR	An option that specifies the behavior of Oracle OLAP when an error is reached when reading from a file using the INFILE command

Table 5–3 Debugging Options

Statement	Description
EXPTRACE	An option that controls whether system DML programs are traced when the PRGTRACE option is set to YES.
PRGTRACE	An option that determines whether each line of a program is recorded in the current outfile or in a debugging file during execution of the program.

Table 5–3 Debugging Options

Statement	Description
MODTRACE	An option that controls whether each equation in a model is recorded in a file during execution of the model.
MODERROR	Specifies the action to be taken when a model equation diverges or a block fails to converge. The possible values are STOP, CONTINUE, and DEBUG.

Generating Diagnostic Messages

Each time you run the program, confirm that the program executes its commands in the correct sequence and that the output is correct. As an aid in analyzing the execution of your program, you can include `SHOW` commands in the program to produce diagnostic or status messages. Then delete the `SHOW` commands after your tests are complete.

When you detect or suspect an error in your program or a nested program, you can track down the error by using the debugging techniques that are described in the rest of this section.

Identifying Bad Lines of Code

When you set the `BADLINE` option to `YES`, additional information is produced, along with any error message when a bad line of code is encountered. When the error occurs, the error message, the name of the program, and the program line that triggered the error are sent to the current outfile. You can edit the specified program to correct the error and then run the original program. Example 8–22, "Using the BADLINE Option" on page 8-66 illustrates using the BADLINE option.

Sending Output to a Debugging File

When your program contains an error in logic, then the program might execute without producing an error message, but it executes the wrong set of commands or produces incorrect results. For example, suppose you write a Boolean expression incorrectly in an `IF` command (for example, you use `NE` instead of `EQ`). The program executes the commands you specified, but it does so under the wrong conditions.

To find an error in program logic, you often need to see the order in which the commands are being executed. One way you can do this is to create a debugging file and then examine the file to diagnose any problems in your programs.

To create a debugging file, you use the DBGOUTFILE command.

The DBGOUTFILE command merely creates a file for debugging. To specify that you want each program line to be sent, as it executes, to the debugging file, set the PRGTRACE option to YES.

When you want the debugging file to interweave the program lines with both the program input and error messages, then set the ECHOPROMPT option to YES.

> **See Also:** The following examples of using a debugging file:
>
> - Example 9–38, "Debugging with a Debugging File" on page 9-96
> - Example 9–39, "Sending Debugging Information to a File" on page 9-97

Executing Programs

You can invoke a program that does not return a value by using the CALL command. You enclose arguments in parentheses, and they are passed by value.

For example, suppose you create a simple program named addit to add two integers. You can use the CALL command in the main program of your application to invoke the program.

```
CALL addit (3, 4)
```

You can also invoke programs in much the same way as you issue OLAP DML statements. In this case, for a program that does not return a value, you merely use the program name as you would an OLAP DML command. If the OLAP DML program returns a value then it is a user-defined function. You invoke user-defined functions in the same way as you use built-in functions. You merely use the program name in an expression and enclose the program arguments, if any, in parentheses.

For example:

- You can use the program name as an expression in a command.

  The following REPORT command uses the value that is returned by the user-defined function isrecent that has a single argument, actual.

  ```
  REPORT isrecent(actual)
  ```

- You can use SET to assign the return value of the function to a variable.

The following command assigns the return value of the user-defined function named `tempsales` to a temporary variable called `mytempsales`.

```
mytempsales = tempsales
```

Note: Although you can also run user-defined functions using the CALL command, you do not have access to the return value.

You can also create programs that execute automatically when a user attaches an analytic workspace as described in "Startup Programs" on page 1-11.

Part II

Alphabetic Reference

Part II consists of a topic for each of the OLAP DML statements, arranged alphabetically. Each OLAP DML topic provides a description, syntax, argument descriptions, notes, and example for the statement.

This part contains the following chapters:

6 $AGGMAP to AGGMAP

This chapter contains the following OLAP DML statements:

- MODEL (in an aggregation)
- RELATION (for aggregation)

- AGGMAP ADD or REMOVE model
- AGGMAP SET

$AGGMAP

The $AGGMAP property specifies the default AGGMAP type aggmap for a variable. When calculating the data in a variable, Oracle OLAP checks to see if the variable has an $AGGMAP property and, if it does, uses the aggmap object specified by that property as the default aggregation specification for a variable.

> **Tip:** You can also use a AGGMAP SET statement to specify the default aggregation specification for a variable or the $ALLOCMAP property to specify the default allocation specification for a variable.

> **See:** The PROPERTY command for general information on using properties in the OLAP DML.

Syntax

You add or delete an $AGGMAP property to the most recently defined or considered object (see DEFINE PROGRAM and CONSIDER) using a PROPERTY statement with the following syntax.

PROPERTY {*addproperty* | *deleteproperty*}

where

- *addproperty* has the following syntax.

  '$AGGMAP' *agggmap-name*

- *deleteproperty* has the following syntax.

  DELETE '$AGGMAP'

Arguments

aggmap-name
A `TEXT` expression that is the name of a previously defined aggmap object.

DELETE '$AGGMAP'
Deletes the $AGGMAP property.

Examples

Example 6–1 Using the $AGGMAP Property

Example 6–2, "Using the $AGGREGATE_FROM Property" on page 6-6 illustrates how the AGGREGATE command shown in Example 6–22, "Using a CACHE Statement in an Aggregation Specification" on page 6-52 can be simplified to the following statement.

```
AGGREGATE sales_by_revenue USING revenue_aggmap
```

You can further simplify the AGGREGATE command if you place an $AGGMAP property on the `sales_by_revenue` variable. To define an $AGGMAP property on the `sales_by_revenue` variable, issue the following statements.

```
CONSIDER sales_by_revenue
PROPERTY ('$AGGMAP' 'revenue_aggmap')
```

Now you can aggregate the data by issuing the following AGGREGATE command that does not include a USING clause.

```
AGGREGATE sales_by_revenue
```

$AGGREGATE_FROM

The $AGGREGATE_FROM property specifies the name of an object from which to obtain detail data when aggregating data. When aggregating the data in a variable, Oracle OLAP checks to see if the variable has an $AGGREGATE_FROM property and, if it does, obtains the detail data for the aggregation from the variable specified by that property.

> **See:** "Ways of Specifying Where to Obtain Detail Data for Aggregation" on page 7-13 for a discussion of all of the ways in which you can specify the variables from which detail data should be obtained when performing aggregation.
>
> The PROPERTY command for general information on using properties in the OLAP DML.

Syntax

You add or delete an $AGGREGATE_FROM property to the most recently defined or considered object (see DEFINE PROGRAM and CONSIDER) using a PROPERTY statement with the following syntax.

PROPERTY {*addproperty* | *deleteproperty*}

where

- *addproperty* has the following syntax.

  '$AGGREGATE_FROM' *fromspec* ACROSS *dimname*

- *deleteproperty* has the following syntax.

  DELETE '$AGGREGATE_FROM'

Arguments

fromspec
A `TEXT` expression that specifies an arbitrarily dimensioned variable, formula, or relation from which the detail data for the aggregation is obtained.

ACROSS *dimname*
Specifies the dimension or a named composite that the aggregation loops over to discover the cells in *fromspec*. Because *fromspec* can be a formula, you can realize a

significant performance advantage by supplying a looping dimension that eliminates the sparsity from the *fromspec* loop.

DELETE '$AGGREGATE_FROM'
Deletes the $AGGREGATE_FROM property.

Examples

Example 6–2 Using the $AGGREGATE_FROM Property

Example 6–22, "Using a CACHE Statement in an Aggregation Specification" on page 6-52 uses the following AGGREGATE command to aggregate the data.

```
AGGREGATE sales_by_revenue USING revenue_aggmap FROM units_aggmap
```

You can place a $AGGREGATE_FROM property on the `sales_by_revenue` variable by issuing the following statements.

```
CONSIDER sales_by_revenue
PROPERTY ('$AGGREGATE_FROM' 'units_aggmap')
```

Now you can aggregate the data by issuing the following AGGREGATE command that does not include a FROM clause.

```
AGGREGATE sales_by_revenue USING revenue_aggmap
```

$AGGREGATE_FROMVAR

The $AGGREGATE_FROMVAR property specifies two or more objects from which to obtain detail data when aggregating data. When aggregating the data in a variable, Oracle OLAP checks to see if the variable has an $AGGREGATE_FROMVAR property and, if it does, obtains the detail data for the aggregation from the variables specified by that property.

> **See:** "Ways of Specifying Where to Obtain Detail Data for Aggregation" on page 7-13 for a discussion of all of the ways in which you can specify the variables from which detail data should be obtained when performing aggregation.
>
> The PROPERTY command for general information on using properties in the OLAP DML

Syntax

You add or delete an $AGGGREGATE_FROMVAR property to the most recently defined or considered object (see DEFINE PROGRAM and CONSIDER) using a PROPERTY statement with the following syntax.

PROPERTY {*addproperty* | *deleteproperty*}

where

- *addproperty* has the following syntax.

  '$AGGREGATE_FROMVAR' *textvar* ACROSS *dimname*

- *deleteproperty* has the following syntax.

  DELETE '$AGGREGATE_FROMVAR'

Arguments

textvar
A `TEXT` expression that specifies an arbitrarily dimensioned variable or formula that specifies the names of the objects from which to obtain detail data when performing a capstone aggregation. Specify `NA` to indicate that a node does not need detail data to calculate the value.

ACROSS *dimname*
Specifies the dimension or a named composite that the aggregation loops over to discover the cells in the objects specified by *textvar*. Because the objects specified by *textvar* can be formulas, you can realize a significant performance advantage by supplying a looping dimension that eliminates the sparsity.

DELETE $AGGREGATE_FROMVAR
Deletes the $AGGREGATE_FROMVAR property.

Examples

Example 6–3 Capstone Aggregation Using the $AGGREGATE_FROMVAR Property

Example 7–7, "Capstone Aggregation" on page 7-17 uses the following AGGREGATE command to perform the final capstone aggregation.

```
AGGREGATE sales_capstone76 USING capstone_aggmap FROMVAR capstone_source
```

You can omit the FROMVAR clause in the second AGGREGATE command if there is a $AGGREGATE_FROMVAR property on the `sales_capstone76` variable.

To create a $FROMVAR property, issue the following OLAP DML statements.

```
CONSIDER sales_capstone76
PROPERTY ('$AGGREGATE_FROMVAR'  'capstone_source')
```

Now you can perform the final capstone aggregation by issuing the following statement.

```
AGGREGATE sales_capstone76 USING capstone_aggmap
```

$ALLOCMAP

The $ALLOCMAP property specifies the default aggmap for allocation for a variable. When calculating the data in a variable, Oracle OLAP checks to see if the variable has an $ALLOCMAP property and, if it does, uses the aggmap object specified by that property as the default allocation specification for a variable.

> **Note:** The $ALLOCMAP property is only one way in which you can specify a default aggmap for a variable. You can also use a $AGGMAP property or a AGGMAP SET statement to specify the default aggregation specification for a variable.

See: The PROPERTY command for general information on using properties in the OLAP DML.

Syntax

You add or delete an $ALLOCMAP property to the most recently defined or considered object (see DEFINE PROGRAM and CONSIDER) using a PROPERTY statement with the following syntax.

PROPERTY {*addproperty* | *deleteproperty*}

where

- *addproperty* has the following syntax.

  '$ALLOCMAP' *aggmap-name*

- *deleteproperty* has the following syntax.

  DELETE '$ALLOCMAP'

Arguments

aggmap-name
A `TEXT` expression that specifies the name of a previously defined ALLOCMAP type aggmap object.

DELETE $ALLOCMAP
Deletes the $ALLOCMAP property.

Examples

Example 6–4 Using $ALLOCMAP to Specify a Default allocation Specification

Example 7–16, "Recursive Even Allocation with a Lock" on page 7-41 uses the following statement to allocated data in the `projbudget` variable using the `projbudgmap` allocation specification.

```
ALLOCATE projbudget USING projbudgmap
```

You can specify that `projbudgmap` is the default allocation specification for the `projbudget` variable by issuing the following statements.

```
CONSIDER projbudget
PROPERTY ('$ALLOCMAP' "projbugmap')
```

Now, merely by issuing the following statement, you can allocate data in the `projbudget` variable using the `projbudgmap` allocation specification.

```
ALLOCATE projbudget
```

$COUNTVAR

The $COUNTVAR property specifies that Oracle OLAP counts the number of leaf nodes that contributed to an aggregate value when an AGGREGATE function executes. Leaf nodes that are `NA` are not included in the tally. Indicates that the number of leaf nodes that contributed to an aggregate value are counted. When calculating the data in a variable, Oracle OLAP checks to see if the variable has an $COUNTVAR property and, if it does, counts the number of leaf nodes that contributed to an aggregate value.

Note: The $COUNTVAR property is only one way in which you can specify that Oracle OLAP should count the number of leaf nodes that contributed to an aggregate value when an AGGREGATE function executes. You can also specify this by including a COUNTVAR phrase in the AGGREGATE function.

See: The PROPERTY command for general information on using properties in the OLAP DML.

Syntax

You add or delete a $COUNTVAR property to the most recently defined or considered object (see DEFINE PROGRAM and CONSIDER) using a PROPERTY statement with the following syntax.

PROPERTY {*addproperty* | *deleteproperty*}

where

- *addproperty* has the following syntax.

  '$COUNTVAR' *agggmap-name*

- *deleteproperty* has the following syntax.

  DELETE '$COUNTVAR'

Arguments

aggmap-name
A `TEXT` expression that specifies the name of a previously defined AGGMAP type aggmap object.

DELETE $COUNTVAR
Deletes the $COUNTVAR property.

Using $COUNTVAR

For a variable named `v1`, the following statements cause Oracle OLAP to count the number of leaf nodes that contributed to an aggregate value that is the result of the execution of the `myaggmap` aggmap object by a AGGREGATE function.

```
CONSIDER v1
PROPERTY '$COUNTVAR' 'myaggmap'
```

$NATRIGGER

The $NATRIGGER property specifies values to substitute for NA values that are in the object, but not in the session cache for the object (if any). To calculate the values, Oracle OLAP takes the steps described in "How Oracle OLAP Calculates Data for a Variable with NA Values" on page 6-14. The results of the calculation are either stored in the variable or cached in the session cache for the variable as described in "How Oracle OLAP Determines Whether to Store or Cache Results of $NATRIGGER" on page 6-21.

> **See:** The PROPERTY command for general information on using properties in the OLAP DML.

Syntax

You add or delete a $NATRIGGER property to the most recently defined or considered object (see DEFINE PROGRAM and CONSIDER) using a PROPERTY statement with the following syntax.

PROPERTY {*addproperty* | *deleteproperty*}

where

- *addproperty* has the following syntax.

  '$NATRIGGER' *value*

- *deleteproperty* has the following syntax.

  DELETE '$NATRIGGER'

Arguments

value
A TEXT expression that is the value of the property. The text can be any expression that is valid for defining a formula

DELETE $NATRIGGER
Deletes the $NATRIGGER property.

Notes

How Oracle OLAP Calculates Data for a Variable with NA Values

When calculating the data for a dimensioned variable, Oracle OLAP takes the following steps for each cell in the variable:

1. Is there is a session cache for the variable.
   - Yes. Go to step 2.
   - No. Go to step 3.
2. Does that cell in the session cache for the variable have an `NA` value.
   - Yes. Go to step 3.
   - No. Go to step 7.
3. Does that cell in variable storage have an `NA` value.
   - Yes. Go to step 4.
   - No. Go to step 7.
4. Does the variable have an $AGGMAP property?
   - Yes. Aggregate the variable using the aggmap specified for the $AGGMAP property and, then, go to step 5.
   - No. Go to step 6.
5. What is the value of the cell after aggregating the variable?
   - NA, go to step 6.
   - Non-NA, go to step 7.
6. Does the variable have a $NATRIGGER property?
   - Yes. Execute the expression specified for the $NATRIGGER property and, then, go to step 7.
   - No. Go to step 7.
7. Calculate the data.

Setting the $NATRIGGER Property on Objects that are not Dimensioned Variables

When you set the $NATRIGGER property on an object that is not a dimensioned variable, including a single-cell variable, then Oracle OLAP treats it as any other user-assigned property with no special meaning for NA values

Making NA Triggers Recursive or Mutually Recursive

You can make NA triggers recursive or mutually recursive by including triggered objects within the value expression. You must set the RECURSIVE option to `YES` before a formula, program, or other $NATRIGGER expression can invoke a trigger expression again while it is executing. For limiting the number of triggers that can execute simultaneously, see the TRIGGERMAXDEPTH option.

Using $NATRIGGER with Composites

You can set an `$NATRIGGER` expression on a variable that is dimensioned by a composite, but Oracle OLAP evaluates the `$NATRIGGER` expression only for the dimension-value combinations that exist in the composite. Suppose you had the following dimensions and variables defined.

```
DEFINE d1 DIMENSION INTEGER
DEFINE d2 DIMENSION INTEGER
DEFINE v1 DECIMAL <d1 d2>
DEFINE v2 DECIMAL <SPARSE <d1 d2>>
PROPERTY '$NATRIGGER' 'v1 + 500.0'
DEFINE v3 DECIMAL <SPARSE <d1 d2>>
```

The following statement is an example of looping over a composite.

```
v3 = v2 ACROSS <SPARSE <d1 d2>>
```

$NATRIGGER Takes Precedence over NAFILL or NA Options

Oracle OLAP evaluates an `$NATRIGGER` property expression before applying the NAFILL function or the NASKIP, NASKIP2, or NASPELL options. When the `$NATRIGGER` expression is `NA`, then the NAFILL function and the `NA` options have an effect.

$NATRIGGER Ignored by EXPORT, ROLLUP, and AGGREGATE

The ROLLUP command, AGGREGATE command, and the AGGREGATE function ignore the $NATRIGGER property setting for a variable during a rollup operation. The statements fetch the stored value only, and do not invoke the $NATRIGGER expression. The $NATRIGGER property remains in effect for other operations.

In executing an EXPORT (to EIF) command, Oracle OLAP does not evaluate the $NATRIGGER property expression on a variable when it simply exports the variable. However, Oracle OLAP does evaluate the $NATRIGGER property expression when the variable is part of an expression that Oracle OLAP calculates during the export operation. Suppose you had the following `d1` dimension and `v1` variable definitions.

```
DEFINE d1 INTEGER DIMENSION
MAINTAIN d1 ADD 2
DEFINE v1 DECIMAL <d1>
PROPERTY '$NATRIGGER' '500'
```

For the following statement, Oracle OLAP would not evaluate the `$NATRIGGER` property expression for the `v1` variable. It would export the `$NATRIGGER` property as part of the description of the variable. The value in `v2` would be `NA`.

```
EXPORT v1 AS v2 TO EIF FILE 'myeif.eif'
```

For the following statement, Oracle OLAP would evaluate the `$NATRIGGER` property expression for the `v1` variable. The value in `v1plus1` would be `501`.

```
EXPORT v1 + 1 AS v1plus1 TO EIF FILE 'myeif.eif'
```

Examples

Example 6–5 Triggering Aggregation Using an $NATRIGGER

Instead of specifying the AGGREGATE function in every statement that you want to return aggregate data, you use the $NATRIGGER property to cause the aggregation to occur when a cell in the variable has an NA value. To use $NATRIGGER for this purpose, assign an $NATRIGGER property to the variable with a call to the AGGREGATE function specified as the $NATRIGGER expression.

The following statements add the $NATRIGGER property to the `sales` variable, so that unsolved data is aggregated using the `sales.aggmap` aggmap.

```
CONSIDER sales
PROPERTY '$NATRIGGER' 'AGGREGATE(sales USING sales.aggmap)'
```

Example 6–6 Adding an $NATRIGGER Property to a Variable

The following statements define a dimension with three values and define a variable that is dimensioned by the dimension. They add the $NATRIGGER property to the variable, then put a value in one cell of the variable and leave the other cells empty so their values are NA. Finally, they report the values in the cells of the variable.

```
DEFINE d1 INTEGER DIMENSION
MAINTAIN d1 ADD 3
DEFINE v1 DECIMAL <d1>
PROPERTY '$NATRIGGER' '500.0'
v1(d1 1) = 333.3
REPORT v1
```

The preceding statements produce the following output.

```
D1            V1
--------- ----------
        1     333.3
        2     500.0
        3     500.0
```

$STORETRIGGERVAL

The $STORETRIGGERVAL property specifies that when a $NATRIGGER expression executes, Oracle OLAP replaces the `NA` values in the variable with the results of the expression.

Note: Most applications use the $VARCACHE property to specify that Oracle OLAP should store variable data that is the result of $NATRIGGER expression execution since the functionality of the $STORETRIGGERVAL property is subsumed within the $VARCACHE property.

See: "How Oracle OLAP Determines Whether to Store or Cache Results of $NATRIGGER" on page 6-21 for a discussion of all of the factors that Oracle OLAP uses to determine what to do with variable data that is the result of $NATRIGGER expression execution.

The PROPERTY command for general information on using properties in the OLAP DML.

Syntax

You add or delete a $STORETRIGGERVAL property to the most recently defined or considered object (see DEFINE PROGRAM and CONSIDER) using a PROPERTY statement with the following syntax.

PROPERTY {*addproperty* | *deleteproperty*}

where

- *addproperty* has the following syntax.

  '$STORETRIGGERVAL' *value*

- *deleteproperty* has the following syntax.

  DELETE '$STORERIGGERVAL'

Arguments

value
A BOOLEAN expression that contains the value of the property.

DELETE $STORETRIGGERVAL
Deletes the $STORETRIGGERVAL property.

You can also delete the $STORETRIGGERVAL property along with all properties from an object by issuing a `PROPERTY DELETE ALL` statement.

Notes

Relationship With the TRIGGERSTOREOK Option and $NATRIGGER Property

To permanently replace the `NA` values in the cells of a variable for which you have set an $AGGREGATE_FROM property, you must set the TRIGGERSTOREOK option setting to `YES` and you must set the $STORETRIGGERVAL property of the variable to the Boolean value `YES`. When the $STORETRIGGERVAL property for the variable has a Boolean value of `NO`, then Oracle OLAP does not replace the `NA` values in the cells of the variable with the $NATRIGGER expression value.

When the value of the TRIGGERSTOREOK option is `NO`, then Oracle OLAP does not replace the `NA` values in the cells of the variable with the $AGGREGATE_FROM expression value, even when the value of the $STORETRIGGERVAL property for the variable is `YES`.

Examples

Example 6–7 Storing an $NATRIGGER Property Value

The following statements cause Oracle OLAP to store the $NATRIGGER expression value in the `NA` cells of the `v1` variable when Oracle OLAP evaluates the expression.

```
TRIGGERSTOREOK = yes
CONSIDER v1
PROPERTY '$STORETRIGGERVAL' yes
```

$VARCACHE

The $VARCACHE property specifies whether Oracle OLAP stores or caches variable data that is the result of the execution of a AGGREGATE function or $NATRIGGER expression.

> **See:** "How Oracle OLAP Determines Whether to Store or Cache Aggregated Data" on page 6-23 and "How Oracle OLAP Determines Whether to Store or Cache Results of $NATRIGGER" on page 6-21 for discussions of all of the various factors that Oracle OLAP uses to determine whether variable data computed when the AGGREGATE function or $NATRIGGER property executes is stored or cached.
>
> The PROPERTY command for general information on using properties in the OLAP DML.

Syntax

You add or delete a $VARCACHE property to the most recently defined or considered object (see DEFINE PROGRAM and CONSIDER) using a PROPERTY statement with the following syntax.

PROPERTY {*addproperty* | *deleteproperty*}

where

- *addproperty* has the following syntax.

  '$VARCACHE' *value*

- *deleteproperty* has the following syntax.

  DELETE '$VARCACHE'

Arguments

value

One of the following TEXT expressions that indicate where Oracle OLAP should place variable data that is the result of calculations performed when the AGGREGATE function or $NATRIGGER value executes:

- **VARIABLE** specifies that Oracle OLAP populates the variable with data that is the result of the execution of the AGGREGATE function or $NATRIGGER property. When you specify this option, the data that is the result of the aggregation is permanently stored in the variable when the analytic workspace is updated and committed.
- **SESSION** specifies that Oracle OLAP caches data that is the result of the execution of the AGGREGATE function or $NATRIGGER property in the session cache (See "What is an Oracle OLAP Session Cache?" on page 21-54). When you specify this option, the data that is the result of the execution of the AGGREGATE function or $NATRIGGER property is ignored during updates and commits and is discarded at the end of the session.

> **Important:** When SESSCACHE is set to `NO`, Oracle OLAP does not cache the data even when you specify `SESSION`. In this case, specifying `SESSION` is the same as specifying `NONE`.

- **NONE** specifies that Oracle OLAP calculates new variable data each time the AGGREGATE function or $NATRIGGER value executes; Oracle OLAP does not store or cache the data.
- **DEFAULT** specifies that you do not want Oracle OLAP to use the $VARCACHE property when determining what to do with data that is calculated by the AGGREGATE function. (See "How Oracle OLAP Determines Whether to Store or Cache Aggregated Data" on page 6-23.)

DELETE $VARCACHE
Deletes the $VARCACHE property.

You can also delete the $VARCACHE property along with all properties from an object by issuing a `PROPERTY DELETE ALL` statement.

Notes

How Oracle OLAP Determines Whether to Store or Cache Results of $NATRIGGER

When a $NATRIGGER expression executes, what Oracle OLAP does with variable data that results from the execution of the expression is determined based on whether or not the variable that has the $NATRIGGER property also has a $STORETRIGGERVAL property and, if not, if the value of the $NATRIGGER property is an AGGREGATE function.

When a a $NATRIGGER expression executes, Oracle OLAP goes through the following process:

1. Does the variable with the $NATRIGGER property also have a $STORETRIGGERVAL property?

   Yes. Go to step 1a.

   No. Go to step 2.

   a. Is the value of the TRIGGERSTOREOK option, `'YES'` or 'NO?

      Yes. Go to step 1b.

      No. Go to step 2.

   b. Is the value of the $STORETRIGGERVAL property, `YES` or `NO`?

      Yes. Store the results of the $NATRIGGER expression. End decision-making process.

      No. Do *not* store the results of the $NATRIGGER expression. End decision-making process

2. Is the $NATRIGGER expression is an AGGREGATE function?

   Yes. Follow the steps described in "How Oracle OLAP Determines Whether to Store or Cache Aggregated Data" on page 6-23 to determine what to do with the result of $NATRIGGER expression execution.

   No. Go to step 3.

3. Does the variable with the $NATRIGGER property also have a $VARCACHE property?

   Yes. Go to step 4.

   No. Go to step 5.

4. Does the $VARCACHE property have a value of `DEFAULT`?

   Yes. Go to step 5.

   No. Use the value of the $VARCACHE property (that is, `STORE`, `CACHE`, or `NONE`) to determine what happens to the variable data values that are the result of $NATRIGGER expression execution. End decision-making process.

5. Use the current setting of the VARCACHE option to determine what happens to the variable data values that are the result of $NATRIGGER expression execution. End decision-making process.

See also: The following topics:

- The description of the NA keyword of the CACHE command for information on caching NA values calculated by the AGGREGATE function.
- "What is an Oracle OLAP Session Cache?" on page 21-54 for a description of the data that is in a session cache.

How Oracle OLAP Determines Whether to Store or Cache Aggregated Data

When an AGGREGATE command executes, Oracle OLAP *always* stores the results of the calculation directly in the variable in the same way it stores the results of an assignment statement.

However, when an AGGREGATE function executes, Oracle OLAP sometimes stores the results of the calculation directly in the variable and sometimes caches it in the session cache. (See "What is an Oracle OLAP Session Cache?" on page 21-54 for more information about the session cache.)

To determine where to place the data that is the result of AGGREGATE function execution, Oracle OLAP goes through the following process to determine whether to store or cache aggregated variable data:

1. Is there a CACHE statement in the specification for the aggmap that is being used by the current AGGREGATE function?
   - Yes. Go to step 2.
   - No. Go to step 3.
2. Is the CACHE statement a CACHE DEFAULT statement?
   - Yes. Go to step 3.
   - No. Use the CACHE statement in the aggregation specification to determine what to do with variable data that is the result of the calculation. End decision-making process.
3. Does the variable being aggregated have a $VARCACHE property?
   - Yes. Go to Step 4.
   - No. Go to step 5.

4. Does the $VARCACHE property have a value of DEFAULT?
   - Yes. Go to step 5.
   - No. Use the value of the $VARCACHE property determines what happens to the variable data calculated using the AGGREGATE function. End decision-making process.
5. Use the current setting of the VARCACHE option to determine what happens to the variable data calculated using the AGGREGATE function. End decision-making process.

Examples

Example 6–8 Setting the $VARCACHE Property

- For a variable named `v1`, the following statements cause Oracle OLAP to cache the variable data that is the result of the execution of an AGGREGATE function or $NATRIGGER expression.

```
CONSIDER v1
PROPERTY '$SVARCACHE' 'v1'
```

ABS

The ABS function calculates the absolute value of an expression.

Return Value

DECIMAL.

The dimensionality of the result is the same as the specified expression.

Syntax

ABS(*expression*)

Arguments

expression
The expression whose absolute value is to be calculated.

Examples

Example 6–9 Finding Values in an Absolute Range

Suppose you are interested in how close your planned 1996 sales figures for sportswear in Boston were to the actual sales. You would like to see those months where budgeted figures are off by more than $5,000 in either direction. You can use ABS to help you find those months.

```
LIMIT product TO 'Sportswear'
LIMIT district TO 'Boston'
LIMIT month TO YEAR 'Yr96'
LIMIT month KEEP ABS(sales - sales.plan) GT 5000
REPORT DOWN month sales sales.plan sales - sales.plan
```

These statements produce the following output.

```
DISTRICT: BOSTON
               ------------PRODUCT-------------
               -----------SPORTSWEAR-----------
                                      SALES -
MONTH            SALES    SALES.PLAN SALES.PLAN
-------------- ---------- ---------- ----------
Jun96           79,630.20  73,568.52   6,061.68
Jul96           95,707.30  80,744.18  14,963.12
Aug96           82,004.00  71,811.45  10,192.55
Sep96           89,988.60  78,282.07  11,706.53
Dec96           50,281.40  56,720.87  -6,439.47
```

ACQUIRE

When an analytic workspace is attached in multiwriter mode, the ACQUIRE command acquires and (optionally) resynchronizes the specified objects so that their changes can be updated and committed.

Syntax

```
ACQUIRE [acquired_noresync_objects] [RESYNC resync_objects  [WAIT]] -
    [CONSISTENT WITH consistency_objects [WAIT]]
```

Arguments

acquired_noresync_objects
A list of one or more variables, relations, valuesets, or dimension names, separated by commas, that you want to acquire without resynchronizing. Acquiring objects in this manner preserves all read-only changes made to the objects. You can update variables and dimensions acquired in this manner using the UPDATE command.

RESYNC
Specifies acquisition of the latest generation of the specified objects with all private changes discarded.

***resync_objects* [WAIT]**
A list of one or more variables, relations, valuesets, or dimension names, separated by commas, that you want to acquire and resynchronize.

When you do not specify WAIT, the ACQUIRE statement fails when another user has acquired any of the objects in *resync_objects* in read/write mode. When you specify WAIT, Oracle OLAP waits until all objects in *resync_objects* it can be acquired or the wait times out.

CONSISTENT WITH
Specifies the behavior of the ACQUIRE statement when a specified object is already acquired by another user and resynchronizes the specified objects when the ACQUIRE statement succeeds.

***consistency_objects* [WAIT]**
A list of one or more a list of one or more variables, relations, valuesets, or dimension names, separated by commas, that you want Oracle OLAP to determine if another user has already acquired.

When you do not specify WAIT, the ACQUIRE statement fails when any of the objects in the *consistency_objects* are acquired by another user. When you specify the WAIT keyword, Oracle OLAP waits to execute the ACQUIRE statement until none of the objects in *consistency_objects* are acquired by another user or until the wait times out.

Notes

Understanding Consistency

To some extent you can think of an ACQUIRE statement with a CONSISTENT WITH phrase as a combination of ACQUIRE and RELEASE statements.

```
ACQUIRE [avar...] RESYNC [rvar ...] cvar ... [WAIT]
RELEASE cvar ...
```

The difference is that an ACQUIRE CONSISTENT WITH statement succeeds even when the user does not have sufficient permissions to acquire `cvar` variables.

Failure and Error-Handling

When a specified object has been acquired by another user or when your read-only generation for a specified object is not the latest generation for the object, the ACQUIRE statement fails.

Also, it can take a long time for the ACQUIRE statement to complete when you specify WAIT for either the RESYNC or CONSISTENT phrase. During the wait, some variables in the acquisition lists may be released while others may have been acquired. It is even possible for a deadlock to occur which causes the ACQUIRE statement to fail with a timeout error.

To avoid problems caused by deadlock, be thoughtful about the order in which you code ACQUIRE and RELEASE statements and include appropriate error handling routines.

Whenever an ACQUIRE statement fails, none of the objects are acquired. Consequently, either all of the requested objects are acquired or none of them are acquired.

Acquiring Objects Dimensioned by Composites

You can acquire any variable, valueset, dimension, or relation that is dimensioned by a composite unless that composite has a composite as one of its base dimensions.

Examples

Example 6–10 Acquiring, Updating and Releasing Objects

A classic use of multiwriter attachment mode is to allow two users to modify two different objects in the same analytic workspace. For example, assume that an analytic workspace has two variables: `actuals` and `budget`. Assume also that one user (user A) wants to modify `actuals`, while another user (user B) wants to modify `budget`. In this case, after attaching the analytic workspace in the multiwriter mode, each user acquires the desired variable, performs the desired modification, updates, commits the changes, and then, either detaches the workspace or releases the acquired variable.

User A executes the following statements.

```
AW ATTACH myworkspace MULTI
ACQUIRE actuals
... make modifications
UPDATE MULTI actuals
COMMIT
RELEASE actuals
AW DETACH myworkspace
```

While, at the same time, User B executes the following statements.

```
AW ATTACH myworkspace MULTI
ACQUIRE budget
…make modifications
UPDATE MULTI budget
COMMIT
RELEASE budget
AW DETACH myworkspace
```

Example 6–11 Acquiring and Resynchronizing Objects

Assume that two users (named B1 and B2) both need to make what-if changes to `budget` and possibly modify their parts of `budget` when they like the results of the what-if changes Neither user knows if anyone else will need to access `budget` at the same time that they are or if they will need to make any permanent changes to `budget`. Consequently, they do not want to block anyone while they are performing what-if changes.

In this case, both users perform their what-if computation after attaching the analytic workspace in the multiwriter mode but without acquiring `budget`. When they later decide to make their what-if changes permanent, they try to acquire budget in unresynchronized mode. When the acquire succeeds, they update `budget` and commit the changes. The following OLAP DML statements show this scenario.

```
AW ATTACH myworkspace MULTI
...perform what-if computations
ACQUIRE budget
...maybe make some additional final changes
UPDATE MULTI budget
COMMIT
RELEASE budget
AW DETACH myworkspace
```

However, when the first acquire does not succeed, however, the users try again to acquire `budget` in resynchronized mode (possibly requesting a wait). When the resynchronized acquisition succeeds, they re-create the changes (since some relevant numbers might have changed) and then proceed to update and commit their analytic workspace. The following OLAP DML statements show this scenario.

```
AW ATTACH myworkspace MULTI
... perform what-if computations
ACQUIRE budget
...maybe make some additional final changes
UPDATE MULTI budget
COMMIT
RELEASE budget
AW DETACH myworkspace
AW ATTACH myworkspace MULTI
...perform what-if computations
ACQUIRE budget --> failed
ACQUIRE RESYNC budget WAIT
...determine that the changes are still needed
...make changes to make permanent
UPDATE MULTI budget
COMMIT
RELEASE budget
AW DETACH myworkspace
```

Example 6–12 Acquiring Objects While Keeping Consistency

Sometimes you must keep some objects consistent with each other, which requires special care in multiwriter mode.

Assume that two users (User B1 and User B2) both need to modify `budget`, that `budget` must be kept consistent with `investment`, and that another user (User I) needs to modify `investment`. In this scenario, even though none of the users needs to modify both `budget` and `investment`, they all must ensure that when they acquire either `budget` or `investment` that no one else has either budget or investment already acquired. To achieve this effect, each user must issue an ACQUIRE statement with the CONSISTENT WITH phrase as shown in the following example code. Note that all of the users must be aware that the objects listed in the CONSISTENT phrase may be resynchronized by the ACQUIRE statement, if needed.

For example, User B1 could issue the following OLAP DML statements.

```
AW ATTACH myworkspace MULTI
... perform what-if computations
ACQUIRE budget CONSISTENT WITH investment
... maybe make some additional final changes
UPDATE MULTI budget
COMMIT
RELEASE budget, investment
AW DETACH myworkspace
```

User B2 could issue the following OLAP DML statements.

```
AW ATTACH myworkspace MULTI
... perform what-if computations
ACQUIRE budget CONSISTENT WITH investment --> failed
ACQUIRE RESYNC budget CONSISTENT WITH investment WAIT
... determine that the changes are still needed
... make changes to make permanent
UPDATE MULTI budget
COMMIT
RELEASE budget, investment
AW DETACH myworkspace
```

User I could issue the following OLAP DML statements.

```
AW ATTACH myworkspace MULTI
ACQUIRE investment CONSISTENT WITH budget --> failed
ACQUIRE RESYNC investment CONSISTENT WITH budget WAIT
... make changes to investment
UPDATE MULTI investment
COMMIT
RELEASE budget, investment
AW DETACH myworkspace
```

ACROSS

The ACROSS command specifies a text expression that contains one or more statements to be executed in a loop. The repetition of these statements is controlled by the status of the dimensions and composites specified in the ACROSS command.

Syntax

ACROSS *dimension*... DO *dml-statements*

Arguments

dimension

One or more dimensions or composites whose current status controls the repetition of one or more statements, which are contained in *dml-statements*. The statements are repeated for each combination of the values of the specified dimensions in the current status. When two or more dimensions are specified, the first one varies the slowest.

DO *dml-statements*

A multiline text expression that is one or more OLAP DML statements to be executed for each iteration of the loop.

Notes

Statements That You Cannot Use

You cannot specify statements in the text expression of ACROSS that are typically used as part of a multiple-line construct in a program. For example, the IF...THEN...ELSE, WHILE, FOR, or SWITCH statements cannot be executed by the ACROSS command.

Code Compiles When Executed

The compiled code for the loop body will not be generated until the ACROSS command or the program that contains it is executed. This allows for the possibility that, because the statements are contained within a text expression, the contents of an ACROSS loop may change between compilation and execution.

ACROSS Dimension

ACROSS temporarily sets status to the values that are in current status for the specified dimensions. After the ACROSS statement executes, dimension status is restored to what it was before the loop, and execution of the program resumes with the next statement.

Examples

Example 6–13 Repeating ROW Commands

In a report program, you want to show the unit sales of tents for each of three months. Use the following ACROSS command to repeat ROW commands for each value of the `month` dimension.

```
LIMIT product TO 'Tents'
LIMIT month TO 'Jan95' to 'Mar95'
ACROSS month DO 'ROW INDENT 5 month WIDTH 6 unit'

     Jan95     533363
     Feb95     572796
     Mar95     707198
```

ADD_MONTHS

The ADD_MONTHS function returns the date that is *n* months after the specified date.

Return Value

DATETIME

Syntax

ADD_MONTHS(*start_datetime*, *n*)

Arguments

start_datetime
A DATETIME expression that identifies the starting date.

n
An INTEGER that identifies the number of months to be added to *start_datetime*.

Notes

End of Month
When the day component of *start_datetime* is the last day of the month or when the returned month has fewer days, then the returned day component is the last day of the month. Otherwise, the day component of the returned date is the same as the day component of *start_datetime*. See Example 6–14, "End-of-Month Calculation" on page 6-35.

Examples

Example 6–14 End-of-Month Calculation

The following statement displays the date of the day that is one month after January 30, 2000.

```
SHOW ADD_MONTHS('30Jan00', 1)
```

Since February 29 was the last day of February 2000, ADD_MONTHS returns February 29, 2000.

```
29-Feb-00
```

AGGMAP

The AGGMAP command identifies an aggmap object as a specification for aggregation and adds an aggregation specification to the definition of the current aggmap object.

Tip: The current aggmap definition is the definition of the aggmap object most recently defined or considered. Issue a CONSIDER statement to explicitly make an aggmap definition the current aggmap definition.

Note: There are two other OLAP DML statements that are also sometimes referred to as "AGGMAP statements":

- The AGGMAP ADD or REMOVE model statement that you can use to add or remove a model from an aggmap object of type AGGMAP
- The AGGMAP SET that you can use to specify the default aggmap for a variable.

Syntax

AGGMAP [*specification*]

Arguments

specification

A multiline text expression that is the aggregation specification for the current aggmap object. Separate statements with newline delimiters (`\n`), or use JOINLINES. An aggregation specification begins with an ALLOCMAP statement and ends with an `END` statement. Between these statements, you code one or more the following statements depending on the calculation that you want to specify. Minimally, an aggregation specification consists of a RELATION (for aggregation) statement. You can create more complex aggregation specifications by including one or more of the following statements in the specification as described in "Aggregations" on page 4-2:

AGGINDEX
BREAKOUT DIMENSION
CACHE
DIMENSION (for aggregation)
DROP DIMENSION
MEASUREDIM (for aggregation)
MODEL (in an aggregation)
RELATION (for aggregation)

Note: Special considerations apply when you are writing a specification to aggregate a variable dimensioned by a compressed composite. See "Aggregating Variables Dimensioned by Compressed Composites" on page 6-38.

You cannot specify a conjoint dimension in the specification for the aggmap; use composites instead.

Notes

Aggregating Variables Dimensioned by Compressed Composites

Keep the following points in mind when designing aggregating values in a variable dimensioned by a compressed composite:

- You must use the AGGREGATE command to aggregate data in a variable dimensioned by a compressed composite; you cannot use the AGGREGATE function.
- When coding the aggregation specification, follow these guidelines:
  - MODEL statements must precede RELATION statements.
  - MODEL statements must either exclude a PRECOMPUTE clause or specify PRECOMPUTE ALL. You cannot include a dynamic model in the aggregation specification.
  - RELATION statements for the hierarchical dimensions in the compressed composite must include a PRECOMPUTE clause.
- Once the data in a variable dimensioned by a compressed composite is aggregated, it is read-only-data. Before you can recalculate values using the

AGGREGATE command, you must first delete the values using the CLEAR AGGREGATES command.

- There is no support for parallel aggregation. Instead, use multiple sessions to compute variables or partitions that have their own compressed composites.
- There is no support for incremental aggregation. Instead, partition using a dense time dimension with local compressed composites. In this way you can aggregate only those partitions that contain new data.

Aggregation Options

A number of options can impact aggregation as outlined in Table 6–1, " Aggregation Options" on page 6-39.

Table 6–1 Aggregation Options

Statement	Description
MULTIPATHHIER	An option that specifies that a given cell that contains detail data can have more than one path into a cell that contains aggregated data.
POUTFILEUNIT	An option that identifies a destination for status information about an aggregation operation.
SESSCACHE	An option that that controls whether Oracle OLAP uses the session cache to store variable data that has been aggregated on the fly.
VARCACHE	An option that specifies if and where variable data that has been aggregated on the fly should be stored. (The VARCACHE option is only one factor that can determine this decision.)

System Properties Related to Aggregation

Table 6–2, " System Properties Used When Aggregating or Allocating Data" on page 6-39 lists system properties documented in this manual that relate to aggregation or allocation.

Table 6–2 System Properties Used When Aggregating or Allocating Data

Property	Description
$AGGMAP	A property that specifies the default aggregation specification for a variable.
$AGGREGATE_FROM	A property that specifies the name of an object from which to obtain detail data when aggregating data.

Table 6–2 (Cont.) System Properties Used When Aggregating or Allocating Data

Property	Description
$AGGREGATE_FROMVAR	A property that specifies the objects from which to obtain detail data when aggregating data.
$ALLOCMAP	A property that specifies the default allocation specification for a variable.
$COUNTVAR	A property that specifies that Oracle OLAP counts the number of leaf nodes that contributed to an aggregate value when an AGGREGATE function executes.
$VARCACHE	A property that specifies whether Oracle OLAP stores or caches variable data that is the result of the execution of a AGGREGATE function or $NATRIGGER expression .

Checking for Circularity

AGGREGATE automatically checks relations for circularity in and among multiple hierarchies. When you first define hierarchies, check for circularity by setting PRECOMPUTE statements to `NA` and AGGINDEX to `NO`. A XSHIERCK01 error during aggregation indicates that a circular hierarchy may have been detected. However, when the message includes a reference to UNDIRECTED, then multiple paths to an ancestor from a detail data cell have been detected. Some calculations require that a detail data cell use multiple paths to the same ancestor cell. When this is the case, then you need to set the MULTIPATHHIER option to `YES` before you execute the AGGREGATE command. Otherwise, you need to correct the error in the hierarchy structure. For more details about this error message and how to interpret it, see the MULTIPATHHIER option.

Examples

Example 6–15 Combining Pre-calculation and Calculation on the Fly

This example describes the steps you can take to pre-calculate some of the data in your analytic workspace and specify that the rest should be calculated when users request it.

Suppose you define an analytic workspace named `mydtb` that has a `units` variable with the following definition.

```
DEFINE units INTEGER <time, SPARSE <product, geography>>
```

You now need to create and add a specification to the aggmap, which will specify the data that should be aggregated. This example shows you how to use an input file, which contains OLAP DML statements that define the aggmap and add a specification to it:

1. Identify the name of each dimension's hierarchy. When you have defined the hierarchies as self-relations, you use the names of the self-relations.
2. Decide which data to aggregate.

   Suppose you want to calculate data for all levels of the `time` and `product` dimensions, but not for `geography`. The `geography` dimension's lowest level of data is at the city level. The second level of the hierarchy has three dimension values that represent regions: `East`, `Central`, and `West`. The third level of the hierarchy has one dimension value: `Total`.

   Suppose that you want to pre-calculate the data for `East` and store it in the analytic workspace. You want the data for `Central`, `West`, and `Total` to be calculated only when users request that data — that data will not be stored in the analytic workspace. Therefore, you need to specify this information in the specification that you add to your aggmap object.
3. Create an ASCII text file named `units.txt`. Add the following OLAP DML statements to your text file.

   ```
   DEFINE units.agg AGGMAP <time, SPARSE <product, geography>>
   AGGMAP
   RELATION myti.parent
   RELATION mypr.parent
   RELATION myge.parent PRECOMPUTE ('East')
   END
   ```

   The preceding statements define an aggmap named `units.agg`, then add the three RELATION statements to the aggregation specification when you read the units.txt file into your analytic workspace.
4. To read the `units.txt` file into your analytic workspace, execute the following statement.

   ```
   INFILE 'units.txt'
   ```
5. The `units.agg` aggmap should now exist in your analytic workspace. You can aggregate the `units` variable with the following statement.

   ```
   AGGREGATE units USING units.agg
   ```

Now the data for `East` for all times and products has been calculated and stored in the analytic workspace.

6. Set up the analytic workspace so that when a user requests data for `Central`, `West`, or `Total`, that data will be calculated and displayed. It is generally a good idea to compile the aggmap object before using it with the AGGREGATE function, as shown by the following statement.

   ```
   COMPILE units.agg
   ```

   This is not an issue when you are just using the AGGREGATE command, because the command compiles the aggmap object before it uses it. However, when you do not use the FUNCDATA keyword with the AGGREGATE command, the metadata that is needed to perform calculation on the fly has not been compiled yet. As long as you have performed all other necessary calculations (such as calculating models), it is a good practice to compile the aggmap when you load data. When you fail to do so, that means that every time a user opens the analytic workspace, that user will have to wait for the aggregation to be compiled automatically. In other words, when any data will be calculated on the fly, you can improve query performance for all of your users by compiling the aggmap before making the analytic workspace available to your users.

7. Add a property to the `units` variable.

   ```
   CONSIDER units
   PROPERTY '$NATRIGGER' 'AGGREGATE(units USING units.agg)'
   ```

   This property indicates that when a data cell contains an `NA` value, Oracle OLAP will call the AGGREGATE function to aggregate the data for that cell. Therefore, any `units` data that is requested by a user will be displayed. However, only the data for the `East` dimension value of the `geography` dimension has actually been aggregated and stored in the analytic workspace. All other data (for `Central`, `West`, and `Total`) is calculated only when users request it.

Example 6–16 Performing Non-additive Aggregation

This example shows how to use operators and arguments to combine additive and non-additive aggregation.

Suppose that you have defined four variables: `sales`, `debt`, `interest_rate`, and `inventory`. The variables have been defined with the same dimensionality where `cp` is a composite that has been defined with the `product` and `geography` dimensions.

```
<time cp<product geography>>
```

Suppose you want to use one AGGREGATE command to aggregate all four variables. The `debt` variable requires additive aggregation. The `sales` variable requires a weighted sum aggregation, and `interest_rate` requires a hierarchical weighted average. Therefore, `sales` and`interest_rate` will each require a weight object, which you need to define and populate with weight values. `inventory` requires a result that represents the total inventory, which is the last value in the hierarchy.

You will specify the aggregation operation for `debt` and `inventory` with the OPERATOR keyword. However, because `sales` and `interest_rate` have aggregation operations that require weight objects, you must use the ARGS keyword to specify their operations. You define an operator variable to use the OPERATOR keyword. Typically, the operator variable is dimensioned by a measure dimension or a line item dimension.

Here are the steps to define the aggregation you want to occur:

1. Because you will also be using a measure dimension to define an argument variable to use with the ARGS keyword, define that `measure` dimension, as illustrated by the following statements.

   ```
   DEFINE measure DIMENSION TEXT
   MAINTAIN measure 'sales', 'debt', 'interest_rate', 'inventory'
   ```

   Note: Whenever you use a `measure` dimension in a RELATION statement, you must include a MEASUREDIM statement in the same aggregation specification

2. Define an operator variable named `opvar` and populate it.The statements specify that the aggregation for `debt` should use the `SUM` operator, and the aggregation for `inventory` should use the `HLAST` operator.

   ```
   DEFINE opvar TEXT <measure>
   opvar (measure 'sales') = 'WSUM'
   opvar (measure 'debt') = 'SUM'
   opvar (measure 'interest_rate') = 'HWAVERAGE'
   opvar (measure 'inventory') = 'HLAST'
   ```

3. Because `sales` and `interest_rate` require weight objects, define and populate those weight objects. The following statement defines a weight object named `currency` (to be used by `sales`).

   ```
   DEFINE currency DECIMAL <time geography>
   ```

   Notice that the `currency` variable is dimensioned only by `time` and `geography`. The purpose of this variable is to provide weights that act as currency conversion information for foreign countries; therefore, it is unnecessary to include the `product` dimension.

4. Populate `currency` with the weight values that you want to use.

5. The `interest_rate` variable's nonaddictive aggregation (hierarchical weighted average) requires the sum of the variable `debt`. In other words, `interest_rate` cannot be aggregated without the results of the aggregation of `debt`.

   You can now define an argument variable, which you will need to specify the aggregation results of `debt` as a weight object for `interest_rate`. You will use the same argument variable to specify `currency` as the weight object for the `sales` variable. The following statement defines an argument variable named `argvar`.

   ```
   DEFINE argvar TEXT <measure>
   ```

6. The next few statements populate the argument variable.

   ```
   argvar (measure 'sales') = 'weightby currency'
   argvar (measure 'debt') = NA
   argvar (measure 'interest_rate') = 'weightby debt'
   argvar (measure 'inventory') = NA
   ```

7. For the aggregation of `product` and `geography`, the data for the `sales`, `debt`, and `interest_rate` variables can simply be added. But the `inventory` variable requires a hierarchical weighted average. This means that

it is necessary to define a second operator variable and a second argument variable, both of which will be used in the RELATION statement for `product` and `geography`.

The following statements define the second operator variable and populate it.

```
DEFINE opvar2 TEXT <measure>
opvar (measure 'sales') = 'Sum'
opvar (measure 'debt') = 'Sum'
opvar (measure 'interest_rate') = 'Sum'
opvar (measure 'inventory') = 'HWAverage'
```

The following statements define the second argument variable and populate it.

```
DEFINE argvar2 TEXT <measure>
argvar (measure 'sales') = NA
argvar (measure 'debt') = NA
argvar (measure 'interest_rate') = NA
argvar (measure 'inventory') = 'weightby debt'
```

8. Now create the aggmap, by issuing the following statements.

```
DEFINE sales.agg AGGMAP <time, CP<product geography>>
AGGMAP
RELATION time.r OPERATOR opvar ARGS argvar
RELATION product.r OPERATOR opvar2 ARGS argvar2
RELATION geography.r OPERATOR opvar2 ARGS argvar2
MEASUREDIM measure
END
```

9. Finally, use the following statement to aggregate all four variables.

```
AGGREGATE sales debt interest_rate inventory USING sales.agg
```

Example 6–17 Programmatically Defining an Aggmap

The following program uses the EXISTS function to test whether an AGGMAP already exists, and defines the AGGMAP when it does not. It then uses the AGGMAP command to define the specification for the aggmap.

```
DEFINE MAKEAGGMAP PROGRAM
LD Create dynamic aggmap
PROGRAM
IF NOT EXISTS ('test.agg')
   THEN DEFINE test.agg AGGMAP <geography product channel time>
   ELSE CONSIDER test.agg
AGGMAP JOINLINES(-
   'RELATION geography.parentrel PRECOMPUTE (geography.lvldim 2 4)' -
   'RELATION product.parentrel' -
   'RELATION channel.parentrel' -
   'RELATION time.parentrel' -
   'END')
END
```

Example 6–18 Creating an Aggmap Using an Input File

Suppose that you have created a disk file called `salesagg.txt`, which contains the following aggmap definition and specification.

```
DEFINE sales.agg AGGMAP <time, product, geography>
AGGMAP
RELATION time.r PRECOMPUTE (time NE 'Year99')
RELATION product.r PRECOMPUTE (product NE 'ALL')
RELATION geography.r
CACHE STORE
END
```

To include the `sales.agg` aggmap in your analytic workspace, execute the following statement, where `inf` is the alias for the directory where the file is stored.

```
INFILE 'inf/salesagg.txt'
```

The `sales.agg` aggmap has now been defined and contains the three RELATION statements and the CACHE statement. In this example, you are specifying that all of the data for the hierarchy for the `time` dimension, `time.r`, should be aggregated, except for any data that has a `time` dimension value of `Year99`. All of the data for the hierarchy for the `product` dimension, `product.r`, should be aggregated, except for any data that has a `product` dimension value of `All`. All `geography` dimension values are aggregated. The CACHE STORE statement specifies that any data that are rolled up on the fly should be calculated just once and stored in the cache for other access requests during the same session.

You can now use the `sales.agg` aggmap with an AGGREGATE command, such as.

```
AGGREGATE sales USING sales.agg
```

In this example, any data value that dimensioned by a `Year99` value of the `time` dimension or an `All` value of the `product` dimension is calculated on the fly. All other data is aggregated and stored in the analytic workspace.

Example 6–19 Using Multiple Aggmaps

When you use a forecast, you must make sure that all of the input data that is required by that forecast has been pre-calculated. Otherwise, the forecast uses incorrect or nonexistent data. For example, suppose your forecast requires that all line items are aggregated. Using a `budget` variable that is dimensioned by `time`, `line`, and `division`, one approach would be to perform a complete aggregation of the `line` dimension, forecast the `dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR`, and then aggregate the remaining dimension, `division`.

You can support this processing by defining three aggmap objects:

1. Define the first aggmap, named `forecast.agg1`, which aggregates the data needed by the forecast. It contains the following statement.

   ```
   RELATION line.parentrel
   ```

2. Define the second aggmap, named `forecast.agg2`, which aggregates the data generated using the first aggmap and the forecast. It contains the following statement.

   ```
   RELATION division.parentrel PRECOMPUTE ('L3')
   ```

3. Define the third aggmap, named `forecast.agg3`, which contains the RELATION statements in the specifications of the first two aggmaps.

   ```
   RELATION line.parentrel
   RELATION division.parentrel PRECOMPUTE ('L3')
   ```

When your forecast is in a program named `fore.prg`, then you would use the followinfstatements to aggregate the data.

```
AGGREGATE budget USING forecast.agg1   "Aggregate over LINE
CALL fore.prg                          "Forecast over TIME
AGGREGATE budget USING forecast.agg2   "Aggregate over DIVISION
"Compile the limit map for LINE and DIVISION
COMPILE forecast.agg3
"Use the combined aggmap for the AGGREGATE function
CONSIDER budget
PROPERTY 'NATRIGGER' 'AGGREGATE(budget USING forecast.agg3)'
```

Example 6–20   Using an AGGINDEX Statement in an Aggregation Specification

Suppose you have two variables, `sales1` and `sales2`, with the following definitions.

```
DEFINE sales1 DECIMAL <time, SPARSE<product, channel, customer>>
DEFINE sales2 DECIMAL <time, SPARSE<product, channel, customer>>
```

You do not want to precompute and commit all of the `sales` data to the database, because disk space is limited and you need to improve performance. Therefore, you need to create an aggmap, in which you specify which data should be pre-computed and which data should be calculated on the fly.

You define the aggmap, named `sales.agg`, with the following statement.

```
DEFINE sales.agg AGGMAP <time, SPARSE<product, channel, customer>>
```

Next, you use the AGGMAP command to enter the following specification for `sales.agg`.

```
RELATION time.r PRECOMPUTE (time NE 'Year99')
RELATION product.r PRECOMPUTE (product NE 'All')
RELATION channel.r
RELATION customer.r
AGGINDEX NO
```

This aggregation specification tells Oracle OLAP that all `sales` data should be rolled up committed to the database with the exception of any data that has a `time` dimension value of `Year99` or a `product` dimension value of `All`—the data for those cells is calculated the first time a user accesses them. The AGGINDEX value of `NO` tells Oracle OLAP not to create the indexes for data that should be calculated on the fly.

Now you execute the following statement.

```
sales2 = AGGREGATE(sales1 USING sales.agg) ACROSS SPARSE -
   <product, channel, customer>
```

`sales2` now contains all of the data in `sales1`, plus any data that is aggregated for `Year99`—this is because `time` is not included in a composite.

On the other hand, the data that is aggregated for the `product` value of `All` is not computed and stored in `sales2`. This is because the `product` dimension is included in a composite—the indexes that are required for dimensions that are included in composites were not created because the aggregation specification contains an `AGGINDEX NO` statement. Since the indexes did not exist, Oracle OLAP never called the AGGREGATE function to compute the data to be calculated on the fly.

Example 6–21 Aggregating By Dimension Attributes

Assume that when your business makes a sales it keeps records of the customer's name, sex, age, and the amount of the sale. To hold this data, your analytic workspace contains a dimension named `customer` and three variables (named `customer_sex`, `customer_age`, and `sales`) that are dimensioned by `customer`.

```
 REPORT W 14 <customer_sex customer_age sales>

CUSTOMER        CUSTOMER_SEX   CUSTOMER_AGE      SALES
-------------- -------------- -------------- --------------
Clarke         M                          26      26,000.00
Smith          M                          47      15,000.00
Ilsa           F                          24      33,000.00
Rick           M                          33      22,000.00
```

You want to aggregate the detail sales data over sex and age to calculate the amount of sales you have made to males and females, and the amount of sales for different age ranges. To hold this data you will need an INTEGER variable that is dimensioned by hierarchical dimensions for sex and age. You will also need an aggmap object that specifies the calculations that Oracle OLAP will perform to populate this variable from the data in the `sales` variable.

To create and populate the necessary objects, you take the following steps:

1. Create and populate dimensions and self-relations for hierarchical dimensions named `sex` and `age`.

```
DEFINE sex DIMENSION TEXT
DEFINE sex.parentrel RELATION sex <sex>
DEFINE age DIMENSION TEXT
DEFINE age.parentrel RELATION age <age>

AGE                AGE.PARENTREL
-------------- --------------------
0-20           All
21-30          All
31-50          All
51-100         All
No Response    All
All            NA

SEX                SEX.PARENTREL
-------------- --------------------
M              All
F              All
No Reponse     All
All            NA
```

2. Create and populate relations that map the `age` and `sex` dimensions to the `customer` dimension.

```
DEFINE customer.age.rel RELATION age <customer>
DEFINE customer.sex.rel RELATION sex <customer>

CUSTOMER          CUSTOMER.AGE.REL     CUSTOMER.SEX.REL
-------------- -------------------- --------------------
Clarke         21-30                M
Smith          31-50                M
Ilsa           21-30                F
Rick           31-50                M
```

3. Create a variable named `sales_by_sex_age` to hold the aggregated data. Like the `sales` variable this variable is of type DECIMAL, but it is dimensioned by `sex` and `age` rather than by `customer`.

```
DEFINE sales_by_sex_age VARIABLE DECIMAL <sex age>
```

4. Define an AGGMAP type aggmap object named `ssa_aggmap` to calculate the values of the `sales_by_sex_age` variable.

   ```
   DEFINE SSA_AGGMAP AGGMAP
   AGGMAP
   RELATION sex.parentrel OPERATOR SUM
   RELATION age.parentrel OPERATOR SUM
   BREAKOUT DIMENSION customer -
   BY customer.sex.rel, customer.age.rel OPERATOR SUM
   END
   ```

   Notice that the specification for the `ssa_aggmap` includes the following statements:

   - A BREAKOUT DIMENSION statement that specifies how to map the `customer` dimension of the `sales` variable to the lowest-level values of the `sales_by_sex_age` variable. This statement specifies the name of the dimension of the variable that contains the detail values (that is, `customer`) and the names of the relations (`customer.sex.rel` and `customer.age.rel`) that define the relations between `customer` dimension and the `sex` and `age` dimensions.
   - Two RELATION statements that specify how to aggregate up the `sex` and `age` dimensions of the `sales_by_sex_age` variable. Each of these statements includes the name of the child-parent relation (`sex.parentrel` or `age.parentrel`) that define the self-relation for the hierarchal dimension (`sex` or `age`).

5. Populate the `sales_by_sex_age` variable by issuing and AGGREGATE command that specifies that the detail data for the aggregation comes from the `sales` variable.

   ```
   AGGREGATE sales_by_sex_age USING ssa_aggmap FROM sales
   ```

After performing the aggregation, a report of `sales_by_sex_age` shows the calculated values.

```
               ---------------------SALES_BY_SEX_AGE----------------------
               ----------------------------SEX----------------------------
AGE                 M              F          No Reponse        All
-------------- -------------- -------------- -------------- --------------
0-20                       NA             NA             NA             NA
21-30               26,000.00      33,000.00             NA      59,000.00
31-50               37,000.00             NA             NA      37,000.00
51-100                     NA             NA             NA             NA
No Response                NA             NA             NA             NA
All                 63,000.00      33,000.00             NA      96,000.00
```

Example 6–22 Using a CACHE Statement in an Aggregation Specification

Suppose you have a `sales` variable with the following definition.

```
DEFINE sales DECIMAL <time, SPARSE<product, channel, customer>>
```

You do not want to pre-compute and commit all of the `sales` data, because space is limited and you need to improve performance. Therefore, you need to create an aggmap, in which you will specify which data should be pre-computed and which data should be calculated on the fly.

You define the aggmap, named `sales.agg`, with the following statement.

```
DEFINE sales.agg AGGMAP <time, SPARSE<product, channel, - customer>>
```

Next, you use the AGGMAP statement to enter the following aggregation specification for`sales.agg`.

```
AGGMAP
RELATION time.r PRECOMPUTE (time NE 'YEAR99')
RELATION product.r PRECOMPUTE (product NE 'ALL')
RELATION channel.r
RELATION customer.r
CACHE SESSION
END
```

This aggregation specification tells Oracle OLAP that all `sales` data should be rolled up and committed, with the exception of any cells that have a time dimension value of `Year99` or a product dimension value of `ALL`; the data for those cells will be calculated the first time a user accesses them. Because the CACHE statement uses the SESSION keyword, that means that when those cells are calculated on the fly, the data is stored in the cache for the remainder of the Oracle OLAP session.

That way, the next time a user accesses the same cell, the data will not have to be calculated again. Instead, the data will be retrieved from the session cache.

Example 6–23 Populating All Levels of a Hierarchy Except the Detail Level

Assume that your analytic workspace contains the relations and dimensions with the following definitions.

```
DEFINE geog.d TEXT DIMENSION
DEFINE geog.r RELATION geog.d <geog.d>
DEFINE sales_by_units   INTEGER VARIABLE <geog.d>
DEFINE sales_by_revenue DECIMAL VARIABLE <geog.d>
DEFINE price_per_unit   DECIMAL VARIABLE <geog.d>
```

Assume that you create two aggmap objects. One aggmap object, named `units_aggmap`, is the specification to aggregate data in the `sales_by_units` variable. The other aggmap object, `revenue_aggmap`, is the specification to calculate all of the data *except* the detail data in the `sales_by_revenue` variable.

```
DEFINE units_aggmap AGGMAP
AGGMAP
  RELATION geog.r OPERATOR SUM
END

DEFINE revenue_aggmap AGGMAP
AGGMAP
  RELATION geog.r OPERATOR WSUM ARGS WEIGHTBY price_per_unit
  CACHE NOLEAF
END
```

The following steps outline the aggregation process:

1. Before either the `sales_by_unit` or `sales_by_revenue` variables are aggregated, they have the following values.

   ```
   GEOG.D    SALES_BY_UNIT SALES_BY_REVENUE
   --------- ------------- ----------------
   Boston                1               NA
   Medford               2               NA
   San Diego             3               NA
   Sunnydale             4               NA
   MA                    NA              NA
   CA                    NA              NA
   USA                   NA              NA
   ```

2. After the data for the `sales_by_unit` variable is aggregated, the `sales_by_unit` and `sales_by_revenue` variables have the following values.

```
AGGREGATE sales_by_unit    USING units_aggmap

GEOG.D    SALES_BY_UNIT SALES_BY_REVENUE
--------- ------------- ----------------
Boston                1               NA
Medford               2               NA
San Diego             3               NA
Sunnydale             4               NA
MA                    3               NA
CA                    7               NA
USA                  10               NA
```

3. After the data for the `sales_by_revue` variable is aggregated, the `sales_by_unit` and `sales_by_revenue` variables have the following values.

```
AGGREGATE sales_by_revenue USING revenue_aggmap FROM units_aggmap

GEOG.D    SALES_BY_UNIT SALES_BY_REVENUE
--------- ------------- ----------------
Boston                1               NA
Medford               2               NA
San Diego             3               NA
Sunnydale             4               NA
MA                    3             13.5
CA                    7             31.5
USA                  10             45.0
```

Example 6–24   Aggregating into a Different Variable

Assume that there is a variable named `sales` that is dimensioned by `time`, a hierarchical dimension, and `district`, a non-hierarchical dimension.

```
DEFINE time DIMENSION TEXT
DEFINE time.parentrel RELATION time <time>
DEFINE district DIMENSION TEXT
DEFINE sales VARIABLE DECIMAL <time district>
```

```
             -----------------------SALES-----------------------
             ---------------------DISTRICT----------------------
TIME            North        South         West         East
------------ ------------ ------------ ------------ ------------
1976Q1         168,776.81   362,367.87   219,667.47   149,815.65
1976Q2         330,062.49   293,392.29   237,128.26   167,808.03
1976Q3         304,953.04   354,240.51   170,892.80   298,737.70
1976Q4         252,757.33   206,189.01   139,954.56   175,063.51
1976                   NA           NA           NA           NA
```

Assume also that you want to calculate the total sales for each quarter and year for all districts *except* the North district. To perform this calculation using an aggmap object, you take the following steps:

1. Define a valueset named not_north that represents the values of district for which you want to aggregate data.

   ```
   DEFINE not_north VALUESET district
   LIMIT not_north TO ALL
   LIMIT not_north REMOVE 'North'
   ```

2. Define a variable named total_sales_exclud_north to hold the results of the calculation.

   ```
   DEFINE total_sales_exclud_north VARIABLE DECIMAL <time>
   ```

   Notice that, like sales, the total_sales_exclud_north variable is dimensioned by time. However, unlike sales, the total_sales_exclud_north variable is not dimensioned by district since it will hold detail data for each district, but only the total (aggregated) values for the South, West, and East districts (that is, all districts *except* North).

3. Define an aggmap object that specifies the calculation that you want performed.

   ```
   DEFINE agg_sales_exclud_north AGGMAP
   AGGMAP
   RELATION time.parentrel OPERATOR SUM
   DROP DIMENSION district OPERATOR SUM VALUES not_north
   END
   ```

   Notice that the aggregation specification consists of two statements that specify how to perform the aggregation:

- A RELATION statement that specifies how to aggregate up the hierarchical `time` dimension
- A DROP DIMENSION statement that specifies how to aggregate across the non-hierarchical `district` dimension. In this case, the DROP DIMENSION also uses the `not_north` valueset to specify that values for the `North` district are excluded when performing the aggregation

4. Aggregate the data.

```
AGGREGATE total_sales_exclud_north USING agg_sales_exclud_north FROM sales
```

The report of the `total_sales_exclud_north` variable shows the aggregated values.

```
TIME             ALL_SALES_EXCEPT_NORTH
------------ ------------------------------
1976Q1                           731,850.99
1976Q2                           698,328.58
1976Q3                           823,871.02
1976Q4                           521,207.09
1976                           2,775,257.69
```

Example 6–25 Using a MEASUREDIM Statement in an Aggregation Specification

Suppose you have defined a measure dimension named `measure`. You then define an operation variable named `myopvar`, which is dimensioned by `measure`. When you use `myopvar` in an aggregation specification, you must also include a MEASUREDIM statement that identifies `measure` as the dimension is included in the definition of `myopvar`.

The MEASUREDIM statement should follow the last RELATION (for aggregation) statement in the aggregation specification, as shown in the following example.

```
DEFINE sales.agg AGGMAP <time, product, geography>
AGGMAP
RELATION time.r OPERATOR myopvar
RELATION product.r
RELATION geography.r
MEASUREDIM measure
END
```

Example 6–26 Solving a Model in an Aggregation

This example uses the `budget` variable.

```
DEFINE budget VARIABLE DECIMAL <line time>
LD Budgeted $ Financial
```

The `time` dimension has two hierarchies (`Standard` and `YTD`) and a parent relation named `time.parentrel` as follows.

```
               -----TIME.PARENTREL------
               ----TIME.HIERARCHIES-----
TIME             Standard       YTD
-------------- ------------ ------------
Last.YTD       NA           NA
Current.YTD    NA           NA
Jan01          Q1.01        Last.YTD
...
Dec01          Q4.01        Last.YTD
Jan02          Q1.02        Current.YTD
Feb02          Q1.02        Current.YTD
Mar02          Q1.02        Current.YTD
Apr02          Q2.02        Current.YTD
May02          Q2.02        Current.YTD
Q1.01          2001         NA
...
Q4.01          2001         NA
Q1.02          2002         NA
Q2.02          2002         NA
2001           NA           NA
2002           NA           NA
```

The relationships among line items are defined in the following model.

```
DEFINE income.budget MODEL
MODEL
DIMENSION line time
opr.income = gross.margin - marketing
gross.margin = revenue - cogs
revenue = LAG(revenue, 12, time) * 1.02
cogs = LAG(cogs, 1, time) * 1.01
marketing = LAG(opr.income, 1, time) * 0.20
END
```

The following aggregation specification pre-aggregates all of the data. Notice that all of the data must be pre-aggregated because the model includes both LAG functions and a simultaneous equation.

```
DEFINE budget.aggmap1 AGGMAP
AGGMAP
MODEL income.budget
RELATION time.parentrel
END
```

Example 6–27 Aggregating Up a Hierarchy

Suppose you define a `sales` variable with the following statement.

```
DEFINE sales VARIABLE <time, SPARSE <product, geography>>
```

The aggregation specification for `sales` might include RELATION statements like the following.

```
AGGMAP
RELATION time.r PRECOMPUTE ('Yr98', 'Yr99')
RELATION product.r
RELATION geography.r PRECOMPUTE (geography NE 'Atlanta')
END
```

The AGGREGATE command will aggregate values for `Yr98` and `Yr99`, over all of products, and over all geographic areas except for `Atlanta`. All other aggregates are calculated on the fly.

Example 6–28 Using Valuesets

Suppose you have a hierarchy dimension named time.type, whose dimension values are `Fiscal` and `Calendar`, in that order. These hierarchies are in conflict, and you want to precompute some `time` data but calculate the rest on the fly. Because the `Calendar` hierarchy is the last dimension value in the hierarchy dimension, this means that you need to define a valueset in order to get the correct results for the `Fiscal` hierarchy.

First, use the following statements to define and populate a valueset.

```
DEFINE time.vs VALUESET time
LIMIT time.vs TO 'Calendar' 'Fiscal'
```

You can then use the valueset in the following RELATION statement. Because the Fiscal hierarchy is the last hierarchy in the valueset, the data that is aggregated will be accurate for the Fiscal hierarchy.

```
RELATION time.r(time.vs) PRECOMPUTE ('Yr99', 'Yr00')
```

Example 6–29 Aggregating with a RELATION Statement That Uses an ARGS Keyword

You can list the arguments in a RELATIION statement directly in the statement or as the value of a text variable. For example, the following statement specifies `WEIGHTBY wobj` as an argument.

```
RELATION time.r OPERATOR wsum ARGS WEIGHTBY wobj
```

Alternatively, you can define an variable for the argument whose value is the text of the WEIGHTBY clause.

```
DEFINE argvar TEXT
argvar = 'WEIGHTBY wobj'
```

Then the RELATION statement can specify the text variable that contains the WEIGHTBY clause.

```
RELATION time.r OPERATOR WSUM ARGS argvar
```

Example 6–30 Aggregating Using a Measure Dimension

Suppose you want to use a single AGGREGATE command to aggregate the `sales`, `units`, `price`, and `inventory` variables. When you want to use the same operator for each variable, then you do not need to use a measure dimension. However, when you want to specify different aggregation operations, then you need to use a measure dimension.

The following statement defines a dimension named `measure`.

```
DEFINE measure DIMENSION TEXT
```

You can then use a MAINTAIN statement to add dimension values to the `measure` dimension.

```
MAINTAIN measure ADD 'sales', 'units', 'quota', 'inventory'
```

Use the `measure` dimension to dimension a text variable named `meas.opvar` that you will use as the operator variable.

```
DEFINE meas.opvar TEXT WIDTH 2 <measure>
```

The following statements add values to OPVAR

```
meas.opvar (measure 'sales') = 'SU'
meas.opvar (measure 'units') = 'SU'
meas.opvar (measure 'price') = 'HA'
meas.opvar (measure 'inventory') = 'HL'
```

The aggregation specification might look like the following. Note that when you specify an operator variable in a RELATION statement, you must include a MEASUREDIM statement that specifies the name of the measure dimension (`measure` in the following example) in the aggregation specification.

```
DEFINE opvar.aggmap AGGMAP
AGGMAP
RELATION geography.parentrel PRECOMPUTE (geography.lvldim 2 4)
RELATION product.parentrel OPERATOR opvar
RELATION channel.parentrel OPERATOR opvar
RELATION time.parentrel OPERATOR opvar
MEASUREDIM measure
END
```

Example 6–31 Aggregating Using a Line Item Dimension

Suppose you have two variables, `actual` and `budget`, that have these dimensions.

```
<time line division>
```

You want to use different methods to calculate different line items. You create a text variable that you will use as the operator variable.

```
DEFINE line.opvar TEXT WIDTH 2 <line>
```

You then populate `line.opvar` with the appropriate operator for each line item, for example:

```
line.opvar (line 'Net.Income') = 'SU'
line.opvar (line 'Tax.Rate') = 'AV'
```

The aggregation specification might look like this.

```
DEFINE LINE.AGGMAP AGGMAP
AGGMAP
RELATION time.parentrel OPERATOR line.opvar
RELATION division.parentrel
END
```

Example 6–32 Skip-Level Aggregation

Suppose you want to aggregate `sales` data. The `sales` variable is dimensioned by `geography`, `product`, `channel`, and `time`.

First, consider the hierarchy for each dimension. How many levels does each hierarchy have? What levels of data do users typically query? When you are designing a new workspace, what levels of data do your users plan to query?

Suppose you learn the information described in the following table about how users tend to query `sales` data for the `time` hierarchy.

Time Level Names	Descriptive Level Name	Examples of Dimension Values	Do users query this level often?
L1	Year	`Year99,Year00`	yes
L2	Quarter	`Q3.99,Q3.99,Q1.00`	yes
L3	Month	`Jan99,Dec00`	yes

While the next table shows how your users tend to query `sales` data for the `geography` hierarchy.

Geography Level Names	Descriptive Level Name	Examples of Dimension Values	Do users query this level often?
L1	World	`World`	yes
L2	Continent	`Europe, Americas`	no
L3	Country	`Hungary, Spain`	yes
L4	City	Budapest, Madrid	yes

Finally, the next table shows how your users tend to query `sales` data for the `product` dimension hierarchy.

Product Level Names	Descriptive Level Name	Examples of Dimension Values	Do users query this level often?
L1	All Products	`Totalprod`	yes
L2	Division	`Audiodiv, Videodiv`	yes
L3	Category	`TV, VCR`	yes
L4	Product	Tuner, CDplayer	yes

Using this information about how users query data, you should use the following strategy for aggregation:

- Fully aggregate `time` and `product` because all levels are queried frequently.
- For the `geography` dimension, aggregate data for `L1` (`World`) and `L3` (`Country`) because they are queried frequently. However, `L2` is queried less often and so can be calculated on the fly.

The lowest level of data was loaded into the analytic workspace. The aggregate data is calculated from this source data.

Therefore, the aggregation specification might look like the following.

```
RELATION time.parentrel
RELATION geography.parentrel PRECOMPUTE (geog.leveldim 'L3' 'L1')
RELATION product.parentrel
```

Example 6–33 Aggregation Specification with RELATION Statements That Include PRECOMPUTE Clauses

This aggregation specification uses PRECOMPUTE clauses in the RELATION statements to limit the data that is aggregated by the AGGREGATE command.

```
DEFINE gpct.aggmap AGGMAP
LD Aggmap for sales, units, quota, costs
AGGMAP
RELATION geography.parentrel PRECOMPUTE (geography.levelrel 'L3')
RELATION product.parentrel PRECOMPUTE (LIMIT(product complement 'TotalProd'))
RELATION channel.parentrel
RELATION time.parentrel PRECOMPUTE (time NE '2001')
END
```

AGGINDEX

Within an aggregation specification, an AGGINDEX statement in an aggregation specification tells Oracle OLAP whether the compilation of that aggmap should create indexes (meaning, composite tuples) for data cells that are calculated on the fly by the AGGREGATE function. Therefore, the AGGINDEX statement has an effect on a dimension that is included in a composite but it has no effect on a dimension that is not included in a composite.

These indexes are used in the MODEL (in an aggregation) statement and in statements that use the ACROSS phrase to help Oracle OLAP loop over variables that are dimensioned by composites. These statements expect all data to be calculated. When you specify calculating some data on the fly, that data appears to be missing. When you set AGGINDEX to `YES`, then the statements try to access the missing data whether or not you are using the AGGREGATE function to perform calculation on the fly (meaning, you have added to the variable whose data is being aggregated an `NA` trigger property that calls the AGGREGATE function).

When the indexes have been created and you use AGGREGATION with the AGGREGATE function, then when MODEL (or a statement that uses the ACROSS phrase) requests the missing data, that data is calculated on the fly. That means that the results of the MODEL (or other statement) are correct, because the statement has all of the data that it needs.

When these indexes have not been created, then the missing data cannot be calculated. As a result, the statements that need the indexes interpret the missing data as `NA` data, even when you use the AGGREGATE function.

Syntax

AGGINDEX {<u>YES</u>|NO}

Arguments

YES
Tells the AGGMAP compiler to make sure that all possible indexes are created whenever an aggmap is recompiled. In other words, indexes are created both for the data that is being pre-calculated and the data that is calculated on the fly. This happens when you use a COMPILE statement to compile the aggmap, as well as when the AGGREGATE command automatically compiles an aggmap whose specification has changed since the last time it was compiled. The creation of all

possible indexes results in a longer compilation time but faster execution of the AGGREGATE function. (Default)

NO
Does not create the indexes for data that is calculated on the fly. Omitting the creation of these index values accelerates the compilation time, but causes Oracle OLAP to treat the uncomputed data as `NA` data whenever the MODEL (in an aggregation) statement or the ACROSS phrase is used.

Notes

When You Should Use an AGGINDEX Value of YES

The primary advantage to using an AGGINDEX value of `YES` is that then Oracle OLAP always try to access data that you have specified to be calculated on the fly. When you have created an $NATRIGGER property for a variable that calls the AGGREGATE function, the variable appears to have been fully precomputed. That means that when any `NA` value is encountered, the `NA` trigger is called during a MODEL (in an aggregation) statement or a statement using the ACROSS phrase. When the `NA` trigger is called, the AGGREGATE function is executed, and the data is calculated on the fly.

When AGGINDEX has a value of `NO`, then the `NA` trigger is called only to aggregate data for dimensions that are not included in a composite. Data for dimensions that are included in composites is interpreted as `NA` values.

For example, suppose you have two variables called `sales1` and `sales2`, which are defined with the following definitions.

```
DEFINE sales1 DECIMAL <time, SPARSE <product, geography>>
DEFINE sales2 DECIMAL <time, SPARSE <product, geography>>
```

Now suppose you have an aggmap object named `sales.agg`, which has the following definition.

```
DEFINE sales.agg AGGMAP <time, SPARSE <product, geography>>
```

When you add a specification to the `sales.agg` aggmap, you enter RELATION (for aggregation) statements for `time`, `product` and `geography` with `PRECOMPUTE` clauses that specify `NA`. This specifies that no data is aggregated—instead, all of the data for any variable that uses this aggmap is calculated on the fly.

```
RELATION time.r PRECOMPUTE (NA)
RELATION product.r PRECOMPUTE (NA)
RELATION geography.r PRECOMPUTE (NA)
```

Now attach the following $NATRIGGER property to the `sales1` variable.

```
CONSIDER sales1
PROPERTY '$NATRIGGER' 'AGGREGATE(sales1 USING sales.agg)'
```

Consider the effect of AGGINDEX in the following statement. Because you did not enter an AGGINDEX statement in the `sales.agg` aggregation specification, the default of AGGINDEX `YES` is assumed.

```
sales2 = sales1 ACROSS SPARSE <product, geography>
```

This statement loops over the data in `sales1` and copies the values into `sales2`. This statement causes the `NA` trigger to call the AGGREGATE function for all of the data that you have specified to be calculated on the fly in `sales1`. This means that after the aggregation that `sales2` contains a copy of `sales1` plus all the aggregate data cells (the cells that would have been calculated if the `sales1` data had been completely precomputed, meaning, fully rolled up).

However, when you put an AGGINDEX `NO` statement in the `sales.agg` aggregation specification, then `sales2` contains a copy of the data in `sales1` and the aggregate data cells for the `time` dimension.

Note that in both cases, $NATRIGGER is called to aggregate `time` data, because the `time` dimension is not included in the composite, so the value of AGGINDEX has no effect on it.

When You Should Use an AGGINDEX Value of NO

You can use an AGGINDEX value of `NO` when you know that either of the following is true:

- Your application does not use a MODEL (in an aggregation) statement or an ACROSS phrase.
- The results of your MODEL (in an aggregation) statement or ACROSS phrase are additive, and data that needs to be aggregated can be calculated safely on the fly.

Each of the preceding cases ensures that the data that you have specified to be calculated on the fly is available at the appropriate time.

By setting AGGINDEX to NO, the size of the indexes is reduced, and overall application performance improves.

When Using an AGGINDEX Value Of NO Causes Problems

When you run a MODEL that assumes all data that should be aggregated has been aggregated, then you may get NA data where real data should occur. For instance, suppose you have a variable that has a composite that includes the time dimension. You perform a calculation that subtracts the fourth quarter from the total for the year. When the value of Year is to be calculated dynamically, and the AGGINDEX statement is set to NO, then the result of the calculation is NA. When the value of Year was precomputed or when AGGINDEX is set to YES, then the MODEL correctly calculates a result equal to the sum of the first three quarters.

Index Creation Is Based on Existing Data

Only the indexes that are needed to aggregate existing data are created when AGGINDEX has a value of YES. For example, suppose one of the dimensions in your composite is a dimension named time. The lowest-level data for the time dimension is at the monthly level. Therefore, the dimension values that are associated with the lowest-level data are Jan99, Feb99, and so on. The monthly data aggregates to quarters and to years. Suppose you have data for the first six months of the year. When AGGINDEX has a value of YES, indexes are created for the Q1, Q2, and Yr99 dimension values, but not for Q3 and Q4.

Reducing Compilation Time When AGGINDEX is YES

One disadvantage of using the default of AGGINDEX YES is that the compilation of the aggmap takes a longer time to complete. You can eliminate the cost of this extra time by using the FUNCDATA keyword with the AGGREGATE command. When you use the FUNCDATA keyword, all possible indexes (regardless of how you have limited your data) are created. However, do not use the FUNCDATA keyword when you use a different aggmap to execute the AGGREGATE command and the AGGREGATE function.

Examples

For an example of using an AGGINDEX statement, see Example 6–20, "Using an AGGINDEX Statement in an Aggregation Specification" on page 6-48.

BREAKOUT DIMENSION

Within an aggregation specification, a BREAKOUT DIMENSION statement specifies how a dimension of the target variable maps to one or more dimensions of the source variable. You use this statement in an aggregation specification when you will be aggregating the detail data from one variable (the source variable) into another variable (the target variable) that has a different dimension (that is, a "breakout" dimension) than the variable that contains the detail data.

Syntax

BREAKOUT DIMENSION *dimname* BY *relationname* [, *relationname*...] -

 OPERATOR *operation* [ARGS *argument*]

where:

argument specifies the settings of various options and is one or more of the following phrases:

```
DIVIDEBYZERO {YES|NO}
DECIMALOVERFLOW {YES|NO}
NASKIP {YES|NO}
WEIGHTBY [WNAFILL {number|NA}] wobj
```

Arguments

dimname
The name of a dimension in the variable that contains the detail data (that is, the source variable).

relationname
The name of a relation whose values relate a dimension of the target variable to *dimname*.

OPERATOR
Identifies the calculation method used to aggregate the data.

operation
A keyword that describes the type of aggregation to perform. The keywords are listed in Table 6–3, " Aggregation Operators" on page 6-83.

ARGS
Indicates optional handling of the aggregation.

DIVIDEBYZERO
Specifies whether to allow division by zero.

YES allows division by zero; a statement involving division by zero executes without error but produces `NA` results.

NO disallows division by zero; a statement involving division by zero stops executing and produces an error message.

The default value is the current value of the DIVIDEBYZERO option.

DECIMALOVERFLOW
Specifies whether to allow decimal overflow, which occurs when the result of a calculation is very large and can no longer be represented by the exponent portion of the numerical representation.

YES allows overflow; a calculation that generates overflow executes without error and produces `NA` results.

NO disallows overflow; a calculation involving overflow stops executing and generates an error message.

The default value is the current value of the DECIMALOVERFLOW option.

NASKIP
Specifies whether `NA` values are input.

YES specifies that `NA` values are ignored when aggregating. Only actual values are used in calculations.

NO specifies that `NA` values are considered when aggregating. When any of the values being considered are `NA`, the calculation returns `NA`.

The default value is the current value of the NASKIP option.

The value that you specify for the NASKIP phrase does *not* effect calculation performed when you specify HAVERAGE, HFIRST, HLAST, HWAVERAGE, HWFIRST, HWLAST for *operation*.

WEIGHTBY
Indicates that weighted aggregation is to be performed. You must include a WEIGHTBY clause when you specify HWAVERAGE, HWFIRST, HWLAST, SSUM, WAVERAGE, WFIRST, WLAST, or WSUM for *operation*. The WEIGHTBY phrase always includes a *wobj* argument and can optionally include the WNAFILL

keyword. For more information about the use of the WEIGHTBY phrase, see RELATION (for aggregation).

WNAFILL
Indicates handling for `NA` values. The default values for WNAFILL vary depending on the value of operation.

number
Substitutes a number for every `NA` value. That number will replace every `NA` value in the weight object, weight formula, or weight relation.

- `0.0` is the default for HWAVERAGE and SSUM.
- `1.0` is the default for HWFIRST, HWLAST, WAVERAGE, WFIRST, WLAST, and WSUM.

NA
Specifies that `NA` values are to be specified as `NA`. `NA` is the default for OR.

For more information about using the WNAFILL phrase, see RELATION (for aggregation).

wobj
A variable, formula, or relation that provides the weighted values. It can be numeric or BOOLEAN. When *wobj* is BOOLEAN, then `TRUE` has a weight of `1.0` and `FALSE` has a weight of `0.0`. A formula is queried only when needed, depending on the dimensionality of the formula and the of the variable being aggregated. When *wobj* is a relation, it should be a one-dimensional self-relation. For more information about specifying values for *wobj*, see RELATION (for aggregation).

Examples

For an example of using the BREAKOUT DIMENSION statement, see Example 6–21, "Aggregating By Dimension Attributes" on page 6-49.

CACHE

Within an aggregation specification, a CACHE statement tells Oracle OLAP whether to cache or store the calculated data, whether to populate leaf or detail data when the variable data is aggregated using detail data from another variable, and whether to cache `NA` values when a summary values calculates to `NA`.

Note: The CACHE statement is only one factor that determines whether variable data that has been aggregated on-the-fly using the AGGREGATE function is stored or cached. See "How Oracle OLAP Determines Whether to Store or Cache Aggregated Data" on page 6-23.

Syntax

CACHE {NOSTORE|NONE|STORE|SESSION|<u>DEFAULT</u>} [LEAF|NOLEAF] [NA|NONA]

Arguments

NONE
NOSTORE
For data that is calculated using the AGGREGATE function, specifies that Oracle OLAP calculates the data each time the AGGREGATE function executes. When you specify either of these keywords, Oracle OLAP does not store or cache the data calculated by the AGGREGATE function.

STORE
For data that is calculated using the AGGREGATE function, specifies that Oracle OLAP stores data calculated by the AGGREGATE function in the variable in the database. When you specify this option, the results of the aggregation are permanently stored in the variable when the analytic workspace is updated and committed.

SESSION
For data that is calculated using the AGGREGATE function, specifies that Oracle OLAP caches data calculated by the AGGREGATE function in the session cache (see "What is an Oracle OLAP Session Cache?" on page 21-54). When you specify this option, the results of the aggregation are ignored during updates and commits and are discarded at the end of the session.

Note: When SESSCACHE is set to `NO`, Oracle OLAP does not cache the data even when you specify `SESSION`. In this case, specifying `SESSION` is the same as specifying `NONE`.

DEFAULT

For data that is calculated using the AGGREGATE function, specifies that you do not want Oracle OLAP to use the CACHE statement when determining what to do with data that is calculated by the AGGREGATE function. See "How Oracle OLAP Determines Whether to Store or Cache Aggregated Data" on page 6-23. (Default)

LEAF

When the variable data is aggregated using detail data from another variable, specifies that Oracle OLAP calculates the leaf data for the variable.

NOLEAF

When the variable data is aggregated using detail data from another variable, specifies that Oracle OLAP does not calculate the leaf data for the variable.

NA

For data that is calculated using the AGGREGATE function, specifies that Oracle OLAP places any `NA` values that are the results of the execution of the AGGREGATE function in the Oracle OLAP session cache. In this case, when there is a variable has an $NATRIGGER property with an AGGREGATE function as its expression, Oracle OLAP does *not* recalculate the values for the variable. (For more information on the caching `NA` values, see "How Oracle OLAP Determines Whether to Store or Cache Results of $NATRIGGER" on page 6-21.)

NONA

For data that is calculated using the AGGREGATE function, specifies that Oracle OLAP does *not* cache any `NA` values that are the results of the execution of the AGGREGATE function. In this case, when a variable has an $NATRIGGER property with an AGGREGATE function as its expression, Oracle OLAP recalculates the values for the variable.

Notes

When to Use NOSTORE

You should use NOSTORE when you know that your users are likely to modify pre-computed data, and you want any data that calculated by the AGGREGATE function to consistent with any of those users' changes.

In other words, suppose a user makes a change to detail-level data, such as `sales` figures for three stores, which are in a `geography` dimension. The `geography` dimension rolls up data from stores to cities to states to regions to countries. In other words, there are five levels in the `geography` dimension's hierarchy. Now suppose that users tend to access data only at the store level (your detail data), the regions level, and the countries level. Those are the levels for which you roll up sales data and commit it to the database. Because users do not access data at the city and state level, you specify that the data cells in those two levels will be calculated on the fly. When users modify the store-level data and then access city data, the city data will be calculated every time that a user requests it. Therefore, any changes that a user makes to the store-level details will accurately rollup to the city and state level every time that user accesses a data cell in the city or state level. (However, this will not be true of the data in the region and country levels, because those cells store pre-computed data.)

When to Use STORE or SESSION

The advantage to using STORE or SESSION is that it improves query performance. For example, suppose your users use a Table tool to look at a variable's data and an individual user requests the same data cells several times in the same session. When you use the default of NOSTORE, then any data that is not aggregated using the AGGREGATE command will have to be calculated every time the user requests that data even if you do not use the FORECALC keyword in the AGGREGATE function. On the other hand, when you use STORE or SESSION, then any given cell of data is calculated only once because it is available in either the variable or the cache for the entire session. Therefore, the next time a user requests that data cell, the data is returned from the variable or the cache instead of being calculated on the fly, which results in faster query time for the user.

Frequently you do not want the data that is calculated using the AGGREGATE function to be stored permanently in the database since that would defeat the purpose of calculating data on the fly.

- To ensure that the aggregated values cannot be permanently committed to the database, use SESSION.

- Use STORE when you know either of the following is true which also ensures that the data that is calculated on the fly using the AGGREGATE function will not be committed to the database:
  - The users of the analytic workspace can only open it as read-only
  - You know that the users of the analytic workspace will not or cannot issue UPDATE and COMMIT commands.

Note: You should use STORE with caution when it is likely that your users modify pre-computed data, and they access data that you have specified to be calculated on the fly using the AGGREGATE function. The problem is that any data that is calculated using the AGGREGATE function before the user's modification will not reflect the user's change unless the user made the change using an AGGREGATE function with the FORCECALC keyword.

Examples

For examples of using a CACHE statement in an aggregation specification, see Example 6–22, "Using a CACHE Statement in an Aggregation Specification" on page 6-52 and Example 6–23, "Populating All Levels of a Hierarchy Except the Detail Level" on page 6-53.

DIMENSION (for aggregation)

Within an aggregation specification, a DIMENSION statement sets the status to a single value of a dimension. When an aggregation specification does not specify such single values with DIMENSION statements, Oracle OLAP uses the current status values of the dimensions when performing the aggregation.

You use a DIMENSION statement to ensure that the status of a dimension is set to the value that you want it to have for the aggregation. You must use a separate DIMENSION statement for each dimension that is not shared by the source, basis, and target objects.

Syntax

DIMENSION *dimension* '*dimval*'

Arguments

dimension
the name of the dimension that you want to limit.

d*imval*
The single value of the dimension to which you want the status of the dimension set for the duration of an aggregation.

DROP DIMENSION

Within an aggregation specification, a DROP DIMENSION statement specifies how non-hierarchical aggregation across variables is performed. You use this statement in aggregation specification when you will be aggregating the detail data from one variable (the source variable) into another variable (the target variable) and you want to aggregate across a non-hierarchical dimension of the source variable. In this case, the target variable has one less dimension (the "dropped" dimension) than the source variable because the values of the source variable associated with this dimension are aggregated to populate the target variable.

Syntax

DROP DIMENSION *dimname* [VALUES {*valsetname*|ALL} OPERATOR *operation* [ARGS *argument*]

where

argument is one or more of the following phrases:

DIVIDEBYZERO {YES|NO}

DECIMALOVERFLOW {YES|NO}

NASKIP {YES|NO}

WEIGHTBY [WNAFILL {*number*|NA}] *wobj*

Arguments

dimname
The name of a dimension in the source variable that contains the detail data.

VALUES
Sets the status of *dimname* during the aggregation.

valsetname
The name of a valueset object that determines the status of the dimension specified by *dimname*.

ALL
Specifies that all of the values of *dimname* are in status.

OPERATOR
Identifies the calculation method used to aggregate the data.

operation
A keyword that describes the type of aggregation to perform. The keywords are listed in Table 6–3, " Aggregation Operators" on page 6-83.

ARGS
Indicates optional handling of the aggregation.

DIVIDEBYZERO
Specifies whether to allow division by zero.

YES allows division by zero; a statement involving division by zero executes without error but produces `NA` results.

NO disallows division by zero; a statement involving division by zero stops executing and produces an error message.

The default value is the current value of the DIVIDEBYZERO option.

DECIMALOVERFLOW
Specifies whether to allow decimal overflow, which occurs when the result of a calculation is very large and can no longer be represented by the exponent portion of the numerical representation.

YES allows overflow; a calculation that generates overflow executes without error and produces `NA` results.

NO disallows overflow; a calculation involving overflow stops executing and generates an error message.

The default value is the current value of the DECIMALOVERFLOW option.

NASKIP
Specifies whether `NA` values are input.

YES specifies that `NA` values are ignored when aggregating. Only actual values are used in calculations.

NO specifies that `NA` values are considered when aggregating. When any of the values being considered are `NA`, the calculation returns `NA`.

The default value is the current value of the NASKIP option.

The value that you specify for the NASKIP phrase does *not* effect calculation performed when you specify HAVERAGE, HFIRST, HLAST, HWAVERAGE, HWFIRST, HWLAST for *operation*.

WEIGHTBY

Indicates that weighted aggregation is to be performed. You must include a WEIGHTBY clause when you specify HWAVERAGE, HWFIRST, HWLAST, SSUM, WAVERAGE, WFIRST, WLAST, or WSUM for *operation*. The WEIGHTBY phrase always includes a *wobj* argument and can optionally include the WNAFILL keyword. For more information about the use of the WEIGHTBY phrase, see RELATION (for aggregation).

WNAFILL

Indicates handling for `NA` values. The default values for WNAFILL vary depending on the value of operation. For more information about using the WNAFILL phrase, see RELATION (for aggregation).

number

Substitutes a number for every `NA` value. That number will replace every `NA` value in the weight object, weight formula, or weight relation.

- `0.0` is the default for HWAVERAGE and SSUM.
- `1.0` is the default for HWFIRST, HWLAST, WAVERAGE, WFIRST, WLAST, and WSUM.

NA

Specifies that `NA` values are to be specified as `NA`. `NA` is the default for OR.

wobj

A variable, formula, or relation that provides the weighted values. It can be numeric or BOOLEAN. When *wobj* is BOOLEAN, then `TRUE` has a weight of `1.0` and `FALSE` has a weight of `0.0`. A formula is queried only when needed, depending on the dimensionality of the formula and the of the variable being aggregated. When *wobj* is a relation, it should be a one-dimensional self-relation. For more information about specifying values for *wobj*,, see RELATION (for aggregation).

Examples

For an example of using a DROPT DIMENSION statement in an aggregation specification, see Example 6–24, "Aggregating into a Different Variable" on page 6-54.

MEASUREDIM (for aggregation)

Within an aggregation specification, a MEASUREDIM statement identifies the name of a measure dimension that is specified in the definition of an operator variable or an argument variable.

Syntax

MEASUREDIM *name*

Arguments

name
The name of the measure dimension. A measure dimension is a dimension that you define. The dimension values are names of existing variables.

Note: You cannot specify a measure dimension when it is included in the definition of the aggmap object.

Notes

Defining a Measure Dimension

The following statement defines a dimension named MEASURE.

```
DEFINE measure DIMENSION TEXT
```

Populating a Measure Dimension

Once you have defined a measure dimension, you can then use a MAINTAIN statement to add dimension values to the MEASURE dimension.

The following statement adds the names of the `sales`, `units`, `price`, and `inventory` variables to `measure` as its dimension values.

```
MAINTAIN measure ADD 'sales', 'units', 'price', 'inventory'
```

Using a Measure Dimension with an Operator Variable

The purpose of using measure dimensions is to take advantage of the flexibility of using non-additive aggregation operators. You can use measure dimensions in the definition of operation variables or argument variables.

The following statements show how to define an operator variable named `opvar` and populate it.

```
DEFINE opvar TEXT <measure>
opvar (measure 'sales') = 'SUM'
opvar (measure 'inventory') = 'HLAST'
```

Examples

For an example of an aggregation specification that includes a MEASUREDIM statement, see Example 6–25, "Using a MEASUREDIM Statement in an Aggregation Specification" on page 6-56.

MODEL (in an aggregation)

Within an aggregation specification, a MODEL statement executes a predefined model.

Syntax

MODEL *modelname* [PRECOMPUTE ALL|NA]

Arguments

modelname
A text expression that contains the name of a predefined MODEL object.

PRECOMPUTE ALL
Specifies that the AGGREGATE command will execute the model as a data maintenance step. The following conditions must be met:

- Any RELATION (for aggregation) or MODEL statements that precede it in the aggregation specification must also be specified as PRECOMPUTE ALL.
- Any RELATION (for aggregation) or MODEL statements that follow it in the aggregation specification can either be specified as PRECOMPUTE ALL or PRECOMPUTE NA.

PRECOMPUTE NA
Specifies that the AGGREGATE function will execute the model at runtime. The following conditions must be met for runtime execution of the model:

- All RELATION (for aggregation) statements in the aggregation specification must appear before the MODEL statements specified as PRECOMPUTE NA.
- Any additional MODEL statements that follow it in the aggregation specification must also be specified as PRECOMPUTE NA.

Notes

Adding a Model to an An Aggregation Specification

You add a model to an aggregation specification using an AGGMAP ADD statement.

Examples

For an example of using a model in an aggregation specification, see Example 6–26, "Solving a Model in an Aggregation" on page 6-57.

RELATION (for aggregation)

Within an aggregation specification, a RELATION statement specifies how data is aggregated across a hierarchical dimension. Frequently, an aggregation specification contains one RELATION statement for each of the hierarchical dimensions of a variable.

Syntax

RELATION *rel-name* [(*valueset...*)] -

  [PRECOMPUTE (*dimension-values* | *valueset2* | ALL | {NA | NONE}] -

  [OPERATOR {*operation*|*opvar*}] -

  [PARENTALIAS *dimension-alias-name*] -

  [ARGS {*argument*|*argsvar*}]

where

argument is one of the following phrases:

  DIVIDEBYZERO {YES|NO}

  DECIMALOVERFLOW {YES|NO}

  NASKIP {YES|NO}

  WEIGHTBY [WNAFILL {*number*|NA}] *wobj*

argsvar is a text variable that contains *argument* phrases for some or all dimension values.

Arguments

rel-name
A relation that defines a hierarchy by identifying the parent of every dimension value in a hierarchy.

valueset
Sets the status of one or more dimensions for the duration of the aggregation. It overrides the current status.

PRECOMPUTE
Indicates dimension values for which data should be precalculated with the AGGREGATE command.

dimension-values
A list of one or more values of dimension.

valueset2
The name of a valueset object. When you include this argument, only data that is dimensioned by the dimension values in the valueset should be precalculated with the AGGREGATE command. The rest of the values can be calculated on the fly.

Note that the current status of a dimension can also limit the data that is precalculated. See the AGGREGATE command for details.

ALL
Specifies that data should be precalculated for all dimension values.

NA
NONE
Specifies that all values should be calculated on the fly (that is, that no data should be precalculated with the AGGREGATE command).

OPERATOR
Identifies the calculation method used to aggregate the data.

operation
A keyword that describes the type of aggregation to perform. The keywords are listed in Table 6–3, " Aggregation Operators". You can specify a fixed-length three-character abbreviation for the keywords by specifying only the first three characters.

Table 6–3 Aggregation Operators

Keyword	Description
SUM	Adds data values. (Default)
SSUM	(Scaled Sum) Adds the value of a weight object to each data value, then adds the data values. When you use this keyword, you must include the WEIGHTBY argument keyword with a variable, formula, or relation as the weight object.

Table 6–3 (Cont.) Aggregation Operators

Keyword	**Description**
WSUM	(Weighted Sum) Multiplies each data value by a weight factor, then adds the data values. When you use this keyword, you must include the WEIGHTBY argument keyword with a variable, formula, or relation as the weight object.
AVERAGE	Adds data values, then divides the sum by the number of data values that were added together. When you use AVERAGE, there are special considerations described in "Average Operators" on page 6-90.
HAVERAGE	(Hierarchical Average) Adds data values, then divides the sum by the number of the children in the dimension hierarchy. Unlike AVERAGE, which counts only non-NA children, HAVERAGE counts all of the logical children of a parent, regardless of whether each child does or does not have a value. This keyword is not affected by the setting of the NASKIP option for *argument*.
WAVERAGE	(Weighted Average) Multiplies each data value by a weight factor, adds the data values, and then divides that result by the sum of the weight factors. When you use this keyword, you must include the WEIGHTBY argument keyword with a variable, formula, or relation as the weight object.
HWAVERAGE	(Hierarchical Weighted Average) Multiplies non-NA child data values by their corresponding weight values then divides the result by the sum of the weight values. Unlike WAVERAGE, HWAVERAGE includes weight values in the denominator sum even when the corresponding child values are NA. When you use this keyword, you must include the WEIGHTBY argument keyword with a variable, formula, or relation as the weight object. This keyword is not affected by the setting of the NASKIP option for *argument*.
FIRST	The first non-NA data value.
HFIRST	(Hierarchical First) The first data value that is specified by the hierarchy, even when that value is NA. This keyword is not affected by the setting of the NASKIP option for *argument*.
WFIRST	(Weighted First) The first non-NA data value multiplied by its corresponding weight value. When you use this keyword, you must include the WEIGHTBY argument keyword with a variable, formula, or relation as the weight object.

Table 6–3 (Cont.) Aggregation Operators

Keyword	Description
HWFIRST	(Hierarchical Weighted First) The first data value that is specified by the hierarchy multiplied by its corresponding weight value, even when that value is `NA`. When you use this keyword, you must include the WEIGHTBY argument keyword with a variable, formula, or relation as the weight object. This keyword is not affected by the setting of the NASKIP option for *argument*.
LAST	The last non-`NA` data value.
HLAST	(Hierarchical Last) The last data value that is specified by the hierarchy, even when that value is `NA`. This keyword is not affected by the setting of the NASKIP option for *argument*.
WLAST	(Weighted Last) The last non-`NA` data value multiplied by its corresponding weight value. When you use this keyword, you must include the WEIGHTBY argument keyword with a variable, formula, or relation as the weight object.
HWLAST	(Hierarchical Weighted Last) The last data value that is specified by the hierarchy multiplied by its corresponding weight value, even when that value is `NA`. When you use this keyword, you must include the WEIGHTBY argument keyword with a variable, formula, or relation as the weight object. This keyword is not affected by the setting of the NASKIP option for *argument*.
MAX	The largest data value among the children of any parent data value.
MIN	The smallest data value among the children of any parent data value.
AND	When any child data value is FALSE, then the data value of its parent is FALSE. A parent is TRUE only when all of its children are TRUE. (BOOLEAN variables only)
OR	When any child data value is TRUE, then the data value of its parent is TRUE. A parent is FALSE only when all of its children are FALSE. (BOOLEAN variables only)
NOAGG	Do *not* aggregate any data for this dimension. Use this keyword only in an operator variable. It has no effect otherwise.

opvar
A text variable that specifies different the operation for each of its dimension values. The *opvar* argument is used in two ways:

- Measure dimension -- Changes the aggregation method depending upon the variable being aggregated. This is useful when a single aggmap is used to aggregate several variables that need to be aggregated with different methods. Whether you pre-aggregate all of the measures in a single AGGREGATE command or in separate statements, AGGREGATE uses the operation variable to identify the calculation method. The values of the measure dimension are the names of the variables to be aggregated. It dimensions a text variable whose values identify the operation to be used to aggregate each measure. The aggregation specification must include a MEASUREDIM (for aggregation) statement that identifies the measure dimension. See Example 6–30, "Aggregating Using a Measure Dimension" on page 6-59.
- Line item dimension -- Changes the aggregation method depending upon the line item being aggregated. The line item dimension is typically non-hierarchical and identifies financial allocations. The line item dimension is used both to dimension the data variable and to dimension a text variable that identifies the operation to be used to aggregate each item. The operation variable is typically used to aggregate line items over time. You do not use the MEASUREDIM (for aggregation) statement in the aggmap. See Example 6–31, "Aggregating Using a Line Item Dimension" on page 6-60.

The *opvar* argument cannot be dimensioned by the dimension it is used to aggregate. For example, when you want to specify different operations for the `geography` dimension, then *opvar* cannot be dimensioned by `geography`.

To minimize the amount of paging for the operator variable, define the operation variable as type of `TEXT` with a fixed width of `8`.

PARENTALIAS
Specifies that an alias dimension for the dimension being aggregated is QDRd to the parent value currently being aggregated.

dimension-alias-name
The name of the alias dimension for the dimension of *rel-name*.

ARGS
Indicates optional handling of the aggregation.

DIVIDEBYZERO
Specifies whether to allow division by zero.

YES allows division by zero; a statement involving division by zero executes without error but produces `NA` results.

NO disallows division by zero; a statement involving division by zero stops executing and produces an error message.

The default value is the current value of the DIVIDEBYZERO option.

DECIMALOVERFLOW
Specifies whether to allow decimal overflow, which occurs when the result of a calculation is very large and can no longer be represented by the exponent portion of the numerical representation.

YES allows overflow; a calculation that generates overflow executes without error and produces `NA` results.

NO disallows overflow; a calculation involving overflow stops executing and generates an error message.

The default value is the current value of the DECIMALOVERFLOW option.

NASKIP
Specifies whether `NA` values are input.

YES specifies that `NA` values are ignored when aggregating. Only actual values are used in calculations.

NO specifies that `NA` values are considered when aggregating. When any of the values being considered are `NA`, the calculation returns `NA`.

The default value is the current value of the NASKIP option.

The value that you specify for the NASKIP phrase does *not* effect calculation performed when you specify HAVERAGE, HFIRST, HLAST, HWAVERAGE, HWFIRST, HWLAST for *operation*.

WEIGHTBY
Indicates that weighted aggregation is to be performed. You must include a WEIGHTBY clause when you specify HWAVERAGE, HWFIRST, HWLAST, SSUM, WAVERAGE, WFIRST, WLAST, or WSUM for *operation*. The WEIGHTBY phrase always includes a *wobj* argument and can optionally include the WNAFILL keyword. It can also include several other deprecated keywords (see "Deprecated WEIGHTBY Keywords" on page 6-92 for details).

WNAFILL {*number*|NA}
Indicates handling for NA values. The default values for WNAFILL vary depending on the value of operation:

- `0.0` is the default for HWAVERAGE and SSUM.
- `1.0` is the default for HWFIRST, HWLAST, WAVERAGE, WFIRST, WLAST, and WSUM.
- NA is the default for OR.

Important: Be aware that WNAFILL defaults for each operator in an aggregation specification. In other words, when one RELATION statement includes a WSUM OPERATOR, then WNAFILL defaults to 1.0. When the next RELATION statement includes an SSUM OPERATOR, then WNAFILL defaults to `0.0`, and so on. See "Using WNAFILL" on page 6-94.

---

number substitutes a number for every NA value. That number will replace every NA value in the weight object, weight formula, or weight relation.

NA specifies that NA values are to be specified as NA.

wobj
A variable, formula, or relation that provides the weighted values. It can be numeric or BOOLEAN. When *wobj* is BOOLEAN, then `TRUE` has a weight of `1.0` and `FALSE` has a weight of `0.0`. A formula is queried only when needed, depending on the dimensionality of the formula and the of the variable being aggregated. When *wobj* is a relation, it should be a one-dimensional self-relation. See "Working with Weight Objects" on page 6-93 for more information about specifying values for *wobj*.

argsvar
A text variable that contains the *argument* options for some or all dimension values.

Notes

Specifying the Precomputed Data
The PRECOMPUTE clause of the RELATION statement limits the data that is aggregated by the AGGREGATE command. In its simplest form, you can think of the PRECOMPUTE clause as working like a LIMIT *dimension* TO statement. Notice that the default limit is on the dimension, which is not explicitly named in the RELATION statement.

For example, this LIMIT statement selects the `Audiodiv`, `Videodiv`, and `Accdiv` values of the `product` dimension:

```
LIMIT product TO 'Audiodiv' 'Videodiv' 'Accdiv'
```

The equivalent RELATION statement looks like this:

```
RELATION product.parentrel PRECOMPUTE ('Audiodiv' 'Videodiv' 'Accdiv')
```

Two Ways to use Valuesets

You can use valuesets in two different ways.

- You can use a valueset to limit hierarchy dimensions. You can limit which hierarchies will be used by the AGGREGATE command and AGGREGATE function, as well as the order in which these hierarchies should be used. The valueset that you use specifies the names of a dimension's hierarchies. To use a valueset in this way, use the following syntax.

  ```
  RELATION rel-name (valueset)
  ```

  In this case, using valuesets provides a way to manage hierarchies that are in conflict with each other, meaning, when the same dimension value stores data for different children in different hierarchies (such as, `Q1` stores data for `Jan`, `Feb`, and `Mar` in the `Calendar` hierarchy, but `Q1` stores data for `May`, `Jun`, and `Jul` in the `Fiscal` hierarchy).

- You can use a valueset to specify which values should be calculated on the fly by the AGGREGATE function and which values should be pre-calculated by the AGGREGATE command. The valueset that you use specifies the names of dimension values. To use a valueset in this way, use the following syntax.

  ```
  RELATION rel-name PRECOMPUTE (valueset)
  ```

  In this case, you use the valueset that follows the PRECOMPUTE keyword.

  When you use valuesets to limit hierarchy dimensions, you must also use the FORCECALC keyword in the AGGREGATE function when using more than one aggmap and the hierarchies are inconsistent.

When You Change a PRECOMPUTE or an OPERATOR Clause

Any time you make changes to a PRECOMPUTE or an OPERATOR clause, you should aggregate the variable data again and recompile the aggmap in order to produce accurate data.

Aggregating Data Loaded into Different Hierarchy Levels

When data is loaded into dimension values that are at different levels of a hierarchy, then you need to be careful in how you set status in the PRECOMPUTE clause in a RELATION statement in your aggregation specification.

Suppose that a `time` dimension has a hierarchy with three levels: months aggregate into quarters, and quarters aggregate into years. Some data is loaded into month dimension values, while other data is loaded into quarter dimension values. For example, `Q1` is the parent of `January`, `February`, and `March`. Data for `March` is loaded into the `March` dimension value. But the sum of data for `January` and `February` is loaded directly into the `Q1` dimension value. In fact, the `January` and `February` dimension values contain `NA` values instead of data. Your goal is to add the data in `March` to the data in `Q1`.

When you attempt to aggregate `January`, `February`, and `March` into `Q1`, the data in `March` will simply replace the data in `Q1`. When this happens, `Q1` will only contain the `March` data instead of the sum of `January`, `February`, and `March`.

To aggregate data that is loaded into different levels of a hierarchy, create a valueset for only those dimension values that contain data.

```
DEFINE all_but_q4 VALUESET time
LIMIT all_but_q4 TO ALL
LIMIT all_but_q4 REMOVE 'Q4'
```

Within the aggregation specification, use that valueset to specify that the detail-level data should be added to the data that already exists in its parent, `Q1`, as shown in the following statement.

```
RELATION time.r PRECOMPUTE (all_but_q4)
```

Average Operators

There are a number of issues involved in using the AVERAGE, HAVERAGE, WAVERAGE, and HWAVERAGE operators:

- Accuracy of when averaging—All decimal data is converted to floating point format, both for storing and for calculations, consequently, in some cases, an average aggregation computed on a DECIMAL or SHORTDECIMAL variable can differ in the least significant digits from a result you compute by hand. For this reason, you might want to use the `NUMBER` data type when accuracy is more important than computational speed, such as variables that contain currency amounts. See "Numeric Expressions" on page 3-11 for more information.

- Using Average operators when aggregating using an AGGREGATE Function—When you use an average operator in an aggregation specification that used by an AGGREGATE function, you must specify that you want Oracle OLAP to count the number of leaf nodes that contributed to an aggregate value when the AGGREGATE function executes. You can specify this behavior in either of the following ways:
  - Create $COUNTVAR properties for the variables that use the aggregation specification with a RELATION statement with an average operator.
  - Include the COUNTVAR keyword in the AGGREGATE command with an integer variable as its weight object when you use the specification to aggregate data.
- Using Average operators when aggregating using an AGGREGATE Command—When you use an average operator with the PRECOMPUTE keyword, the best practice is to use variables that have a decimal or NUMBER data type in order to ensure the accuracy of the results.
- Using Average operators for partial aggregations—When you use an average operator in a partial aggregation, then you must always use the same integer variable with the COUNTVAR keyword in the AGGREGATE commands. Do not change the values that are stored in the integer variable between AGGREGATE commands. Finally, the number of integer variables that you use with COUNTVAR must match the number of variables that are being aggregated by the AGGREGATE command. Following these rules will ensure the accuracy of your data.

HAVERAGE, HFIRST, HLAST, AND HWAVERAGE Operators

The "hierarchical" operators are intended to provide an alternative form of `NA` handling.

FIRST, HFIRST, LAST, AND HLAST Operators

These operators rely on the existing order of the dimension values, which are assumed to be the default logical order of that dimension. For example, in a month dimension, it is assumed that February follows January, March follows February, and so on.

When you need to change the default order, use the MAINTAIN statement to do so. For example, suppose `Q1` includes `January`, `February`, and `March`, but you need to make `February` the last month in the `Q1` instead of `March`. Use the following statement to do so.

```
MAINTAIN time MOVE 'Feb01' AFTER 'Mar01'
```

Now, the LAST operator will assume that `FEB01` is the last month in `Q1`.

Read Permissions and Aggmaps

When you change the read permission to *rel-name* in a RELATION statement, then you must recompile the aggmap before using it with the AGGREGATE function. This is not an issue when you use the AGGREGATE command, because the aggmap will be recompiled automatically. However, when you do not have read access to every *rel-name* in the aggmap, then attempting to use that aggmap will result in an error message.

Deprecated WEIGHTBY Keywords

Aggmap objects are reentrant and you can use a aggmap object to aggregate a weight object in the same way you would any other object. Consequently, the following keywords of the WEIGHTBY phrase are deprecated:

- **WAGG** or **WNOAGG** —WAGG aggregates the weight object. WAGG is the default when the weight object is a variable or relation (except when the operator is SSUM or WSUM). WNOAGG prevents the aggregation of a weight object. Using WNOAGG means that you will not be able to use WSOTRE and WNOSTORE. You can use WNOAGG with weight variables, weight formulas, and weight relations.
- **WSTORE** or **WNOSTORE—**WSTORE stores the aggregated weight object values in the weight object for later use. The *wobj* argument must be dimensioned exactly the same as the variable being aggregated; otherwise, the aggregated data might not be stored correctly. You cannot use WSTORE when the weight object is a formula. WNOSTORE stores the aggregated weight object values in a temporary variable so that they cannot be saved for later use. You cannot use WNOSTORE with a weight formula. WNOSTORE is the default when the weight object is a variable or relation.
- **WPREAGG —**WPREAGG specifies that the weight object is used only for weighting the detail data when aggregating the variable's data. Use this operator when every level of aggregate data can be aggregated either directly from detail data or aggregated into each intermediate level and still yield the same results. You do not have to specify this option, since it is the option for WEIGHTBY when the operator is any weight operation,

Using Weighted Aggregation Methods

When you use one of the weighted methods of aggregation (that is, HWAVERAGE, HWFIRST, HWLAST, SSUM, WAVERAGE, WFIRST, WLAST, or WSUM), you must define and populate an object that contains the weights. You identify the aggregation method in the OPERATOR clause and the weight object in the ARGS clause.

Working with Weight Objects

You must specify at least one weight object as the *wobj* argument when you use the WEIGHTBY keyword. The weight object can be a variable, a formula, or a relation. Special considerations apply depending on the type of object. the data type of the weight object, and whether or not you are performing a partial aggregation.

Weight Object Considerations Based on Type of Object The following considerations apply depending on the type of object that you use for the weight object:

- When the weight object is a variable, you can define it with a numeric or BOOLEAN data type. Use a variable as your weight object when you want to pre-calculate weight values and commit them to the database. You can use a variable weight object with any weight option.
- When the weight object is a relation, you should define it as a one-dimensional self-relation. *Y*ou can use the weight object to specify that the weight for a specific cell is contained in the current variable at a different location. Use a relation as your weight object when you use a line item or a measure dimension. In this case, one line item is used as the weight to calculate the aggregate value of another line item. Using a relation enables you to specify another set of cells in the variable being aggregated as the weight values for a weighted operation.
- When the weight object is a formula, that formula will be queried only as often as needed, depending on the dimensionality of the formula and the dimensionality of the variable whose data is being aggregated. You can define the formula with a numeric or BOOLEAN data type. Use a formula as your weight object when you want to calculate weight values on the fly. A formula weight object is similar to a variable weight object, except that it cannot be aggregated. The value of a formula weight object is executed dynamically. Therefore, you cannot use a formula weight object with many of the weight options.

Considerations Based on Data Type of the Weight Object The following considerations apply when the weight object is numeric or BOOLEAN:

- When the weight object has a numeric data type, It is good practice for the weight object variable to have the same dimensionality (or a subset thereof) as the variable to which it corresponds, but it is not required. When you use Oracle numbers or decimals to define your data variable, then always use the same data type to define the corresponding weight object. Otherwise, use the same data type for the weight object and the data variable unless you use WAVERAGE or HWAVERAGE; in this case, use a decimal or NUMBER data type to define the weight object.
- When the weight object variable, formula, or relation that you define has a `BOOLEAN` data type, then `TRUE` represents a weight of `1.0` and `FALSE` represents a weight of `0.0`. Furthermore, when an `NA` value is multiplied by any value, the result is `NA`.

Weight Object Considerations When Performing Partial Aggregations When you use any operators that require the WEIGHTBY phrase, and you are performing a partial aggregation, then do not change the values that are stored in the weight object between AGGREGATE commands.

Using WNAFILL

For example, suppose you use the WSUM operator to perform currency conversion. The currency conversation rates will be applied at the detail data level. Only the detail data needs to be converted, because the variable data is aggregated after the conversion. In order to get the correct results, all of the non-detail level weight values in the weight object would have to be `1`. Although this strategy produces correct results, it is inefficient. The best practice is to use the default WNAFILL value of `1`. This specifies that all `NA` values in the weight object should be treated as if they have a weight of `1`. In this case, because the operator is WSUM, you do not have to include WNAFILL in the AGGREGATE command, because the default values are correct.

For example, the following statement causes the value `0.7` to be substituted for every `NA` value in the `salesw` weight object.

```
AGGREGATE sales USING sales.agg WEIGHTBY WNAFILL 0.7 salesw
```

When you do not want to specify a number to replace `NA` values, then you can use `NA` instead of a number, as shown in the following statement.

```
AGGREGATE sales USING sales.agg WEIGHTBY WNAFILL NA salesw
```

Specifying NA after WNAFILL has the following effect:

- When the aggregation specification contains a WAVERAGE or a WSUM OPERATOR, then any child cell in the weight object that has an NA value is treated as an NA cell.
- When the aggregation specification contains an SSUM OPERATOR, then the results depend on how the Oracle OLAP option NASKIP is set. When NASKIP is set to YES, then any NA value is treated as 0.0. However, when NASKIP is set to NO, then any NA value is treated as an NA cell.

Effects of Dimension Status on Aggregation

A RELATION statement only aggregates those source data values that are in status. The parent values are calculated regardless of whether they are in status or not. For example, when only Jan01, Feb01, and Mar01 are in status for the time dimension, then Q1.01 is calculated (but no other quarters), and 2001 is calculated (but no other years) using only Q1.01 as input since the other quarters are NA.

This can be useful when you want to aggregate just the new data in your analytic workspace. However, you must exercise some care, as described in "Weight Object Considerations When Performing Partial Aggregations" on page 6-94.

Assume that there is a variable named sales that is dimensioned by time, a hierarchical dimension, and district, a non-hierarchical dimension.

```
DEFINE time DIMENSION TEXT
DEFINE time.parentrel RELATION time <time>
DEFINE district DIMENSION TEXT
DEFINE sales VARIABLE DECIMAL <time district>

REPORT DOWN time sales

             -----------------------SALES-----------------------
             ---------------------DISTRICT----------------------
TIME            North        South         West         East
------------ ------------ ------------ ------------ ------------
1976Q1         168,776.81   362,367.87   219,667.47   149,815.65
1976Q2         330,062.49   293,392.29   237,128.26   167,808.03
1976Q3         304,953.04   354,240.51   170,892.80   298,737.70
1976Q4         252,757.33   206,189.01   139,954.56   175,063.51
1976                   NA           NA           NA           NA
```

Skip-Level Aggregation

One aggregation strategy is skip-level aggregation illustrated in Example 6–32, "Skip-Level Aggregation" on page 6-61. With skip-level aggregation, you select one or two of the dimensions of a variable and pre-aggregate every other level in those dimension hierarchies. When you know which levels are queried most often by users, you should pre-calculate those levels of data.

Keep the following points in mind when designing skip-level aggregation:

- When selecting the dimensions to aggregate using skip-level aggregation:
    - Use a skip-level approach for only one or two dimensions. You should use the skip-level approach for half or fewer of the dimensions in a variable definition. For example, when there are three dimensions, then you can use the skip-level approach for one dimension; when there are four or more dimensions, then you can use the skip-level approach for two dimensions.
    - The dimensions that are the best candidates for skip-level aggregation are the dimensions whose hierarchies have many levels.
    - When possible, choose a dimension that is either fastest- or intermediate-varying in the variable dimension. Performance of calculation on the fly is always best for dimensions in this position.
- When selecting the levels to skip:
    - Consider skipping every other level in a dimension hierarchy, and avoid skipping more than two levels that are adjacent to each other. For example, when a hierarchy has seven levels, you might skip `L2`, `L4`, and `L6`. That means you would precalculate `L1`, `L3`, and `L5`. (The detail-level data is at `L7`.) Consider how frequently a level is queried. Users experience the best performance when you pre-aggregate the data most frequently queried, and aggregate on the fly the data that is requested occasionally.
    - Do not skip adjacent levels. For example, when you skipped `L2`, `L3`, `L4`, and `L5`, then a query for `L2` data would require AGGREGATE to calculate `L5`, then aggregate that data up to `L4`, then up to `L3`, and finally to `L2`. Alternatively, when you skip `L2`, `L4`, and L6, then a query for `L2` data requires AGGREGATE to aggregate data only from `L3`.
    - The one exception to this rule is when each level has very few children for each parent. When this is true for every adjacent level that you want to skip, then you can skip two or more adjacent levels.

Examples

For examples of aggregation specifications that include RELATION statements, see the examples in AGGMAP.

AGGMAP ADD or REMOVE model

The AGGMAP ADD or REMOVE model command adds or removes a model from a previously defined aggmap object of type AGGMAP. Models are used in aggmap objects to aggregate data over a non-hierarchical dimension (such as line items), which has no parent relation and therefore cannot be aggregated by a RELATION (for aggregation) statement. See MODEL (in an aggregation) for details.

Syntax

AGGMAP {ADD *model* TO *aggmap*|REMOVE *model* FROM *aggmap*}

Arguments

ADD
Temporarily adds a model to an aggmap object. The model is attached to the aggmap only for the duration of the session. Even when the analytic workspace has been updated and committed, the model is discarded from the aggmap when the session is closed.

REMOVE
Removes a model from an aggmap.

model
The name of the model object that you wish to add to the specified aggmap.

aggmap
The name of a previously defined aggmap object of type AGGMAP.

Notes

Creating Temporary or Custom Aggregates

Most aggmap objects are defined to calculate variable values that are dimensioned by permanent dimension members (that is, dimension members that persist from one session to another). However, users might wish to create their own aggregates at runtime for forecasting or what-if analysis, or just because they want to view the data in an unforeseen way. Adding temporary members to dimensions and aggregating data for those members is sometimes called creating temporary or custom aggregates.

Examples

Example 6–34 Temporarily Adding a Model to an Aggmap

Assume for example, that you have an aggmap object named `letter.aggmap` with the following definition.

```
DEFINE LETTER.AGGMAP AGGMAP
AGGMAP
RELATION letter.letter PRECOMPUTE ('AA')
END
```

Assume also that you want to create summarized variable data for the cells that are dimensioned by the dimension values `AAB` and `ABA`. However, you do not want this data to be permanently stored in the analytic workspace. You just want to see the data during your session.

To perform this type of aggregation, you can take the following steps:

1. Create a dimension value for the custom aggregate. This dimension value will be the parent of the dimension values `AAB` and `ABA`. The following statement adds '`BB`' to the `letter` dimension:

   ```
   MAINTAIN letter ADD 'BB'
   ```

2. Create a `MODEL` object that contains an `AGGREGATION` function, which associates child dimension values with the new dimension value. The following model identifies `BB` as the parent of `AAB` and `ABA`. Note that the parent dimension value (in this case, BB) cannot already be defined as a parent in the parent relation (`letter.letter`).

   ```
   DEFINE LETTER.MODEL MODEL
   MODEL
   DIMENSION letter
   BB=AGGREGATION('AAB' 'ABA')
   ```

3. Execute an `AGGMAP ADD` statement to append the model to the existing `AGGMAP` object.

   ```
   AGGMAP ADD letter.model TO letter.aggmap
   ```

The aggmap now looks like this.

```
DEFINE LETTER.AGGMAP AGGMAP
AGGMAP
RELATION letter.letter PRECOMPUTE ('AA')
END
AGGMAP ADD letter.model
```

4. The model is executed only by the `AGGREGATE` function like the one shown here; the `AGGREGATE` command ignores it.

```
REPORT AGGREGATE(units USING letter.aggmap)
```

5. When you wish to remove the model from the aggmap during a session, use the `AGGMAP REMOVE` statement.
6. To ensure that your aggmap does not become a permanent object in the analytic workspace, before you close your session issue the following statement to delete the dimension values that you added in Step 1.

```
MAINTAIN letter DELETE 'BB'
```

When your session ends, Oracle OLAP automatically removes the model added using the AGGMAP ADD statement. You do not have to issue an explicit AGGMAP REMOVE statement.

AGGMAP SET

Specifies the default aggmap for a variable.

Note: You can also use an $AGGMAP property to specify the default aggregation specification for a variable or the $ALLOCMAP property to specify the default allocation specification for a variable.

Syntax

AGGMAP SET *aggmap* AS DEFAULT FOR *variables*

Arguments

aggmap
The name of a previously defined aggmap object.

variables
A text expression that is the name of one or more variables for which the specified aggmap is the default aggmap. When you specify a literal value, separate the names of the variables with commas.

Examples

Example 6–35 Using AGGMAP SET to Specify a Default Aggmap

Example 6–2, "Using the $AGGREGATE_FROM Property" on page 6-6 illustrates how the AGGREGATE command shown in Example 6–22, "Using a CACHE Statement in an Aggregation Specification" on page 6-52 can be simplified to the following statement.

```
AGGREGATE sales_by_revenue USING revenue_aggmap
```

You can further simplify the AGGREGATE command if you make `revenue_aggmap` the default aggmap for `sales_by_revenue` variable. You can do this using either by defining an $AGGMAP property on the `sales_by_revenue` variable or by issuing the following statement.

```
AGGMAP SET revuienue_aggmap AS DEFAULT FOR sales_by_revenue
```

Now you can aggregate the data by issuing the following AGGREGATE command that does not include a USING clause.

```
AGGREGATE sales_by_revenue
```

7 AFFMAPINFO to ARCCOS

This chapter contains the following OLAP DML statements:

- ANTILOG
- ANTILOG10
- ANY
- ARCCOS

AGGMAPINFO

The AGGMAPINFO function returns information about the specification of an aggmap object in your analytic workspace.

Note: You can get information about an aggregation specification (that is, an aggmap of type AGGMAP) only after it has been compiled. You can compile an aggregation specification using a COMPILE statement or by including the FUNCDATA keyword when you execute the AGGREGATE command. When an aggregation specification has not been compiled before you use it with the AGGMAPINFO function, then it is compiled by AGGMAPINFO. You do not need to compile an aggmap for use with ALLOCATE.

Return Value

Varies depending on the type of information that is requested. See Table 7–1, " Keywords for the choice Parameter of the AGGMAPINFO function" on page 7-4 for more information.

Syntax

AGGMAPINFO (*name* {*choice* | {*choice-at-position rel-pos*} })

Arguments

name
The name of the aggmap object.

choice
Specifies the type of information that you want returned. See Table 7–1, " Keywords for the choice Parameter of the AGGMAPINFO function", for details.

Table 7–1 Keywords for the choice Parameter of the AGGMAPINFO function

Keyword	Return Value	Description
ADDED_MODELS	TEXT	The models that are currently added to an aggmap using AGGMAP ADD or REMOVE model statements.The names of the models are returned as a multi-line text string.
AGGINDEX	BOOLEAN	Indicates the setting for the AGGINDEX statement in the aggmap. A `YES` setting specifies that all possible indexes (composite tuples) are created whenever the aggmap is recompiled. (Applies to AGGMAP type aggmaps only.)
CHILDREN *member-name*	TEXT	The dimension members used in the right-hand side of equations used to calculate temporary calculated members added using MAINTAIN ADD SESSION statements. The names of the members are returned as a multi-line text string.
CUSTOMMEMBERS	TEXT	The members added using MAINTAIN ADD SESSION statements. The names of the members are returned as a multi-line text string.
DIMENSION	TEXT	The names of the dimensions of the models or relations used by the aggmap. The names of the members are returned as a multi-line text string.
FCACHE	BOOLEAN	Indicates whether Oracle OLAP has a cache for the AGGREGATE function. (Applies to AGGMAP type aggmaps only.)

Table 7–1 (Cont.) Keywords for the choice Parameter of the AGGMAPINFO function

Keyword	Return Value	Description
MAPTYPE	TEXT	The type of the aggmap. ▪ Returns AGGMAP for an aggregation specification (that is, when the specification has been entered with the AGGMAP command). You can use this type of aggmap only with the AGGREGATE command or AGGREGATE function. ▪ Returns ALLOCMAP for an allocation specification (that is, when the specification has been entered with the ALLOCMAP command). You can use this type of aggmap only with the ALLOCATE command. ▪ Returns `NA` when the aggmap has been defined but a specification has not been entered with the AGGMAP or ALLOCMAP command.
MODELS	TEXT	The models in the aggmap. The names of the models are returned as a multi-line text string.
NUMRELS	INTEGER	The total number of RELATION (for aggregation) statements in an aggmap specification .
RELATIONS	TEXT	The name of relation that is specified by a RELATION (for aggregation) statement in the aggmap object. Each statement is displayed on a separate line.
STORE	BOOLEAN	Indicates whether the CACHE statement in the aggmap is set to STORE. A `YES` setting specifies that the data that is calculated on the fly is stored in the cache. (Applies to AGGMAP type aggmaps only.)
VARIABLES	TEXT	The variables for which this aggmap object has been specified as the default aggmap using AGGMAP ADD or REMOVE model statements or the $AGGMAP property. The names of the variables are returned as a multi-line text string.

choice-at-position
Specifies the exactly which piece of information you want returned.

PRECOMPUTE returns the text of the *limit-clause* that follows the PRECOMPUTE keyword in a RELATION statement. You must use the *rel-pos* argument to specify a single RELATION statement. Returns `NA` when the RELATION statement does not have a PRECOMPUTE keyword. (Applies to AGGMAP type aggmaps only.)

RELATION returns the name of the relation that follows the RELATION statement that you specify with the *rel-pos* argument.

STATUS returns the status list that results from the compilation of the PRECOMPUTE clause in the RELATION statement that you specify with the *rel-pos* argument. (Applies to AGGMAP type aggmaps only.)

rel-pos
An INTEGER that specifies a RELATION statement in the aggmap. The integer indicates the position of the statement in the list of RELATION statements. You can use the *rel-pos* argument only with the RELATION, PRECOMPUTE, or STATUS keywords. For example, to get information about the first RELATION statement in an aggmap, use the integer `1` as the *rel-pos* argument. To get information about the fourth RELATION statement in an aggmap, use the integer `4`, and so on. You may use any integer between `1` and the total number of RELATION statements in an aggmap specification. You can use the NUMRELS keyword to obtain the total number of RELATION statements for an aggmap object.

Examples

Example 7–1 Retrieving Information About an Aggmap Object

Suppose an aggmap named `sales.agg` has been defined with the following statement.

```
DEFINE sales.agg AGGMAP <time, product, geography>
```

Suppose the following specification has been added to `sales.agg` with the AGGMAP command.

```
AGGMAP
RELATION time.r PRECOMPUTE (time ne 'Year98')
RELATION product.r
RELATION geography.r
CACHE STORE
END
```

Once a specification has been added to the aggmap, you can use AGGMAPINFO to get information about its specification.

To see the names of the hierarchies that are specified by the RELATION statements, use the following statement.

```
SHOW AGGMAPINFO(sales.agg RELATIONS)
```

The following results are displayed.

```
time.r
product.r
geography.r
```

The following statement and result tell you how many RELATION statements are in the aggmap object.

```
SHOW AGGMAPINFO(sales.agg NUMRELS)
3
```

The following statement and result verifies that data that is calculated on the fly is stored in the cache for the session. The result is `YES` because the aggmap contains a CACHE STORE statement.

```
show AGGMAPINFO(sales.agg STORE)
YES
```

The following statement displays the relation name that is specified in the second RELATION statement in the aggmap.

```
SHOW AGGMAPINFO(sales.agg RELATION 2)
product.r
```

The following statement displays the limit-clause that follows the PRECOMPUTE keyword in the first RELATION statement in the aggmap.

```
SHOW AGGMAPINFO(sales.agg PRECOMPUTE 1)
time NE 'YEAR98'
```

Suppose the `time` dimension values are `Jan98` to `Dec99`, `Year98`, and `Year99`. The following statement displays the status list for the dimension in the first RELATION statement in the aggmap.

```
SHOW AGGMAPINFO(sales.agg STATUS 1)
Jan98 TO Dec99, Year99
```

Because the *limit-clause* in the RELATION statement specifies that the `time` dimension values should not equal `Year98`, all `time` dimension values other than `Year98` are included in its status.

The following statement displays the aggmap type of `sales.agg`.

```
SHOW AGGMAPINFO(sales.agg MAPTYPE)
AGGMAP
```

AGGREGATE command

The AGGREGATE command calculates summary data from detail data. Use the AGGREGATE command to pre-calculate data and store it in an Oracle OLAP analytic workspace. Use the AGGREGATE function to calculate data at runtime. In either case, the aggregation is limited to the base values that are currently in status.

Syntax

AGGREGATE|AGGR *var*... [USING *aggmap*] [FROM *fromspec*|FROMVAR *textvar*]

[FUNCDATA] [COUNTVAR *intvar*...]

Arguments

var
One or more variables whose data values are to be calculated. Every variable in a single AGGREGATE command must have exactly the same dimensions in exactly the same order. The variables are often numeric, but can also be `TEXT`, `DATETIME`, or `DATE` when the aggregation operation is FIRST, LAST, MIN, or MAX, as specified in the aggmap.

USING
This keyword indicates that the aggregation is performed using the specified aggmap. When you do not include this phrase, the command uses the default aggmap for the variable as previously specified using the AGGMAP command or the $AGGMAP property.

aggmap
The name of a previously-defined aggmap that specifies how the data will be aggregated. For information about aggmaps, see the DEFINE AGGMAP command.

FROM
This keyword indicates that the detail data is obtained from a different object.

A FROM clause is only one way in which you can specify the variable from which detail data should be obtained when performing aggregation. See "Ways of Specifying Where to Obtain Detail Data for Aggregation" on page 7-13.

fromspec
An arbitrarily dimensioned variable, formula, or relation from which the detail data for the aggregation is obtained.

FROMVAR
This keyword indicates that the detail data is obtained from different objects to perform a capstone aggregation. (For an example of using the FROMVAR clause, see Example 7–7, "Capstone Aggregation" on page 7-17.)

A FROMVAR clause is only one way in which you can specify the variable from which detail data should be obtained when performing aggregation. See "Ways of Specifying Where to Obtain Detail Data for Aggregation" on page 7-13.

textvar
An arbitrarily dimensioned variable used to resolve any leaf nodes. Specify `NA` to indicate that a node does not need detail data to calculate the value.

FUNCDATA
Compiles the aggregation specification for future use by the AGGREGATE function. When you use FUNCDATA, you do not have to recompile the aggmap before using the AGGREGATE function, unless afterward you make changes to the aggmap, the relation hierarchies, or a composite.

When the variables have composite dimensions, the indexes (composite tuples) are created and saved for use by the AGGREGATE function. Otherwise, the indexes are re-created each time the AGGREGATE function is called. Refer to AGGINDEX for more information about composite indexes.

COUNTVAR
Indicates that the number of leaf nodes that contributed to an aggregate value are counted. Leaf nodes that are `NA` are not included in the tally. You must include a COUNTVAR phrase when the aggmap contains a RELATION (for aggregation) statement that uses the AVERAGE operator.

intvar
A variable that you have defined with an INTEGER data type. The definition of *intvar* must have exactly the same dimensions in exactly the same order as the dimensions in *var*. When you aggregate several variables together, you must define an INTEGER variable for each one to record the results.

Notes

Terminology

A dimension hierarchy is a tree structure in which the dimension values are the nodes. At the lowest level of the hierarchy are leaves or leaf nodes, and at the highest level is the root or root node. Nodes in a hierarchy have parent-child relationships. Leaf nodes have parents but no children; root nodes have children but no parents.

Effect of Status

The current status only affects dimension values at the lowest level of the hierarchy, that is, the leaf nodes. Only leaf-node dimension values that are currently in status are aggregated. The parent values of leaf nodes in status are calculated, whether the parent values are in status or not (unless you exclude the dimension values in those levels with a RELATION PRECOMPUTE statement in the aggmap). Thus, when you want to aggregate all of the data specified in the aggmap, then be sure to set the status of the dimensions to ALL before performing the aggregation. AGGREGATE uses the parent relation to distinguish among dimension values at different levels of the hierarchy. Alternatively, you can perform a partial aggregation of the data by limiting status. However, this must be done carefully when some of the data will be aggregated at runtime by the AGGREGATE function. See the AGGREGATE function notes for more information.

For example, suppose you use the `area` dimension and the `area.area` child-parent relation that supports one hierarchy for a geography dimension as illustrated in Table 7–2, " Geography Hierarchy".

Table 7–2 Geography Hierarchy

Level	area Dimension	area.area Parent Relation
1	`TotalUS`	`NA`
2	`East`	`TotalUS`
2	`South`	`TotalUS`
3	`Boston`	`East`
3	`New York`	`East`
3	`Atlanta`	`South`

Now suppose you change the data value for `New York`. When you then use AGGREGATE with only `New York`, the calculation occurs without including the child value for `South` (`Atlanta`), but still includes level `2` as it goes from level `3` to level `1` (`TotalUS`). When you want *all* the child values included in rolling up to `TotalUS`, use a `LIMIT TO ALL` statement before you execute the AGGREGATE command.

When the data has changed for some, but not all, of the child values in a hierarchy, you can set the status to calculate just the values that have changed. For example, when your embedded-total dimension is called `d2`, and its parent relation is called `reld2`, first limit `d2` to the values that have changed.

To calculate the data for every hierarchy in a dimension, limit the dimension's hierarchy dimension to `ALL` before you execute the AGGREGATE command.

Controlling the Amount of Data That Is Calculated

You can control how much of the variable data is calculated by using the PRECOMPUTE keyword with the RELATION statement in the aggmap. Use the limit-clause (after the PRECOMPUTE keyword) to set the status of the dimension.

When Users Modify Data

When users are able to change the data in a variable, then you should calculate aggregates on the fly using the AGGREGATE function, so that their changes are reflected in the aggregate data. See the AGGREGATE function for more information about runtime changes to the data.

Generation-Skipping Hierarchies

AGGREGATE automatically distinguishes between generations in the parent relation, even to the extent of allowing *generation-skipping* hierarchies. For example, you can have a four-level hierarchy (for example, `neighborhoods`, `cities`, `states`, and `totalUS`) that has a three-level branch (for example, `Boston`, `Massachusetts`, and `totalUS`).

Restrictions on Permissions

AGGREGATE does not work on variables that have cell-by-cell permissions; it will immediately return an error. It also ignores the PERMITERROR option. However, AGGREGATE will operate on variables with object level or dimension level permission. See the PERMIT and PERMITERROR entries.

Ways of Specifying Where to Obtain Detail Data for Aggregation

You can specify where to obtain detail data when aggregating data in the following ways:

- Assign either a $AGGREGATE_FROM property or a $AGGREGATE_FROMVAR property to a variable.

  Note: You can only assign one of these properties to a variable. A variable cannot have both the $AGGREGATE_FROM and $AGGREGATE_FROMVAR properties assigned to it.

- Include either a FROM or FROMVAR clause in the AGGREGATE command or AGGREGATE function that aggregates the data.

When performing an aggregation, Oracle OLAP determines where to obtain the detail data as follows:

1. When a location has been specified using a FROM or FROMVAR clause, Oracle OLAP uses the detail data at that location.
2. When a location has not been specified using a FROM or FROMVAR clause, Oracle OLAP checks to see if a location has been specified using aa $AGGREGATE_FROM property or a $AGGREGATE_FROMVAR property. When a location has been specified using one of these properties, Oracle OLAP uses the detail data at that location.
3. When a location has not been specified using either FROM or FROMVAR clause or a $AGGREGATE_FROM property or a $AGGREGATE_FROMVAR property, Oracle OLAP performs the aggregation using the detail data in the variable itself.

Examples

This section contains several examples of using the AGGREGATE command. For additional aggregation examples, see the examples in AGGMAP.

Example 7–2 Precalculating Data in a Batch Job

Frequently, you generate precalculated aggregates in a batch window as part of maintaining the data in your database. When you wish, you can use Job Manager to schedule batch jobs in Oracle Enterprise Manager, as described in the *Oracle OLAP Application Developer's Guide.*

To generate precalculated aggregates, you use the AGGREGATE command. The AGGREGATE command aggregates the data for one or more variables according to the specifications provided in the aggmap.

Your batch job should include statements like the following.

```
POUTFILEUNIT=FILEOPEN('userfiles/progress.txt' WRITE)
AGGREGATE sales units USING gpct.aggmap
UPDATE
COMMIT
FILECLOSE POUTFILEUNIT
```

Example 7–3 Aggregating One Variable

Suppose your analytic workspace contains a variable named `actuals`, which has the following definition.

```
DEFINE actuals DECIMAL <time, SPARSE <product, customer, channel>>
```

The next step is to define an aggmap object, whose definition has the same dimensions in the same dimension order. Suppose you define an aggmap object named `act.agg` using the DEFINE AGGMAP command.

```
DEFINE act.agg AGGMAP <time, SPARSE <product, customer, channel>>
```

Suppose that the name of the hierarchy for the `time` dimension is `time.r`, the name of the `product` dimension is `product.r`, and so on Next, you use the AGGMAP command to add the following text in the `act.agg` aggmap.

```
AGGMAP
RELATION time.r
RELATION product.r
RELATION customer.r
RELATION channel.r
END
```

The preceding text specifies the name of each dimension's hierarchy for which data should be rolled up. Assuming that the current status of every dimension is `ALL`, data will be calculated for every dimension value of every dimension in the definition of `actuals`. No data will be calculated on the fly.

Use the following statements to calculate the `actuals` variable. (It is not necessary to compile the aggmap, because the compilation is included as part of the AGGREGATE command.)

```
AGGREGATE actuals USING act.agg
```

Example 7–4 Aggregating Multiple Variables

Suppose your analytic workspace contains a variable named `actuals` and a variable named `forecast`. As shown in the following variable definitions, these variables have the same dimensions in the same dimension order.

```
DEFINE actuals DECIMAL <time, SPARSE <product, customer, channel>>
DEFINE forecast DECIMAL <time, SPARSE <product, customer, channel>>
```

The next step is to define an aggmap object, whose definition has the same dimensions in the same dimension order. Suppose you define the same aggmap object named `act.agg`, as described in "Aggregating One Variable" on page 7-14. As long as you want the data for each variable to be rolled up in exactly the same way, you can use the same aggmap to calculate both variables in a single command.

Use the following statements to calculate the `actuals` and the `forecast` variables.

```
AGGREGATE actuals forecast USING act.agg
```

Because the aggmap specifies that all data for every dimension value in each dimension should be rolled up, this command rolls up all of the data in `actuals` and all of the data in `forecast`.

Example 7–5 Using COUNTVAR with Multiple Variables

Suppose you plan to use one AGGREGATE command to aggregate the data for three variables: `sales`, `units`, and `projected_sales`. Each variable has the following dimensionality.

```
<month product geography>
```

To tally the results with COUNTVAR, you must define three INTEGER variables that have the same dimensionality as `sales`, `units`, and `projected_sales`.

```
DEFINE intsales INTEGER <month product geography>
DEFINE intunits INTEGER <month product geography>
DEFINE intprojsales INTEGER <month product geography>
```

You can then specify the INTEGER variables in the following command:

```
AGGREGATE sales units projected_sales USING sales.agg -
  COUNTVAR intsales intunits inprojsales
```

Example 7–6 Performing a Partial Aggregation

This example limits the `time` dimension to the last two time periods, so that only newly loaded data is aggregated.

The `tp2.agg` aggmap specifies preaggregation for all detail data currently in status.

```
DEFINE TP2.AGG AGGMAP
LD Full preaggregation
AGGMAP
RELATION time.parentrel PRECOMPUTE (ALL)
RELATION product.parentrel PRECOMPUTE (ALL)
END
```

For the aggregation, `time` is limited to the last two time periods and all `product` values are in status.

```
LIMIT time TO LAST 2
STATUS time product
The current status of TIME is:
Apr02, May02
LIMIT product TO ALL
```

The AGGREGATE command calculates `units` using the `tp2.agg` aggmap.

```
AGGREGATE units USING tp2.agg
```

The results of this aggregation show that parent values are calculated, regardless of their own status, when their children are in status.

```
LIMIT time TO '2002' 'Q1.02' 'Q2.02' 'Jan02' to 'May02'
REPORT DOWN time units
```

```
 ----------------------------------------UNITS----------------------------------------
 ---------------------------------------PRODUCT---------------------------------------
TIME     FOOD      SNACKS    DRINKS   POPCORN   COOKIES   CAKES     SODA      JUICE
-------  --------  --------  -------- --------  --------  --------  --------  --------
2002     38        24        14       6         9         9         9         5
Q1.02    NA        NA        NA       NA        NA        NA        NA        NA
Q2.02    38        24        14       6         9         9         9         5
Jan02    NA        NA        NA       8         2         4         5         8
Feb02    NA        NA        NA       5         3         2         2         5
Mar02    NA        NA        NA       3         4         4         2         4
Apr02    21        13        8        2         7         4         6         2
May02    17        11        6        4         2         5         3         3
```

Example 7–7 Capstone Aggregation

Assume that your analytic workspace has the two hierarchical `TEXT` dimensions named `geog.d` and `time.d` with the following values.

```
GEOG.D
--------------
Boston
Medford
San Diego
Sunnydale
Massachusetts
California
United States

TIME.D
--------------
Jan76
Feb76
Mar76
76Q1
```

Assume, also, that there are four variables with the following definitions

```
DEFINE sales_jan76 VARIABLE INTEGER <geog.d>
DEFINE sales_feb76 VARIABLE INTEGER <geog.d>
DEFINE sales_mar76 VARIABLE INTEGER <geog.d>
DEFINE sales_capstone76 VARIABLE INTEGER <geog.d time.d>
```

Assume that you issue the following REPORT statements for the variables. The output of the reports show the detail data in the variables.

```
REPORT sales_jan76  sales_feb76 sales_mar76
REPORT DOWN geog.d sales_capstone76

GEOG.D         SALES_JAN76  SALES_FEB76  SALES_MAR76
-------------- ------------ ------------ ------------
Boston                1,000        2,000        3,000
Medford               2,000        4,000        6,000
San Diego             3,000        6,000        9,000
Sunnydale             4,000        8,000       12,000
Massachusetts            NA           NA           NA
California               NA           NA           NA
United States            NA           NA           NA

               -----------------SALES_CAPSTONE76------------------
               ----------------------TIME.D-----------------------
GEOG.D            Jan76        Feb76        Mar76         76Q1
-------------- ------------ ------------ ------------ ------------
Boston                   NA           NA           NA           NA
Medford                  NA           NA           NA           NA
San Diego                NA           NA           NA           NA
Sunnydale                NA           NA           NA           NA
Massachusetts            NA           NA           NA           NA
California               NA           NA           NA           NA
United States            NA           NA           NA           NA
```

1. Define two aggmap objects with the following definitions.

```
DEFINE leaf_aggmap AGGMAP
AGGMAP
RELATION geog.parentrel OPERATOR SUM
END

DEFINE capstone_aggmap AGGMAP
AGGMAP
RELATION time.parentrel OPERATOR SUM
END
```

2. Define a variable named `capstone_source` with the following definition to use to aggregate the data.

```
DEFINE capstone_source VARIABLE TEXT <time.d>
```

As the following output of a REPORT statement illustrates, for each value of `time.d`, you populate `capstone_source` with the name of the variable that contains the corresponding sales data.

```
TIME.D           CAPSTONE_SOURCE
-------------- ---------------------
Jan76          sales_jan76
Feb76          sales_feb76
Mar76          sales_mar76
76Q1           NA
```

3. Issue the following statements to aggregate the variables.

```
AGGREGATE sales_jan76 sales_feb76 sales_mar76 USING leaf_aggmap
AGGREGATE sales_capstone76 USING capstone_aggmap FROMVAR capstone_source
```

After aggregating the variables, when you issue the REPORT statements, the variables are populated with the calculated data.

```
REPORT sales_jan76  sales_feb76 sales_mar76
REPORT DOWN geog.d sales_capstone76

GEOG.D         SALES_JAN76  SALES_FEB76  SALES_MAR76
-------------- ------------ ------------ ------------
Boston                1,000        2,000        3,000
Medford               2,000        4,000        6,000
San Diego             3,000        6,000        9,000
Sunnydale             4,000        8,000       12,000
Massachusetts         3,000        6,000        9,000
California            7,000       14,000       21,000
United States        10,000       20,000       30,000

               -----------------SALES_CAPSTONE76------------------
               ----------------------TIME.D----------------------
GEOG.D            Jan76        Feb76        Mar76         76Q1
-------------- ------------ ------------ ------------ ------------
Boston                1,000        2,000        3,000        6,000
Medford               2,000        4,000        6,000       12,000
San Diego             3,000        6,000        9,000       18,000
Sunnydale             4,000        8,000       12,000       24,000
Massachusetts         3,000        6,000        9,000       18,000
California            7,000       14,000       21,000       42,000
United States        10,000       20,000       30,000       60,000
```

Example 7–8 Aggregating a Variable with External Partitions

Assume that you have the following objects defined in your analytic workspace.

```
DEFINE YEAR_2003 DIMENSION TEXT
DEFINE YEAR_2002 DIMENSION TEXT
DEFINE PRODUCT DIMENSION TEXT
DEFINE SALES_2003 VARIABLE DECIMAL <YEAR_2003 PRODUCT>
DEFINE SALES_2002 VARIABLE DECIMAL <YEAR_2002 PRODUCT>
DEFINE TIME DIMENSION CONCAT (YEAR_2003 YEAR_2002) UNIQUE
DEFINE TIME_PARENTREL RELATION TIME <TIME>
DEFINE PART_TEMP_SALES_BY_YEAR PARTITION TEMPLATE <TIME PRODUCT> -
   PARTITION BY CONCAT (TIME) -
    (PARTITION PARTITION_2002 <YEAR_2002 PRODUCT> -
     PARTITION PARTITION_2003 <YEAR_2003 PRODUCT>)
DEFINE SALES VARIABLE DECIMAL <PART_TEMP_SALES_BY_YEAR <TIME PRODUCT>> -
    (PARTITION PARTITION_2002 EXTERNAL SALES_2002 -
     PARTITION PARTITION_2003 EXTERNAL SALES_2003)
DEFINE AGG_SALES AGGMAP
  AGGMAP
  RELATION time_parentrel OPERATOR SUM
  END
```

To aggregate `sales` , you issue the following statement.

```
AGGREGATE sales USING agg_sales
```

When you issue REPORT statements on sales, you can see the aggregated values in sales.

```
           --------SALES--------
           -------PRODUCT-------
TIME         00001      00002
---------- ---------- ----------
01Jan2003       10.00      15.21
31Jan2003       10.88      13.37
01Dec2003          NA         NA
31Dec2003          NA         NA
Jan2003         20.88      28.58
Dec2003            NA         NA
2003            20.88      28.58
01Jan2002       14.44      11.03
31Jan2002       15.55      12.20
01Dec2002       11.39      12.80
31Dec2002       10.53      13.77
Jan2002         29.98      23.23
Dec2002         21.92      26.57
2002            51.91      49.80
```

Since `sales_2002` and `sales_2003` are external partitions of `sales`, aggregating `sales` effectively means that you aggregated `sales_2002` and `sales_2003`. When you issue REPORT statements on`sales_2002` and `sales_2003`, you can see the aggregated values in those variables.

```
           -----SALES_2002------
           -------PRODUCT-------
YEAR_2002    00001      00002
---------- ---------- ----------
01Jan2002       14.44      11.03
31Jan2002       15.55      12.20
01Dec2002       11.39      12.80
31Dec2002       10.53      13.77
Jan2002         29.98      23.23
Dec2002         21.92      26.57
2002            51.91      49.80
```

```
           -----SALES_2003------
           -------PRODUCT-------
YEAR_2003    00001      00002
---------- ---------- ----------
01Jan2003       10.00      15.21
31Jan2003       10.88      13.37
01Dec2003          NA         NA
31Dec2003          NA         NA
Jan2003         20.88      28.58
Dec2003            NA         NA
2003            20.88      28.58
```

AGGREGATE function

The AGGREGATE function calculates the data of a variable at runtime, in response to a user's request. The AGGREGATE function returns the requested data by retrieving stored values and calculating the remaining values.

Return Value

The same data type as the aggregated variable

Syntax

AGGREGATE (*var* [USING *aggmap*] [FROM *fromspec*|FROMVAR *textvar*] -

    [FORCECALC FORCEORDER] [COUNTVAR *intvar*])

Arguments

var
The name of the variable whose data will be calculated (if necessary) and returned. It is frequently numeric, but can also be `BOOLEAN`, `TEXT`, `DATETIME`, or `DATE` depending on the operator specified in the RELATION (for aggregation) statements in the aggmap specification.

USING
This keyword indicates that the aggregation is performed using the specified aggmap. When you do not include this phrase, the function uses the default aggmap for the variable as previously specified using the AGGMAP command or the $AGGMAP property.

aggmap
The name of a previously-defined aggmap that specifies how the data will be aggregated. For information about aggmaps, see the DEFINE AGGMAP command.

FROM
This keyword indicates that the detail data is obtained from a different object. A FROM clause is only one way in which you can specify the variable from which detail data should be obtained when performing aggregation. See "Ways of Specifying Where to Obtain Detail Data for Aggregation" on page 7-13.

fromspec
An arbitrarily dimensioned variable, formula, or relation from which the detail data for the aggregation is obtained.

FROMVAR
This keyword indicates that the detail data is obtained from different objects to perform a capstone aggregation. A FROMVAR clause is only one way in which you can specify the variable from which detail data should be obtained when performing aggregation. See "Ways of Specifying Where to Obtain Detail Data for Aggregation" on page 7-13.

textvar
An arbitrarily dimensioned variable used to resolve any leaf nodes. Specify `NA` to indicate that a node does not need detail data to calculate the value.

FORCECALC
Specifies that any value that is not specified in the aggmap's PRECOMPUTE clause should be recalculated, even when there is a value stored in the desired cell. Use the FORCECALC keyword when you want users to be able to change detail data cells and see the changed values reflected in dynamically-computed aggregate cells.

FORCEORDER
Specifies that the calculation must be performed in the order in which the RELATION (for aggregation) statements are listed, which should be from the fastest varying dimension to the slowest varying dimension. Use this option when you have changed some of the values calculated by the AGGREGATE command. Otherwise, the optimization methods used by the AGGREGATE function may cause the modified values to be ignored. FORCEORDER will slow performance.

COUNTVAR
Indicates that the number of leaf nodes that contributed to an aggregate value are counted. Leaf nodes that are `NA` are not included in the tally. You must include a COUNTVAR phrase when the aggmap contains a RELATION (for aggregation) statement that uses the AVERAGE operator.

Note: You can also set a $COUNTVAR property to specify that Oracle OLAP should count the number of leaf nodes that contributed to an aggregate value when an AGGREGATE function executes. In this case, you do not need to include the COUNTVAR keyword with the AGGREGATE function.

---

intvar

A variable that you have defined with an INTEGER data type. The definition of *intvar* must have exactly the same dimensions in exactly the same order as the dimensions in *var*. When you aggregate several variables together, you must define an INTEGER variable for each one to record the results.

Notes

Steps for Supporting Runtime Calculations

Follow these steps when combining pre-aggregation with runtime aggregation:

1. Create an aggmap that limits the amount of data to be precalculated.
2. Execute the AGGREGATE command with the FUNCDATA argument.
3. When you have made any changes after executing the AGGREGATE command (see "Compiling the Aggmap" on page 7-25), recompile the aggmap with the COMPILE command.
4. Add an $AGGREGATE_FROM property to the data variables (see "Using NA Values to Trigger Runtime Calculations" on page 7-26).
5. UPDATE and COMMIT the analytic workspace.

Compiling the Aggmap

Be sure to compile the aggmap at the time you load data, either with an explicit COMPILE command or with the FUNCDATA argument to the AGGREGATE command. Otherwise, the aggmap will be recompiled at runtime for each session in which the AGGREGATE function is used. Perform other calculations (such as calculating models) before you compile the aggmap.

You need to recompile the aggmap after maintaining any of the dimensions in the aggmap definition or any of the relations that are included in the text of the aggmap.

Runtime Changes to Data Values

When users are able to change data values at runtime, then the data may get out of synchronization. You can prevent this problem in the following ways:

- Use the ALLOCATE command to distribute the data in a new aggregate to the contributing values lower in the hierarchy.

- Do not precalculate the data that is subject to runtime changes, since the stored aggregates cannot be altered to reflect changes made at runtime to the contributing values.

Using NA Values to Trigger Runtime Calculations

By adding an $NATRIGGER property to a variable, you can implicitly call the AGGREGATE function each time the data is queried. The following statements cause `sales` data to be aggregated using the `sales.aggmap` aggmap.

```
CONSIDER sales
PROPERTY '$NATRIGGER' 'AGGREGATE(sales USING sales.aggmap)'
```

A statement such as `REPORT SALES` will now execute the AGGREGATE function, so that computed values are returned instead of NAs.

Using the AGGREGATE Function after Partial Rollups

When your batch window is not sufficiently long to preaggregate all of the data that you want to generate, you can perform the aggregation in stages on consecutive days and use the AGGREGATE function to calculate the balance. For each stage, you must do the following:

1. Change the PRECOMPUTE phrase of the RELATION statement in the aggmap so that new data is aggregated.
2. Execute the AGGREGATE command with the FUNCDATA keyword.
3. Verify that the $NATRIGGER property is set on the variables so that the AGGREGATE function will calculate the balance of the data.

Using Multiple Aggmaps

Whenever possible, you should only use one aggmap to rollup a variable. However, in some situations, a variable requires more than one aggmap to roll up the data in the desired manner. This can create problems when some of the data is calculated on the fly, because the metadata retained for the AGGREGATE function corresponds to the last aggmap. The AGGREGATE function needs metadata that is the union of all of the aggmaps used by the AGGREGATE command. The solution is to create an additional aggmap for use by the AGGREGATE function that correctly identifies the `NA` values. Be sure to compile this aggmap.

You should not use the AGGREGATE function with multiple aggmaps unless you feel comfortable answering the following question:

> When the aggmap is compiled for use by the AGGREGATE function, does the status that results from each PRECOMPUTE clause accurately define the nodes within that dimension at which data has been pre-computed?

When you cannot answer "yes" to this question with confidence, you should not use the AGGREGATE function with multiple aggmaps.

Examples

This section contains several examples of using the AGGREGATE function. For additional aggregation examples, see the examples in AGGMAP.

Example 7–9 Using the AGGREGATE Function as the Formula of an Expression

Example 7–7, "Capstone Aggregation" on page 7-17 illustrates performing the final capstone aggregation using an AGGREGATE command. You could also perform the capstone aggregation at runtime as the expression of a formula.

Assume that your analytic workspace contains the following object definitions.

```
DEFINE GEOG.D DIMENSION TEXT
DEFINE GEOG.PARENTREL RELATION GEOG.D <GEOG.D>
DEFINE TIME.D DIMENSION TEXT
DEFINE TIME.PARENTREL RELATION TIME.D <TIME.D>
DEFINE SALES_JAN76 VARIABLE INTEGER <GEOG.D>
DEFINE SALES_FEB76 VARIABLE INTEGER <GEOG.D>
DEFINE SALES_MAR76 VARIABLE INTEGER <GEOG.D>
DEFINE SALES_CAPSTONE76 VARIABLE INTEGER <GEOG.D TIME.D>
DEFINE CAPSTONE_SOURCE VARIABLE TEXT <TIME.D>
```

Now you create two aggmap objects with the following definitions. Note that in this case the `capstone_aggmap` consists of a RELATION (for aggregation) statement with a `PRECOMPUTE NA` clause.

```
DEFINE LEAF_AGGMAP AGGMAP
AGGMAP
RELATION geog.parentrel OPERATOR SUM
END

DEFINE CAPSTONE_AGGMAP AGGMAP
AGGMAP
RELATION time.parentrel OPERATOR SUM PRECOMPUTE (NA)
END
```

In Example 7–7, "Capstone Aggregation" on page 7-17, the final capstone aggregation is performed using an AGGREGATE command. In this example, the capstone aggregation is defined as a formula named `f_sales_capstone76` that has an AGGREGATE function as the expression of the formula.

```
DEFINE F_SALES_CAPSTONE76 FORMULA INTEGER <GEOG.D TIME.D>
EQ AGGREGATE ( sales_capstone76 USING capstone_aggmap fromvar capstone_source)
```

When you report on the unaggregated variables and formulas in your analytic workspace, you see the following results.

```
GEOG.D          SALES_JAN76    SALES_FEB76    SALES_MAR76
-------------- -------------- -------------- --------------
Boston                  1,000          2,000          3,000
Medford                 2,000          4,000          6,000
San Diego               3,000          6,000          9,000
Sunnydale               4,000          8,000         12,000
Massachusetts              NA             NA             NA
California                 NA             NA             NA
United States              NA             NA             NA

               --------------------F_SALES_CAPSTONE76---------------------
               -------------------------TIME.D--------------------------
GEOG.D            Jan76          Feb76          Mar76            76Q1
-------------- -------------- -------------- -------------- --------------
Boston                  1,000          2,000          3,000          6,000
Medford                 2,000          4,000          6,000         12,000
San Diego               3,000          6,000          9,000         18,000
Sunnydale               4,000          8,000         12,000         24,000
Massachusetts              NA             NA             NA             NA
California                 NA             NA             NA             NA
United States              NA             NA             NA             NA

               ---------------------SALES_CAPSTONE76----------------------
               -------------------------TIME.D--------------------------
GEOG.D            Jan76          Feb76          Mar76            76Q1
-------------- -------------- -------------- -------------- --------------
Boston                  1,000          2,000          3,000             NA
Medford                 2,000          4,000          6,000             NA
San Diego               3,000          6,000          9,000             NA
Sunnydale               4,000          8,000         12,000             NA
Massachusetts              NA             NA             NA             NA
California                 NA             NA             NA             NA
United States              NA             NA             NA             NA
```

Now you aggregate the leaf variables using the following AGGREGATE statement.

```
AGGREGATE sales_jan76 sales_feb76 sales_mar76 USING leaf_aggmap
```

A report of the leaf variables shows that they are aggregated.

```
GEOG.D          SALES_JAN76    SALES_FEB76    SALES_MAR76
-------------- -------------- -------------- --------------
Boston                  1,000          2,000          3,000
Medford                 2,000          4,000          6,000
San Diego               3,000          6,000          9,000
Sunnydale               4,000          8,000         12,000
Massachusetts           3,000          6,000          9,000
California              7,000         14,000         21,000
United States          10,000         20,000         30,000
```

A report of the `f_sales_capstone76` formula shows the aggregated values for `76Q1`.

```
               --------------------F_SALES_CAPSTONE76---------------------
               --------------------------TIME.D---------------------------
GEOG.D             Jan76          Feb76          Mar76           76Q1
-------------- -------------- -------------- -------------- --------------
Boston                  1,000          2,000          3,000          6,000
Medford                 2,000          4,000          6,000         12,000
San Diego               3,000          6,000          9,000         18,000
Sunnydale               4,000          8,000         12,000         24,000
Massachusetts           3,000          6,000          9,000         18,000
California              7,000         14,000         21,000         42,000
United States          10,000         20,000         30,000         60,000
```

While a report of the `sales_capstone76` variable does not show the aggregated values for `76Q1` since they are not stored in the variable.

```
               ---------------------SALES_CAPSTONE76----------------------
               --------------------------TIME.D---------------------------
GEOG.D             Jan76          Feb76          Mar76           76Q1
-------------- -------------- -------------- -------------- --------------
Boston                  1,000          2,000          3,000             NA
Medford                 2,000          4,000          6,000             NA
San Diego               3,000          6,000          9,000             NA
Sunnydale               4,000          8,000         12,000             NA
Massachusetts           3,000          6,000          9,000             NA
California              7,000         14,000         21,000             NA
United States          10,000         20,000         30,000             NA
```

Example 7–10 Aggregating Data on the Fly for a Report

The `units` variable is aggregated entirely on the fly using the `tp.agg` aggmap.

This is the object definitions for the variable `units`.

```
DEFINE units VARIABLE INTEGER <time product>
```

The parent relation for `time` contains these values.

```
      ---TIME.PARENTREL----
      --TIME.HIERARCHIES---
TIME       STANDARD   YTD
---------- ---------- ----------
Jan01      Q1.01      Last.Ytd
Feb01      Q1.01      Last.Ytd
Mar01      Q1.01      Last.Ytd
Q1.01      2001       NA
```

The parent relation for the `product` dimension contains these values.

```
    PRODUCT.PA
PRODUCT    RENTREL
---------- ----------
Food       Na
Snacks     Food
Drinks     Food
Popcorn    Snacks
Cookies    Snacks
Cakes      Snacks
Soda       Drinks
Juice      Drinks
```

In the `units` variable, data is stored only at the lowest level of each dimension hierarchy.

```
  -------------------UNITS-------------------
  -------------------TIME--------------------
PRODUCT     Jan01      Feb01      Mar01      Q1.01
----------- ---------- ---------- ---------- ----------
Food        NA         NA         NA         NA
Snacks      NA         NA         NA         NA
Drinks      NA         NA         NA         NA
Popcorn     2          2          4          NA
Cookies     3          6          3          NA
Cakes       4          4          2          NA
Soda        7          3          9          NA
Juice       1          3          2          NA
```

The aggmap specifies that all data will be calculated on the fly.

```
DEFINE tp.agg AGGMAP
LD <time product> Aggmap
AGGMAP
RELATION time.parentrel PRECOMPUTE (NA)
RELATION product.parentrel PRECOMPUTE (NA)
END
```

The following REPORT command uses the AGGREGATE function to calculate the data.

```
REPORT aggregate(units USING tp.agg)

  -------AGGREGATE(UNITS USING TP.AGG)-------
  -------------------TIME--------------------
PRODUCT     Jan01      Feb01      Mar01      Q1.01
----------- ---------- ---------- ---------- ----------
Food        17         18         20         55
Snacks       9         12          9         30
Drinks       8          6         11         25
Popcorn      2          2          4          8
Cookies      3          6          3         12
Cakes        4          4          2         10
Soda         7          3          9         19
Juice        1          3          2          6
```

Example 7–11 Using $NATRIGGER to Aggregate Data

When the AGGREGATE function is added to `units` in the $NATRIGGER property, a simple REPORT command will display aggregated results.

```
CONSIDER units
PROPERTY '$NATRIGGER' 'AGGREGATE(units USING tp.agg)'
REPORT units

  -------------------UNITS------------------
  -------------------TIME-------------------
PRODUCT     Jan01      Feb01      Mar01      Q1.01
----------- ---------- ---------- ---------- ----------
Food        17         18         20         55
Snacks       9         12          9         30
```

Example 7–12 Calculating all but one Value on the Fly

The AGGREGATE function calculates the complement of the data specified in the PRECOMPUTE clause of the RELATION statement. It returns those values that are currently in status.

For example, when you are using an aggmap that contains this RELATION statement.

```
RELATION letter.letter PRECOMPUTE ('AA')
```

Then the AGGREGATE function calculates all aggregations *except* `AA`, as shown here.

```
REPORT AGGREGATE(units USING letter.aggmap)

                 AGGREGATE(UNITS
LETTER         USING LETTER.AGGMAP)
-------------- --------------------
A                                 3
AA                               NA
AB                                3
AAB                               2
ABA                               1
ABB                               2
AAAA                              1
AABA                              2
ABAA                              1
ABBB                              1
ABBA                              1
...
```

AGGREGATION

The AGGREGATION function is available only within a model. Its purpose is to allow you to create a model that represents a custom aggregate. Such an aggmap can be used for dynamic aggregation with the AGGREGATE function.

> **Note:** Because the AGGREGATION function is intended only for dynamic aggregation, a model that contains such a function can*not* be used with the AGGREGATE command.

Syntax

AGGREGATION(*dimval-list*)

Arguments

dimval-list

A list of one or more dimension values to include in the custom aggregation. The specified values must belong to the same dimension to which the target dimension value belongs. You must specify each dimension value as a text literal. That is, they cannot be represented by a text expression such as a variable.

Examples

Example 7–13 Using the AGGREGATION Function to Create a Custom Aggregate

The following lines of code from a program perform these steps:

1. Add the new dimension value `my_time` to the `time` dimension.

   ```
   MAINTAIN time ADD 'My_Time'
   ```

2. Define the model `mytime_custagg` and set the specification of the model using the AGGREGATION function.

   ```
   DEFINE mytime_custagg MODEL
   MODEL JOINLINES('DIMENSION time' 'My_Time = AGGREGATION(\'23\' \'24\')')
   ```

   (Note that backslash escape characters are required to include quotation marks within a quoted string.)

3. Define the `sales_aggmap` aggmap.

```
DEFINE sales_aggmap AGGMAP <time cpc <customer product channel> >
AGGMAP
RELATION prntrel.time
RELATION prntrel.chan
RELATION prntrel.prod
RELATION prntrel.cust
END
```

4. Add the model `mytime_custagg` to `sales_aggmap`.

```
AGGMAP ADD mytime_custagg TO sales_aggmap
```

5. Limit the dimensions to the values of interest and run a report.

```
" Run a report
LIMIT time TO 'My_Time' '23' '24'
LIMIT channel TO '5'
LIMIT product TO '70'
LIMIT customer TO '114'
REPORT DOWN time AGGREGATE(sales USING sales_aggmap)
```

The report generates the following output.

```
CHANNEL: 5
PRODUCT: 70
               --AGGREGATE(SALES---
               USING SALES_AGGMAP)-
               ------CUSTOMER------
TIME                   114
-------------- --------------------
my_time                  682,904.34
23                        84,982.92
24                       597,921.42
```

ALLCOMPILE

The ALLCOMPILE program compiles every compilable object in your current analytic workspace, one at a time. As it works, ALLCOMPILE sends to the current outfile messages that show the name of the object being compiled.

Syntax

ALLCOMPILE [*n*]

Arguments

n

An INTEGER expression with a value of zero or higher. The expression specifies the number of objects to be compiled before an UPDATE command is executed. For example, when you specify `1`, an UPDATE command is executed after each object is compiled. When you specify `0` (zero), all the objects are compiled and an UPDATE command is executed only at the end. When you omit the argument, no UPDATE command is executed by ALLCOMPILE. Frequent updates during an ALLCOMPILE help ensure the most efficient use of space in the analytic workspace.

Notes

Error Messages

ALLCOMPILE uses the COMPILE command. This means that it will check for syntax errors as it compiles an object, and it will record error messages in the current outfile as appropriate.

Examples

Example 7–14 ALLCOMPILE Output

The following example shows the output of ALLCOMPILE when it is run on an analytic workspace that contains four programs.

```
Compiling AUTOGO
Compiling READIT
Compiling REGION.REPORT
Compiling SALES.REPORT
```

ALLOCATE

The ALLOCATE command calculates lower-level data from upper-level data by allocating variable data down a hierarchical dimension. Frequently you allocate data for budgeting, forecasting, and profitability analysis.

Syntax

ALLOCATE *source* [SOURCE *conjoint*] [BASIS *basisname* [ACROSS *dimname*]] -

[TARGET *targetname* [TARGETLOG *targetlogname*]] -

[USING *aggmap*] [ERRORLOG *errorlogfileunit*]

Arguments

source
A variable or formula that provides the values to allocate. When the *source* object is a formula, you must also specify a variable with the TARGET keyword. When you specify a variable as *source* and you do not specify a target variable or a *basisname* variable, then ALLOCATE uses *source* as the basis and the target.

SOURCE *conjoint*
Specifies a conjoint dimension that contains a list of cells the user has changed. The ALLOCATE command uses this list to produce the smallest target status needed to allocate all of the changed source cells.

BASIS *basisname*
Specifies a variable, relation, or formula that provides the data on which the allocation is based. That data determines which cells of the target receive allocated values and, in an even or proportional operation, the amount of the source allocated to a target cell.

When the OPERATOR specified by a RELATION (for allocation) statement in *aggmap* is a COPY operator (COPY, MIN, MAX, FIRST, LAST), the basis tells the ALLOCATE command which target cells to update. When the OPERATOR specified is EVEN, then ALLOCATE derives the counts that it uses for allocation from the basis. When the OPERATOR specified is the PROPORTIONAL, then ALLOCATE uses the basis data to determine the amount to allocate to each target cell. When the OPERATOR is HCOPY, HFIRST, HLAST, or HEVEN, then ALLOCATE does not use a BASIS object. Instead, it allocates the source data to all of the target cells in the dimension hierarchy that is specified by the relation named in the RELATION command.

When you specify the same variable as both the basis and the target, the current values of the target cells determine the allocation. When you do not specify a basis, then the ALLOCATE command uses the source as the basis.

ACROSS *dimname*
Specifies a dimension, which can be a named composite, that the ALLOCATE command loops over to discover the cells in a basis. Because a basis can be a formula, you can realize a significant performance advantage by supplying a looping dimension that eliminates the sparsity from the basis loop.

TARGET *targetname*
Specifies a variable to hold the allocated values. When the source object is a formula, then you must specify a target. When the source object is a variable and you do not specify a target, then ALLOCATE uses the source variable as the target.

TARGETLOG *targetlogname*
Specifies a variable (identically dimensioned to the *targetname* variable), or a relation that specifies such a variable, to which ALLOCATE assigns a copy of the allocation. For instance, when ALLOCATE assigns the value of `100` to the cell of the `costs` variable that is specified by the `time` and `product` dimension values `Jan01` and `TV`, and the `targetlog` relation specifies the cell of the `costacct` variable that is specified by the same dimension values, then ALLOCATE assigns the value of `100` to the specified `costacct` cell, also.

USING *aggmap*
Specifies the name of a previously-defined aggmap to use for the allocation. When you do not include this phrase, the command uses the default allocation specification for the variable as previously specified using the $ALLOCMAP property.

ERRORLOG *errorlogfileunit*
Specifies a file unit that ALLOCATE uses for logging allocation deadlocks, errors, or other information. When the allocation does not generate any deadlocks or errors, ALLOCATE sets *errorlogname* to `NA`. When the allocation produces one or more deadlocks or errors, the events are sent to the specified file. ALLOCATE writes one line in the file for each allocation source that remains unallocated.

When you do not specify a file unit with ERRORLOG, ALLOCATE sends the information to the standard output device.

Notes

Preserving Original Basis Values

Often the source, basis, and target objects are the same variable and therefore the original values in the cells of the target variable determine the proportions of the allocation. The allocation overwrites those original values in the target cells with the allocated values. To preserve original values in a variable, specify the original variable as the basis object and save the allocated values to a new variable as the target object. Using different basis and target objects makes it possible for you to preview the allocated data. When you then want to store the allocated values in the same variable as the basis, you can perform the allocation again with the same object as the basis and the target. Another example of using different basis and target objects is using an actuals variable as the basis of the allocation and a budget variable as the target.

Using a Formula as a Source or Basis

Source and basis objects can be formulas, which makes it possible for you to make complex computations and have the results be the source or basis object. For example, when you want to see the sales of individual products that would be necessary to produce a thirty percent increase in sales for the next year, you could express the increase in the following formula.

```
DEFINE actualsWanted DECIMAL FORMULA <time, product>
EQ LAG(actuals, 1, time) * 1.3
```

You would then use ACTUALSWANTED as the *source* object with the ALLOCATE command. In this example, you would use the ACTUALS variable as the basis.

Tracking Multiple Allocations

When you specify a variable with the TARGETLOG argument, you can store an allocated value in that variable as well as in the `target` variable. This double entry allocation makes it possible for you to track multiple allocations to the same target cell. For example, when you allocate a series of different costs to the same costs centers, then each allocation increases the values in the target cells. You can keep track of the individual allocations by specifying a different *targetlogname* variable for each allocation.

Logging Allocation Errors

When you specify a file with the ERRORLOG argument, you can record errors that result from locks and `NA` basis values. The log can provide feedback to an application about which source values remain unallocated. You can use the information to modify the allocation, for example by using a hierarchical operator such as HEVEN in a RELATION command in the aggmap. You can use the ALLOCERRLOGHEADER and ALLOCERRLOGFORMAT options to format the error log. Within an allocation specification, you can specify other aspects of the error log using the ERRORLOG and ERRORMASK statements.

Logging the Progress of an Allocation

When you specify a file with the POUTFILEUNIT option, then you can record and monitor the progress of an allocation. You can use the file to get feedback during the course of a lengthy allocation and to gain information that might be useful for optimizing the allocation in the future.

Examples

Example 7–15 Direct Even Allocation

This example allocates a value specified at one level of the `time` dimension hierarchy directly to the variable target cells that are specified by lower level values in the hierarchy without allocating values to an intermediate level. The `timemonthyear` relation specifies the hierarchical relationship of the `time` values. The source, basis, and target of the allocation are all the same variable, PROJBUDGET, which is dimensioned by `division`, `time`, and `line`. The `time` dimension is a nonunqiue concat dimension that has as its base dimensions `year`, `quarter`, and `month`. The `time` dimension is limited to `<year: Yr02>`, `<quarter: Q1.02>`, `<quarter: q1.02>`, and `<month: Jan02>` to `<month: Jun02>`. The following statements define the `projbudget` variable, set the value of a cell in to `6000` and then report the variable.

```
DEFINE projbudget VARIABLE DECIMAL <division time line>
projbudget(division 'CAMPING' time '<YEAR: YR02>' line  'MARKETING') = 6000
REPORT projbudget
```

The preceding statement produces the following results.

```
LINE: MARKETING
                 -PROJBUDGET--
                 --DIVISION---
TIME                CAMPING
---------------- -------------
<year: Yr02>          6,000.00
<quarter: Q1.02>            NA
<quarter: Q2.02>            NA
<month: Jan02>              NA
<month: Feb02>              NA
<month: Mar02>              NA
<month: Apr02>              NA
<month: May02>              NA
<month: Jun02>              NA
```

The following statements define a self-relation on the `time` dimension, relate the `month` values directly to the `year` values, and report the values of the relation.

```
DEFINE timemonthyear RELATION time <time>
LIMIT month TO 'JAN02' TO 'JUN02'
timemonthyear(time month) = '<YEAR: YR02>'
REPORT timemonthyear
```

The preceding statement produces the following results.

```
TIME             TIMEMONTHYEAR
---------------- -------------
<year: Yr02>     NA
<quarter: Q1.02> NA
<quarter: Q2.02> NA
<month: Jan02>   <year: Yr02>
<month: Feb02>   <year: Yr02>
<month: Mar02>   <year: Yr02>
<month: Apr02>   <year: Yr02>
<month: May02>   <year: Yr02>
<month: Jun02>   <year: Yr02>
```

The following statements define an aggmap and enter commands into the allocation specification. They allocate the value that is specified by `<year: Yr02>` from `projbudget` to the cells of the same variable that are specified by the `month` dimension values, and then report `projbudget`. The target cells of the variable have `NA` values so the RELATION command in the allocation specification specifies the HEVEN operator. The ALLOCATE command specifies only one variable, projbudget, so that variable is the source and target of the allocation. No basis object is required because the allocation is an HEVEN operation. The allocation is directly from the year source value to the month target values because that is the hierarchy specified by the relation in the allocation specification.

```
DEFINE projbudgmap AGGMAP
ALLOCMAP
RELATION timemonthyear OPERATOR HEVEN
END
ALLOCATE projbudget USING projbudgmap
REPORT projbudget
```

The preceding statement produces the following results.

```
LINE: MARKETING
                 -PROJBUDGET--
                 --DIVISION---
TIME                CAMPING
---------------- -------------
<YEAR: YR02>          6,000.00
<QUARTER: Q1.02>            NA
<QUARTER: Q2.02>            NA
<MONTH: JAN02>        1,000.00
...
<MONTH: JUN02>        1,000.00
```

Example 7–16   Recursive Even Allocation with a Lock

This example allocates a value specified at one level of the time dimension hierarchy first to the target cells at an intermediate level in a variable and then to the cells that are specified by the lowest level values in the hierarchy. The timeparent relation specifies the hierarchical relationship of the time values. The source, basis, and target of the allocation are projbudget. The status of the division, time, and line dimensions are the same as the direct allocation example. At the beginning of this example, the `projbudget` variable again has just the single value, `6000`, in the cell specified by `<year: Yr02>`.

```
DEFINE timeparent RELATION time <time>
LIMIT quarter TO 'Q1.02' 'Q2.02'
timeparent(time quarter) = '<YEAR: YR02>'
LIMIT month TO 'JAN02' TO 'MAR02'
timeparent(time month) = '<QUARTER: Q1.02>'
LIMIT month TO 'APR02' TO 'JUN02'
timeparent(time month) = '<QUARTER: Q1.02>'
REPORT timeparent
```

The preceding statement produces the following results.

```
TIME             TIMEPARENT
---------------- -------------
<year: Yr02>     NA
<quarter: Q1.02> <year: Yr02>
<quarter: Q2.02> <year: Yr02>
<month: Jan02>   <quarter: Q1.02>
<month: Feb02>   <quarter: Q1.02>
<month: Mar02>   <quarter: Q1.02>
<month: Apr02>   <quarter: Q2.02>
<month: May02>   <quarter: Q2.02>
<month: Jun02>   <quarter: Q2.02>
```

This example demonstrates locking a cell so that it does not participate in the allocation. Locking a cell requires a valueset, so the following statements define one, limit the `time` dimension to the desired value, assign a value to the valueset, and then reset the status of the `time` dimension.

```
DEFINE timeval TO '<QUARTER: Q2.02>'
LIMIT time TO '<Year: YR02>' '<Quarter: Q1.02>'  '<Quarter: Q2.02>' -
   '<month: Jan02>' '<month: Feb02>' '<month: Mar02>' -
   '<month: Apr02>' '<month: May02>' '<month: Jun02>
```

The following statements revise the specification of the aggmap named `projbudgmap`. This time the RELATION command in the allocation specification specifies the `timeparent` relation, the HEVEN operator, and the PROTECT argument. The READWRITE keyword specifies that the children of the locked cell also do not participate in the allocation. The NONORMALIZE keyword specifies that the value of the locked cell is not subtracted from the source value before it is allocated to the target cells. The statements then allocate the source value and report the results.

```
CONSIDER projbudgmap
ALLOCMAP
RELATION timeparent OPERATOR HEVEN ARGS PROTECT NONORMALIZE READWRITE timeval
END

ALLOCATE projbudget USING projbudgmap
REPORT projbudget
```

The preceding statement produces the following results.

```
LINE: MARKETING
                 -PROJBUDGET--
                 --DIVISION---
TIME                CAMPING
---------------- -------------
<year: Yr02>          6,000.00
<quarter: Q1.02>      6,000.00
<quarter: Q2.02>            NA
<month: Jan02>        2,000.00
<month: Feb02>        2,000.00
<month: Mar02>        2,000.00
<month: Apr02>              NA
<month: May02>              NA
<month: Jun02>              NA
```

Example 7–17 Recursive Proportional Allocation

This example uses the same relation as the recursive even allocation but it uses the PROPORTIONAL operator and it does not lock any cells. Because a proportional allocation uses the values of the basis object to calculate the values to assign to the target cells, the `projbudget` variable has values assigned to each of its cells. The value of the `<year: Yr02>` cell is `6000.`, which was assigned to that cell. It is not the value an aggregation of the lower levels. A report of `projbudget` before the allocation produces the following results.

```
LINE: MARKETING
                 -PROJBUDGET--
                 --DIVISION---
TIME                CAMPING
---------------- -------------
<year: Yr02>          6,000.00
<quarter: Q1.02>      1,000.00
<quarter: Q2.02>      2,000.00
<month: Jan02>          300.00
<month: Feb02>          100.00
<month: Mar02>          600.00
<month: Apr02>          400.00
<month: May02>          800.00
<month: Jun02>          800.00
```

The following statements replace the previous specification of the aggmap with the new RELATION command, which specifies the PROPORTIONAL operator. The allocation specification includes a `SOURCEVAL ZERO` statement, which specifies that the source value is replace with a zero value after the allocation (see SOURCEVAL for more information). The statements then allocate the source value and report the result.

```
CONSIDER projbudgmap
ALLOCMAP JOINLINES('RELATION timeparent OPERATOR PROPORTIONAL timeval' -
  'SOURCEVAL ZERO' -
  'END')
ALLOCATE projbudget USING projbudgmap
REPORT projbudget
```

The preceding statement produces the following results.

```
TIME             TIMEPARENT
LINE: MARKETING
                 -PROJBUDGET--
                 --DIVISION---
TIME                CAMPING
---------------- -------------
<year: Yr02>                 0
<quarter: Q1.02>      2,000.00
<quarter: Q2.02>      4,000.00
<month: Jan02>          600.00
<month: Feb02>          200.00
<month: Mar02>        1,200.00
<month: Apr02>          800.00
<month: May02>        1,600.00
<month: Jun02>        1,600.00
```

ALLOCERRLOGFORMAT

The ALLOCERRLOGFORMAT option determines the contents and the formatting of the error log that you specify with the ERRORLOG argument to the ALLOCATE command. You can specify a header for the error log with the ALLOCERRLOGHEADER option.

Syntax

ALLOCERRLOGFORMAT = *text*

Arguments

text
Characters that determine the contents and formatting of the error log that you specify with the VNF command. Table 7–3, " Characters for Specify the Contents of the Error Log for ALLOCATE" lists the characters that specify the contents of the error log.

Table 7–3 Characters for Specify the Contents of the Error Log for ALLOCATE

Character	Output Specified
b	The basis object being processed.
c	The child node of the dimension being processed.
d	The name of the dimension being processed.
e	A description of the error encountered.
n	The error code of the error encountered.
p	The parent node of the dimension being processed.
r	The name of the relation being allocated down.
s	The source object being processed.
t	The target object being processed.
n	The basis value of the child cell receiving the allocation.
y	The source value of the parent cell being allocated.
z	The basis value of the parent cell being allocated.

Notes

Specifying the Number of Characters for an Object

By placing an INTEGER value before the formatting character, you can specify the number of characters that the object occupies in the error log. The default value of ALLOCERRLOGFORMAT is the following.

```
'%8p %8y %8z %e (%n)'
```

Specifying Escape Sequences as Formatting Characters

You can specify escape sequences as formatting characters. For valid escape sequences, see "Escape Sequences" on page 2-4.

Specifying How Many Error Conditions to Log

The ERRORLOG command in an allocation specification specifies how many allocation error conditions to log and whether to continue or to stop the allocation when the specified maximum number of errors have been logged.

Examples

Example 7–18 Setting the ALLOCERRLOGFORMAT Option

This example sets the ALLOCERRLOGFORMAT option.

```
ALLOCERRLOGFORMAT = '%8p %8y %8z %e (%n)'
SHOW ALLOCERRLOGFORMAT
```

The preceding statement produces the following results.

```
%8p %8y %8z %e (%n)
```

ALLOCERRLOGHEADER

The ALLOCERRLOGHEADER option determines the column headings for the error log that you specify with the ERRORLOG argument to the ALLOCATE command. To specify additional formatting for the error log, use the ALLOCERRLOGFORMAT option.

Syntax

ALLOCERRLOGHEADER = *text*

Arguments

text
Characters that determine the content and formatting of the column headers that are the first line of the error log that you specify with the ALLOCATE command. (See ALLOCERRLOGFORMAT for a list of the characters you can use.)

When you specify `NA` as the value for this option, then ALLOCATE does not write any header to the error log. The following is the default value of ALLOCERRLOGHEADER.

```
'Dim      Source   Basis\n%-8d %-8v %-8b Description\n
-------- -------- -------- -----------'
```

Notes

Specifying How Many Error Conditions to Log

The ERRORLOG command in an ALLOCMAP type aggmap specifies how many allocation error conditions to log and whether to continue or to stop the allocation when the specified maximum number of errors have been logged.

Examples

Example 7–19 Setting the ALLOCERRLOGHEADER Option

The following statements define the heading for the error log specified by an ALLOCATE command and show the value of the ALLOCERRLOGHEADER option.

```
ALLOCERRLOGHEADER = 'Dim      Source   Basis\n %-8d %-8v %-8b Description \n
-------- -------- -------- -----------'
SHOW ALLOCERRLOGHEADER
```

The preceding statement produces the following results.

```
Dim      Source   Basis
%-8d %-8s %-8b Description
-------- -------- -------- -----------
```

An allocation operation that has a variable named `budget` as both the source and basis objects and which encounters a deadlock when allocating down the `division` dimension produces the following entry in the error log.

```
Dim      Source   Basis
Division Budget   Budget  Description
-------- -------- -------- -----------
Accdiv   650000    NA      A deadlock occurred allocating data (5)
```

ALLOCMAP

The ALLOCMAP command identifies an aggmap object as an allocation specification and enters the contents of the specification.

Syntax

ALLOCMAP [*specification*]

Arguments

specification

A multiline text expression that is the allocation specification for the current aggmap object. An allocation specification begins with an ALLOCMAP statement and ends with an `END` statement. Between these statements, you code one or more of the following statements depending on the calculation that you want to specify:

CHILDLOCK
DEADLOCK
DIMENSION (for allocation)
ERRORLOG
ERRORMASK
MEASUREDIM (for allocation)
RELATION (for allocation)
SOURCEVAL
VALUESET

Each statement is a line of the multiline text expression. Separate statements with newline delimiters (`\n`), or use JOINLINES.

For a discussion of how to determine which statements to include, see "Designing an Allocation Specification" on page 50.

Notes

Designing an Allocation Specification

Minimally, an allocation specification consists of a RELATION (for allocation) statement or a VALUESET statement However, you can create more complex allocation specifications and change the default settings for error handling by including additional OLAP DML statements in the specification, as follows:

1. For hierarchical allocations, a RELATION (for allocation) statement that specifies a self-relation that identifies the child-parent relationships of the hierarchy. List the statements in the order in which you want to perform the various operations; or if this is not important, list the RELATION statements in the same order as the dimensions appear in the variable definition.
2. For non-hierarchical allocations, a VALUESET statement that specifies the values to be used when allocating.
3. A CHILDLOCK statement that tells the ALLOCATE command whether to determine if RELATION statements in the aggmap specify lock on both a parent and a child element of a dimension hierarchy.
4. A DEADLOCK statement that tells the ALLOCATE command whether to continue an allocation when it encounters a deadlock, which occurs when the allocation cannot distribute a value because the targeted cell is locked or, for some operations, has a basis value of `NA`.
5. When a dimension is not shared by the target variable and the source or the basis objects, a DIMENSION (for allocation) statement that specifies a single value to set as the status of that dimension.
6. An ERRORLOG statement that specifies how many errors to allow in the error log specified by the ALLOCATE command and whether to continue the allocation when the maximum number of errors has occurred.
7. An ERRORMASK statement that specifies which error conditions to exclude from the error log.
8. When the source data comes from a variable, a SOURCEVAL statement that specifies whether ALLOCATE changes the source data value after the allocation.

Aggmap Type

You can use the AGGMAPINFO function to learn the type of an aggmap. An aggmap into which you have entered an allocation specification using the ALLOCMAP has the type ALLOCMAP and an aggmap into which you have entered an aggregation specification using the AGGMAP command has the type AGGMAP. When you have defined an aggmap but have not yet entered a specification in it, its type is `NA`.

Allocation Options

A number of options effect allocation. These options are listed in Table 7–4, " Allocation Options" on page 7-52.

Table 7–4 Allocation Options

Statement	Description
ALLOCERRLOGFORMAT	An option that determines the contents and the formatting of the error log that you specify with the ERRORLOG argument to the ALLOCATE command.
ALLOCERRLOGHEADER	An option that determines the column headings for the error log that you specify with the ERRORLOG argument to the ALLOCATE command.
POUTFILEUNIT	An option that identifies a destination for status information about an allocation or aggregation operation. A file unit uniquely identifies the destination file.

One RELATION for Each Dimension

An aggmap can have only one RELATION statement for any given dimension.

One Hierarchy For Each Dimension

An allocation operation proceeds down only one hierarchy in a dimension. When a dimension has more than one hierarchy, then you must limit the dimension to one of the hierarchies with a qualified data reference after the *rel-name* argument.

Examples

Example 7–20 Allocation Specification from an Input File

In this example an aggmap and its specification are defined in an ASCII disk file called `salesalloc.txt`. The statements in the file are then executed in the analytic workspace through the use of the INFILE statement. The statements in `salesalloc.txt` are the following.

```
IF NOT EXISTS ('salesalloc')
  THEN DEFINE salesalloc AGGMAP
  ELSE CONSIDER salesalloc
ALLOCMAP
  RELATION time.parent OPERATOR EVEN
  RELATION product.parent OPERATOR EVEN
  RELATION geography.parent OPERATOR EVEN
  SOURCEVAL ZERO
  DEADLOCK SKIP
END
```

To include the `salesalloc` aggmap in your analytic workspace, execute the following statement.

```
INFILE 'salesalloc.txt'
```

The `sales.agg` aggmap has now been defined and contains three RELATION (for allocation) statements and the SOURCEVAL and DEADLOCK statements. In this example, the ALLOCATE statement allocates its source value evenly to all of the aggregate level cells and the detail level cells of the target variable because the relations `time.parent`, `product.parent`, and `geography.parent` relate each child dimension value to its parent in the dimension hierarchy. The DEADLOCK statement tells the ALLOCATE statement to log an error and continue the allocation when a branch of a target hierarchy is locked or has a value of `NA`. The SOURCEVAL statement tells ALLOCATE to assign a zero value to the source cells after allocating the source data.

You can now use the `salesalloc` aggmap with an ALLOCATE statement, such as.

```
ALLOCATE sales USING salesalloc
```

Example 7–21 Allocation Specification from a Text Expression

In this example the `salesalloc` aggmap has already been defined. The specification is added to the aggmap as a text expression argument to the ALLOCMAP statement.

```
CONSIDER salesalloc
ALLOCMAP
RELATION time.parent OPERATOR EVEN
RELATION product.parent OPERATOR EVEN
RELATION geography.parent OPERATOR EVEN
SOURCEVAL ZERO
DEADLOCK SKIP
```

Example 7–22 Specifying a Single Dimension Value in an Allocation Specification

This example proportionally allocates a value it calculates from the `sales` variable to cells in a `projectedsales` variable. The `sales` variable is dimensioned by the `time`, `product`, `customer`, and `channel` dimensions.

The example defines the `projectedsales` variable to use as the target of the allocation and the `increasefactor` formula to use as the source. The formula multiplies values from `sales` by ten percent. The example limits the `time` dimension and creates the `ytoq.rel` relation, which relates the year 2001 to the quarters of 2002. The next LIMIT commands limit the dimensions shared by `sales` and `projectedsales`.

The example creates an aggmap and uses the ALLOCMAP statement to enter a RELATION (for allocation) and a DIMENSION statement into the map. The RELATION statement specifies the `ytoq.rel` relation as the dimension hierarchy to use for the allocation and specifies that the allocation is proportional. The DIMENSION statement tells ALLOCATE to set the status of the `channel` dimension to `totalchannel` for the duration of the allocation.

```
DEFINE projectedSales DECIMAL VARIABLE <time, SPARSE <product, customer>>
DEFINE increaseFactor DECIMAL FORMULA <product>
EQ sales * 1.1
LIMIT time TO '2001' 'Q1.02' TO 'Q4.02'
DEFINE YtoQ.rel RELATION time <time>
LIMIT time TO 'Q1.02' to 'Q4.02'
YtoQ.rel = '2001'
LIMIT time TO '2001' 'Q1.02' to 'Q4.02'
LIMIT product TO 'TotalProduct' 'Videodiv' 'Audiodiv' 'Accdiv'
LIMIT customers TO 'TotalCustomer'
DEFINE time.alloc AGGMAP
ALLOCMAP
RELATION YtoQ.rel OPERATOR PROPORTIONAL
DIMENSION channel 'TotalChannel'
END
ALLOCATE increaseFactor BASIS sales TARGET projectedSales USING time.alloc
```

The `sales` values that are the basis of the allocation are the following.

```
CHANNEL: TOTALCHANNEL
CUSTOMERS: TOTALCUSTOMER
                 ---------------PROJECTEDSALES---------------
                 --------------------TIME--------------------
PRODUCT           2001      Q1.02      Q2.02      Q3.02      Q4.02
------------     ------     ------     ------     ------     ------
TotalProduct      7000       1000       2000       3000       1000
Videodiv          4100        600       1100       1900        500
Audiodiv          1700        200        600        600        300
Accdiv            1200        200        300        500        200
```

The following shows a report of `projectedsales` for `totalchannel` after the allocation.

```
CHANNEL: TOTALCHANNEL
CUSTOMERS: TOTALCUSTOMER
                 ---------------PROJECTEDSALES---------------
                 --------------------TIME--------------------
PRODUCT           2001      Q1.02      Q2.02      Q3.02      Q4.02
------------     ------     ------     ------     ------     ------
TotalProduct        NA         NA         NA         NA         NA
Videodiv            NA        660       1210       2090        550
Audiodiv            NA        220        660        660        330
Accdiv              NA        220        330        550        220
```

Example 7–23 Entering RELATION Statements in an Allocation Specification

This example defines a `time.type` dimension and adds to it the two hierarchies of the `time` dimension. It defines the `time.time` relation that relates the hierarchy types (that is, `time.type`) to the `time` dimension. The example defines the `time.alloc` aggmap. With the ALLOCMAP command, it enters a RELATION statement in the aggmap. The RELATION statement specifies the values of the `time` dimension hierarchy to use in the allocation, limits the `time` dimension to one hierarchy with the QDR, and the specifies the EVEN operation for the allocation. The ALLOCATE command then allocates data from the source object to the target variable using the `time.alloc` aggmap. In the ALLOCATE command the source, basis, and target objects are the same `sales` variable.

```
DEFINE time.type TEXT DIMENSION
MAINTAIN time.type add 'Fiscal'
MAINTAIN time.type add 'Calendar'
DEFINE time.time RELATION time <time, time.type>
DEFINE time.alloc AGGMAP

ALLOCMAP
RELATION time.time (time.type 'Fiscal') OPERATOR EVEN
END

ALLOCATE sales USING time.alloc
```

CHILDLOCK

Within an allocation specification, a CHILDLOCK statement tells the ALLOCATE statement to determine if RELATION (for allocation) statements in the allocation specification have specified locks on both a parent and on a child of the parent in a dimension hierarchy. Locking both a parent and one of its children can cause incorrect allocation results.

Syntax

```
CHILDLOCK [DETECT|NODETECT]
```

Arguments

DETECT
Tells the ALLOCATE statement to detect that an allocation lock exists on a parent and also on one of its children in a dimension hierarchy. When it detects a locked parent and child, the ALLOCATE statement creates an entry in the error log for the allocation.

NODETECT
Tells the ALLOCATE statement to continue an allocation even when a lock exists on a parent and also on one of its children in a hierarchy. (Default)

DEADLOCK

Within an allocation specification, a DEADLOCK statement tells the ALLOCATE statement what to do when it cannot distribute a source value to a target cell specified by a value in a dimension hierarchy because the target cell is either locked by a RELATION (for allocation) statement in the allocation specification or the cell has a basis value of `NA`.

Syntax

DEADLOCK [SKIP|NOSKIP]

Arguments

SKIP
Tells the ALLOCATE statement to log the error and continue with the allocation even though it cannot distribute source values to cells specified by a branch of a dimension hierarchy because a target cell is locked or the basis value of the cell is `NA`.

NOSKIP
Tells the ALLOCATE statement to stop the allocation and to return an error when it cannot distribute source values to cells in a branch of a dimension hierarchy because a target cell is locked or the basis value is `NA`. This is the default action when you do not include a DEADLOCK statement in the aggmap used by the ALLOCATE command.

DIMENSION (for allocation)

Within an allocation specification, a DIMENSION statement sets the status to a single value of a dimension. Within an allocation specification this dimension is a dimension that the source, basis, and target objects do not have in common. When an allocation specification does not specify such single values with DIMENSION statements, Oracle OLAP uses the current status values of the dimensions when performing the allocation.

You use a DIMENSION statement to ensure that the status of a dimension is set to the value that you want it to have for the allocation. You must use a separate DIMENSION statement for each dimension that is not shared by the source, basis, and target objects.

Syntax

DIMENSION *dimension* '*dimval*'

Arguments

dimension
the name of the dimension that you want to limit.

d*imval*
The single value of the dimension to which you want the status of the dimension set for the duration of an allocation.

Examples

For an example of using a DIMENSION statement in an allocation specification, see Example 7–22, "Specifying a Single Dimension Value in an Allocation Specification" on page 7-53.

ERRORLOG

Within an allocation specification, an ERRORLOG statement specifies how many allocation error conditions to log and whether to continue or to stop the allocation when the specified maximum number of errors have been logged. You specify the error log with the ERRORLOG keyword to the ALLOCATE command.

Syntax

```
ERRORLOG [UNLIMITED|MAX <num>] [STOP|NOSTOP]
```

Arguments

UNLIMITED
Tells the ALLOCATE command to write an unlimited number of errors to the error log. This is the default setting.

MAX *num*
Specifies a maximum number of errors that ALLOCATE can write to the error log.

STOP
NOSTOP
Specifies whether to stop the allocation when ALLOCATE has written the maximum number of errors to the error log. When you specify STOP, the allocation stops. When you specify NOSTOP, the allocation continues but ALLOCATE does not write any more errors to the error log. When you have specified UNLIMITED, then the STOP and NOSTOP arguments have no effect and the allocation continues no matter how many errors occur.

Notes

Formatting the Error Log

The ALLOCERRLOGFORMAT option determines the contents and the formatting of the error log that you specify with the ERRORLOG argument to the ALLOCATE command. You can specify a header for the error log with the ALLOCERRLOGHEADER option.

ERRORMASK

Within an allocation specification, an ERRORMASK statement specifies the error conditions that you do not want to appear in the allocation error log. You specify the error log with the ERRORLOG keyword to the ALLOCATE command.

Syntax

ERRORMASK <*num...*>

Arguments

num...

The number of the error that you do not want to appear in the error log. For example, to exclude CHILDLOCK error, you would enter the following statement in the aggmap.

```
ERRORMASK 10
```

To exclude all errors, you would enter the following statement in the aggmap.

```
ERRORMASK 1 2 3 4 5 6 7 8 9 10
```

MEASUREDIM (for allocation)

Within an allocation specification, a MEASUREDIM statement identifies the name of a measure dimension that is specified in the definition of an operator variable or an argument variable. However, you cannot specify a measure dimension when it is included in the definition of the aggmap object.

Syntax

MEASUREDIM *name*

Arguments

name
The name of the measure dimension. A measure dimension is a dimension that you define. The dimension values are names of existing variables.

Notes

See MEASUREDIM (for aggregation)

RELATION (for allocation)

Within an allocation specification, a RELATION statement identifies a relation that specifies the path through a dimension hierarchy and the method of the allocation.

To allocate a source data down a hierarchy of a dimension, you must specify with a RELATION statement the values of the hierarchy that identify the cells of the variable that are the targets of the allocation. When the target of the allocation is a multidimensional variable, then you must include a separate RELATION statement for each dimension down which you want to allocate the source data. The order of the RELATION statements in an aggmap determines the order of the allocation. The allocation proceeds down the dimension hierarchy in the first RELATION statement, then down the second, and so on.

Oracle OLAP can perform allocations on only one hierarchy in a dimension in one execution of the ALLOCATE command. When a dimension has more than one hierarchy, then you must supply a *qdr* argument to limit the relation to only one hierarchy.

Note: Keep the following restrictions in mind:

- An allocation specification must include either a RELATION statement or a VALUESET statement.
- Only one RELATION statement or VALUESET statement may be used for each dimension in the allocation specification.

Syntax

RELATION *rel-name* [(*qdr*. . .)] OPERATOR {*operator*|} -

[NAOPERATOR *operator*] [REMOPERATOR *operator*] -

[PARENTALIAS *dimension-alias-name*] -

[ARGS {[FLOOR *floorval*] [CEILING *ceilval*] [MIN *minval*] [MAX *maxval]* -

[NAHANDLE {<u>IGNORE</u>|CONSIDER|PREFER}] -

[ADD|<u>ASSIGN</u>] [PROTECT [NONORMALIZE] [<u>READWRITE</u>|WRITE] *lockvalueset*] -

[WEIGHTBY [ADD|<u>MULTIPLY</u>] [WNAFILL *nafillval*] *weightobj*]}]

Arguments

rel-name

An Oracle OLAP self-relation that specifies the values of a dimension hierarchy that identify the path of allocation. The cells in the target variable identified by the values in *rel-name* receive the allocated data.

qdr. . .

One or more qualified data references that specify a single dimension value for each dimension of the relation that is not part of the self-relation. When the self-relation has more than one hierarchy, you must provide a *qdr* for the hierarchy dimension of the self-relation dimension that limits to single values any hierarchies not involved in the allocation.

OPERATOR *operator*

The *operator* after the OPERATOR keyword indicates one of the operator types described in Table 7–5, " Allocation Operators" on page 7-63. The operator type specifies the method of the allocation. The method determines the cells of the target variable for the *rel-name* relation to which ALLOCATE assigns a value. For the FIRST, LAST, HFIRST, and HLAST operators, ALLOCATE uses the order of the value in the dimension to determine the cell. The dimension order is the default logical order of the allocation dimension. There is no default operator for allocation.

Table 7–5 Allocation Operators

Operator	Description
COPY	Copies the allocation source to all of the target cells that have a basis data value that is not `NA`.
HCOPY	Copies the allocation source to all of the target cells specified by the hierarchy even when the data in any of those cells is `NA`. When the source data is `NA`, then that `NA` value is not allocated to the target cells of that allocation.
MIN	Copies the allocation source to the target that has the smallest basis data value.
MAX	Copies the allocation source to the target that has the largest basis data value.
FIRST	Copies the allocation source to the first target cell that has a non-`NA` basis data value.
HFIRST	Copies the allocation source to the first target cell specified by the hierarchy even when the current data value of that cell is `NA`

Table 7–5 (Cont.) Allocation Operators

Operator	Description
LAST	Copies the allocation source to the last target cell that has a non-NA basis data value.
HLAST	Copies the allocation source to the last target cell specified by the hierarchy even when the current data value of that cell is NA
EVEN	Divides the allocation source by the number of target cells that have non-NA basis data values and applies the quotient to each target cell.
HEVEN	Divides the allocation source by the number of target cells, including the ones that have NA values, and applies the quotient to each target cell.
PROPORTIONAL	Divides the allocation source by the sum of the data values of the target cells that have non-NA basis data values, multiplies the basis data value of each target cell by the quotient, and applies the resulting data to the target cell.

NAOPERATOR *operator*

The *operator* after the NAOPERATOR keyword specifies the operator that the ALLOCATE operation uses when it encounters an NA or lock-based deadlock. Valid operators are HFIRST, HLAST, and HEVEN which are described in Table 7–5, " Allocation Operators" on page 7-63.

REMOPERATOR *operator*

The *operator* after the REMOPERATOR keyword specifies the operator that the ALLOCATE operation uses when storing a remainder produced by an allocation. For example, assume you allocate the INTEGER 10 to three cells at the same level in a hierarchy, there is a remainder of 1. The REMOPERATOR specifies where you want the allocation operation to store this remainder. Valid operators for REMOPERATOR are MIN, MAX, FIRST, HFIRST, LAST, and HLAST which are described in Table 7–5, " Allocation Operators" on page 7-63.

ARGS

Indicates additional arguments specify additional parameters for the allocation operation. All of these arguments apply uniformly to the dimension hierarchy specified by *rel-name*.

PARENTALIAS *dimension-alias-name*

Specifies specialized allocation depending on the parent (for example, weighting by parent or child). For *dimension-alias-name*, specify the name of the alias for the dimension of *rel-name*.

ARGS *argument...*
One or more arguments after the ARGS keyword that specify additional parameters for the allocation operation. All of these arguments apply uniformly to the dimension hierarchy specified by *rel-name*.

FLOOR *floorval*
Specifies that when an allocated target data value is less than *floorval*, the data allocated to the target cell is `NA`. This argument applies to the relation only when the PROPORTIONAL operator is specified.

CEILING *ceilval*
Specifies that when an allocated target data value is greater than *ceilval*, the data allocated to the target cell is `NA`. This argument applies to the relation only when the PROPORTIONAL operator is specified.

MIN *minval*
Specifies that when an allocated target data value is less than *minval*, the data allocated to the target cell is *minval*.

MAX *maxval*
Specifies that when an allocated target data value is greater than *maxval*, the value allocated to the target cell is *maxval*.

NAHANDLE IGNORE
Valid only when the OPERATOR is MIN or MAX, specifies that ALLOCATE does not consider `NA` values in a MIN or MAX operation. (Default)

NAHANDLE CONSIDER
Valid only when the OPERATOR is MIN or MAX, specifies that ALLOCATE treats an `NA` value as a zero; however, when the data value of a target cell is actually zero, the zero cell receives the allocated data value and not the `NA` cell.

NAHANDLE PREFER
Valid only when the OPERATOR is MIN or MAX, specifies that ALLOCATE treats an `NA` value as a zero and the `NA` has priority over a zero value, so the `NA` cell receives the allocated data value and not the cell with the actual zero value.

ADD
Specifies that ALLOCATE adds the allocated data to the current data in the target cell.

ASSIGN
Specifies that ALLOCATE replaces the data in the target cell with the allocated data, which is the default behavior.

PROTECT *lockvalueset*
Specifies a set of dimension values that you want to lock so that they cannot be targets of the allocation. Before allocating the source data, the allocation operation normalizes the sources by subtracting the data values of the specified locked cells from the source data.

NONORMALIZE
Specifies that the allocation operation does not normalize the source data. Using NONORMALIZE effectively removes from the allocation the values of the hierarchy at and below the dimension values specified by *lockvalueset*.

READWRITE
The READWRITE keyword specifies that the locked data values cannot be used as source data in a subsequent allocation, thereby locking the data of the hierarchy below the *lockvalueset* values.

WRITE
The WRITE keyword specifies that the allocation cannot store data values in the cells identified by the *lockvalueset* dimension values but the allocation can use the data in those cells as source data in its subsequent steps. However, when in the aggmap you include a SOURCEVAL statement that specifies `NA` or `ZERO` and the locked cell is the source of an allocation, then ALLOCATE sets the value of the locked cell to `NA` or zero after the allocation.

WEIGHTBY *weightobj*
The *weightobj* argument with WEIGHTBY is the name of an variable, formula, or relation whose value or values are the weights that Oracle OLAP applies to the allocated data just before it is stored in the target cell. Using this clause allows for processes such as unit or currency conversion. When a relation is used, the target variable is referenced based on the weight relation and the cell is applied to the allocation target cell.

ADD
ADD specifies that ALLOCATE adds the weight value to the existing data value of the target and assigns the sum to the target cell.

MULTIPLY
Specifies that ALLOCATE multiplies the weight value by the data value of the target and assigning the product. (Default)

WNAFILL *nafillval*
Specifies a value with which to replace an `NA` value in a cell before applying the weight object to the *nafillval* value. The default `NA` fill value is `1` unless you specify the ADD option to the WEIGHTBY argument, which changes the default `NA` fill value to `0`.

Notes

Specifying the Path of the Allocation

The path of the allocation is the route the allocation system takes to go from the source data to the target data. Very different results derive from different allocation paths. You specify the path with the RELATION statements that you enter in the aggmap. The relation objects in the RELATION statements and the order of those statements specify the path and the method of allocation.

The allocation path goes from any level in the hierarchy of a dimension to any lower level of the hierarchy. You use a relation object that relates the members of the hierarchy to each other (a self-relation) to identify the elements of the hierarchy that you want to participate in the allocation. The allocation proceeds down the hierarchy of the dimension in the first RELATION statement in the aggmap, then down the hierarchy of the second RELATION statement, and so on.

When the dimension has more than one hierarchy, you must use the *qdr* argument in the RELATION statement to specify which hierarchy to use for the allocation.

The hierarchy that you specify with a relation must not contain a circular relation (for example, one in which dimension value `A` relates to dimension value `B` which relates to dimension value `C` which relates to dimension value `A`).

Types of Allocation Paths

You can allocate values from a source to a target with a direct allocation path, a recursive descent hierarchy path, a multidimensional allocation path, or a simultaneous multidimensional allocation path.

- Direct allocation path — You can allocate values directly from a source to the final target cells with no allocations to intermediate nodes of the hierarchy. For example, you can allocate source data values specified by dimension values at the `Quarter` level of a hierarchical `time` dimension to those at the `Month` level or those specified by dimension values at the `Year` level to those at the `Month` level.

- Recursive descent hierarchy path — You can allocate values to intermediate nodes of the hierarchy and then to final target cells. For example, you can allocate source data values specified by dimension values at the `Category` level of a `product` dimension to those at the `Subcategory` level and then to those at the `ProductID` level.
- Multidimensional allocation path — You can allocate values first down one dimension and then down another dimension. The allocations can be direct or recursive or a combination of both. The results might vary depending on the order of the allocation.
- Simultaneous multidimensional allocation path — You can do a direct allocation of values simultaneously to variable cells specified by more than one dimension by creating a composite dimension that specifies the non-`NA` cells of the variable to which you want to allocate values. You then use that composite as the basis of the allocation.

Locking Cells in the Allocation Path

Sometimes you want a cell to retain its existing value and to not be affected by an allocation. You can lock a value of the hierarchy of the dimension and thereby remove that value from the allocation path.When you lock a value above the detail level in a hierarchy, then you remove the branch of the hierarchy below that value from the allocation. To lock a value, use the PROTECT argument to the RELATION statement.

For example, when you want to allocate a yearly budget that you revise monthly, then you would set the value of the `budget` at the `Year` level of the `time` dimension hierarchy. You would allocate data to the elements that are at the `Month` level. As the year progresses, you would enter the actual data for a month and then lock that element and reallocate the remaining yearly budget value to see the new monthly targets that are required to meet the annual goal.

When you lock an element, you can specify whether the source value is renormalized. By default, when you lock an element of the hierarchy, the value of the cell of the target variable specified by that element is subtracted from the source value and the remainder is allocated to the target cells. When you do not want the source renormalized during the allocation, specify NONORMALIZE after the PROTECT argument.

Examples

For an example of using RELATION statements in an allocation statement, see the examples in ALLOCMAP, especially Example 7–23, "Entering RELATION Statements in an Allocation Specification" on page 7-55.

SOURCEVAL

Within an allocation specification, a SOURCE VAL statement specifies the value that the ALLOCATE command assigns to a source cell in an allocation operation after it successfully allocates the value that the cell contained before the allocation.

The default value of SOURCEVAL is `NA`, which means that ALLOCATE sets the value of each of the allocated source cells to `NA` following the allocation. When you specify CURRENT as the SOURCEVAL, then the allocated source cells retain the values that they had before the allocation. When you specify `ZERO` as the SOURCEVAL, then ALLOCATE assigns a zero value to each source cell that is allocated.

Syntax

SOURCEVAL [CURRENT|ZERO|<u>NA</u>]

Arguments

CURRENT
Specifies that the value of a source cell after the allocation is the same as its value before the allocation.

ZERO
Specifies that the value of a source cell after the allocation is zero.

NA
Specifies that the value of a source cell after the allocation is `NA`. This is the default value.

VALUESET

Within an allocation specification, a VALUESET statement specifies the target dimension values of an allocation. A dimensioned valueset can be used to specify the allocation targets for an entire non-hierarchical dimension such as a measure or line dimension.

Note: Keep the following restrictions in mind:

- An allocation specification must include at least one RELATION (for allocation) statement or a VALUESET statement.
- You can only specify one RELATION statement or VALUESET statement for each dimension specified in the allocation specification.

Syntax

VALUSET *vs-name*[(*nondimvalueset*)| *qdr*...] OPERATOR *operator* | *opvar* –

[NAOPERATOR *text -exp*] [REMOPERATOR *text -exp*] -

[ARGS [FLOOR *floorval*] [CEILING *ceilval*] –

[MIN *minval*] [MAX *maxval*] –

[ADDT[*Boolean*]| ASSIGN] –

[{PROTECTRW| PROTECTW} [NONORMALIZE] *lockvalueset*] –

[WEIGHTBY [ADD] *weightobj* [WNAFILL *nafillval*]] | -

[WEIGHTBY WEIGHTVAR *wobjr*]]

Arguments

vs-name
Specifies the name of a valueset object that specifies the values of a dimension which are the path of allocation. The cells in the target variable identified by the values in *vs-name* receive the allocated data.

nondim*valueset*
When *vs-name* is a dimensioned valueset, specifies a nondimensioned valueset that is the status used to loop the valueset dimension. When you do not include *nondimvalueset* or *qdr*, Oracle OLAP uses the default logical order of the dimensions, not its current status.

qdr
When *vs-name* is a a non-dimensioned valueset, one or more qualified data references that specify the dimension values to use when allocating data.

OPERATOR *operator*
The *operator* argument after the OPERATOR keyword is a text expression that is one of the operator types described in Table 7–5, " Allocation Operators" on page 7-63. The operator type specifies the method of the allocation. The method determines the cells of the target variable for the *vs-name* relation to which ALLOCATE assigns a value. Unless you have specified a different status using *dimorder valueset*, for the FIRST, LAST, HFIRST, and HLAST operators, ALLOCATE uses the default logical order of the allocation dimension to determine the cell. There is no default operator for allocation.

OPERATOR *opvar*
The *opvar* argument after OPERATOR keyword specifies a `TEXT` variable that specifies different the operation for each of the values of a dimension. The values of the variable are the allocation operators described in Table 7–5, " Allocation Operators" on page 7-63. An operator variable is used to change the allocation operator with the values of one dimension. The *opvar* argument is used with the following types of dimensions:

- Measure dimension -- Changes the allocation method depending upon the variable being allocated. This is useful when a single aggmap is used to allocate several variables that need to be allocated with different methods. Whether you preallocate all of the measures in a single ALLOCATE statement or in separate statements, allocate uses the operation variable to identify the calculation method. The values of the measure dimension are the names of the variables to be allocated. It dimensions a text variable whose values identify the operation to be used to allocate each measure. The aggmap must include a MEASUREDIM (for allocation) command that identifies the measure dimension.
- Line item dimension -- Changes the allocation method depending upon the line item being allocated. The line item dimension is typically non-hierarchical and identifies financial allocations. The line item dimension is used both to dimension the data variable and to dimension a text variable that identifies the

operation to be used to allocate each item. The operation variable is typically used to allocate line items over time.

The *opvar* argument cannot be dimensioned by the dimension it is used to allocate. For example, when you want to specify different operations for the `geography` dimension, then *opvar* cannot be dimensioned by `geography`.

> **Tip:** To minimize the amount of paging for the operator variable, define the *opvar* variable as type of `TEXT` with a fixed width of `8`.

NAOPERATOR *text-exp*

The *operator* after the NAOPERATOR keyword specifies the operator that the ALLOCATE operation uses when it encounters an `NA` or lock-based deadlock. Valid operators are HFIRST, HLAST, and HEVEN which are described in Table 7–5, " Allocation Operators" on page 7-63.

REMOPERATOR *text-exp*

The *operator* after the REMOPERATOR keyword specifies the operator that the ALLOCATE operation uses when storing a remainder produced by an allocation. For example, assume you allocate the INTEGER 10 to three cells at the same level in a hierarchy, there is a remainder of 1. The REMOPERATOR specifies where you want the allocation operation to store this remainder. Valid operators for REMOPERATOR are MIN, MAX, FIRST, HFIRST, LAST, and HLAST which are described in Table 7–5, " Allocation Operators" on page 7-63.

ARGS

Indicates that additional arguments specify additional parameters for the allocation operation. All of these arguments apply uniformly to the valueset.

FLOOR *floorval*

Specifies that when an allocated target value falls below the value specified in *floorval*, Oracle OLAP stores the value as NA.

CEILING *ceilval*

Specifies that when an allocated target value exceeds the value specified in *ceilval*, then Oracle OLAP stores the value as NA.

MIN *minval*

Specifies that when an allocated target value falls below the value specified *minval*, then Oracle OLAP stores the value of *minval* in the target.

MAX *maxval*
Specifies that when an allocated target value exceeds the value specified *maxval*, then Oracle OLAP stores the value of *maxval* in the target

ADDT [*Boolean*]
Boolean with the ADDT phrase specifies the sign of the addition when Oracle OLAP adds target cells to the existing contents of the target cell:

- `TRUE` specifies that the results of the allocation are added to the target. (Default)
- `FALSE` specifies that the results of the allocation are subtracted from the target cell.

PROTECTRW *lockvalueset*
Specifies that the dimension members specified by *lockvalueset* cannot be the targets or source values of allocation. This lets users specify an allocation "lock" on a hierarchical subtree. The current contents of the target cell are subtracted from the source and the source and basis is renormalized.

PROTECTW *lockvalueset*
Specifies that the dimension members specified by *lockvalueset* cannot be the targets of an allocation. However, these target cells are used as the source values for subsequent steps in the allocation process. When the SOURCEVAL statement is set to `0` (zero) or `NA` and these values are reallocated, they will be set appropriately.

NONORMALIZE
Specifies that Oracle OLAP should not renormalize the source and basis based on the protected cells. Specifying this keyword has an effect similar to removing a sub-branch from a hierarchy. Frequently, when you use this keyword, if, after allocation, data is aggregated from the allocation level, the source cell will probably not contain the original allocated amount

WEIGHTBY *weightobj*
Specifies a weight that should be applied to the target cell just before it is stored. This allows for processes such and unit or currency conversion. Value weight objects are variables, formulas and relations. When a relation is used, the target variable is referenced based on the weight relation, and the cell is applied the allocation target cell.

ADD
Specifies that Oracle OLAP adds the value of the weight to the allocation target rather than using multiplication.

WNAFILL *nafillval*
Specifies the default value of the weight variable that should be used. When you do not include an ADD clause, the default value of *nafillval* is `1`. When you include the ADD clause, the default value of *nafillval* is `0` (zero).

WEIGHTBY WEIGHTVAR *wobj*
Specifies that the allocated data should be weighted. The *wobj* argument is the name of a variable, relation, or formula whose values are the weights that Oracle OLAP applies to the allocated data just before it is stored in the target cell. Using this clause allows for processes such as unit or currency conversion and enables you to use different weight objects with the different operators specified in the operator variable you created for the OPERATOR *opvar* clause.

ALLSTAT

The ALLSTAT command sets the status of all dimensions in the current analytic workspace to all their values. ALLSTAT does not, however, set the status of the NAME dimension.

Syntax

ALLSTAT

Notes

Limiting One Dimension

You can set the status of a single dimension to all its values with the LIMIT command.

Examples

Example 7–24 Limiting to All Values

The following STATUS stsatement produces the current status of the dimensions of the variable UNITS.

```
status units

The current status of MONTH is:
Jul96 TO Dec96
The current status of PRODUCT is:
Tents TO Racquets
The current status of DISTRICT is:
DALLAS
```

After you execute an ALLSTAT statement the same STATUS statement produces this output.

```
The current status of MONTH is:
ALL
The current status of PRODUCT is:
ALL
The current status of DISTRICT is:
ALL
```

ANTILOG

The ANTILOG function calculates the value of *e* (the base of natural logarithms) raised to a specific power.

Return Value

DECIMAL

Syntax

ANTILOG(*n*)

Arguments

n

The power of `e` to be returned by the ANTILOG function.

Examples

Example 7–25 Calculating the Value of e Raised to the Second Power

The following function calculates the value of `e` raised to the second power.

```
ANTILOG(2)
```

This function returns the following value.

```
7.38906
```

ANTILOG10

The ANTILOG10 function calculates the value of 10 raised to a specified power.

Return Value

DECIMAL

Syntax

ANTILOG10(*n*)

Arguments

n

The power of `10` to be returned by the ANTILOG10 function.

Examples

Example 7–26 Calculating the Value of Ten Raised to the Third Power

The following function calculates the value of `10` raised to the third power.

```
ANTILOG10(3)
```

This function returns the following value.

```
1,000.00
```

ANY

The ANY function returns `YES` when any values of a Boolean expression are `TRUE`, or `NO` when none of the values of the expression are `TRUE`.

Return Value

BOOLEAN.

Syntax

ANY(*boolean-expression* [[STATUS] *dimensions*])

Arguments

boolean-expression
The Boolean expression to be evaluated

STATUS
Can be specified when one or more of the dimensions of the result of the function are not dimensions of the Boolean expression. (See the description of the *dimensions* argument.) When you specify the STATUS keyword when this is not the case, Oracle OLAP produces an error.

In cases where one or more of the dimensions of the result of the function are not dimensions of the Boolean expression, the STATUS keyword might be *required* in order for Oracle OLAP to process the function successfully, or the STATUS keyword might provide a performance enhancement. See "The STATUS Keyword" on page 7-81.

dimensions
The dimensions of the result. By default, ANY returns a single `YES` or `NO` value. When you indicate one or more dimensions for the result, ANY tests for `TRUE` values along the dimensions that are specified and returns an array of values. Each dimension must be either a dimension of *boolean-expression* or related to one of its dimensions. When it is a related dimension, you can specify the name of the relation instead of the dimension name. This makes it possible for you to choose which relation is used when there is more than one.

Notes

NA Values

When the Boolean expression involves an `NA` value, ANY returns a `YES` or `NO` result when it can, as shown in Table 7–6, " Results of ANY with Boolean Expressions with NA Values".

Table 7–6 Results of ANY with Boolean Expressions with NA Values

Boolean Expression	Result
`NA EQ NA`	`YES`
`NA NE NA`	`NO`
`NA EQ` non-NA	`NO`
`NA NE` non-NA	`YES`
`NA AND NO`	`NO`
`NA OR YES`	`YES`

However, in cases where a `YES` or `NO` result would be misleading, ANY returns `NA`. For example, when you test whether an `NA` value is greater than a non-`NA` value, ANY returns `NA`.

The Effect of NASKIP

ANY is affected by the NASKIP option. When NASKIP is set to `YES` (the default), ANY ignores `NA` values and returns `YES` when any of the values of the expression that are not `NA` are `TRUE` and returns `NO` when none of the values are `TRUE`. When NASKIP is set to `NO`, ANY returns `NA` when any value of the expression is `NA`. When all the values of the expression are `NA`, ANY returns `NA` for either setting of NASKIP.

Data with a Type of DAY, WEEK, MONTH, QUERTER, or YEAR

When *boolean-expression* is dimensioned by a dimension with a type of DAY, WEEK, MONTH, QUARTER, or YEAR, you can specify any other dimension of this type as a related *dimension.* Oracle OLAP uses the implicit relation between these dimensions. To control the mapping of one of these dimension to another (for example, from weeks to months), you can define an explicit relation between the dimensions and specify the name of the relation as the *dimension* argument to the ANY function.

For each time period in the related dimension, Oracle OLAP tests the data values for all the source time periods that end in the target time period. This method is used regardless of which dimension has the more aggregate time periods.

The STATUS Keyword

When one or more of the dimensions of the result of the function are not dimensions of the Boolean expression, Oracle OLAP creates a temporary variable to use while processing the function. When you specify the STATUS keyword, Oracle OLAP uses the current status instead of the default status of the related dimensions for calculating the size of this temporary variable.

When the size of the temporary variable for the results of the function would exceed 2 gigabytes, you *must* specify the STATUS keyword in order for Oracle OLAP to execute the function successfully. When the dimensions of the Boolean expression are limited to a few values and are physically fragmented, you can specify the STATUS keyword to improve the performance of the function.

When you use the ANY function with the STATUS keyword in an expression that requires going outside of the status for results (for example, with the LEAD or LAG functions or with a qualified data reference), the results outside of the status will be returned as `NA`.

Related Statements

COUNT, EVERY, and NONE.

Examples

Example 7–27 Testing for Any True Values by District

Suppose you want to find out which districts had at least one month with sales greater than $150,000 for sportswear. You use the ANY function to determine whether the Boolean expression (`sales GT 150000`) is `TRUE` for any month. To have the result dimensioned by district, specify `district` as the second argument in the ANY function.

```
LIMIT product TO 'SPORTSWEAR'
REPORT HEADING 'High Sales' ANY(sales GT 150000, district)
```

The preceding statements produce the following output.

```
DISTRICT       High Sales
-------------- ----------
Boston                 NO
Atlanta               YES
Chicago                NO
Dallas                YES
Denver                 NO
Seattle                NO
```

Example 7–28   Testing for Any True Values by Region

You might also want to find out which regions had at least one month in which at least one district had sportswear sales greater than $150,000. Since the `region` dimension is related to the `district` dimension, you can specify `region` instead of `district` as a dimension for the results of ANY.

```
report heading 'High Sales' any(sales gt 150000, region)
```

The preceding statement produces the following output.

```
REGION         High Sales
-------------- ----------
East                  YES
Central               YES
west                   NO
```

ARCCOS

The ARCCOS function calculates the angle value (in radians) of a specified cosine.

Return Value

DECIMAL

Syntax

ARCCOS(*expression*)

Arguments

expression
An expression that contains the decimal value of a cosine.

Notes

Invalid Cosine Values
When you provide an ineligible value for the cosine *expression* (that is, a value greater than `1` or less than `-1`), ARCCOS returns a value of `NA`.

Examples

Example 7–29 Calculating the Arc of a Cosine

This example calculates the arc of a cosine that has a value of `0.54030`. The statement

```
SHOW ARCCOS(.54030)
```

produces the following result.

```
1.00
```

8 ARCSIN to CHARLIST

This chapter contains the following OLAP DML statements:

- AW function
- AWDESCRIBE
- AWWAITTIME
- BACK
- BADLINE
- BASEDIM
- BASEVAL
- BEGINDATE
- BITAND
- BLANKSTRIP
- BMARGIN
- BREAK
- CALENDARWEEK
- CALL
- CALLTYPE
- CATEGORIZE
- CDA
- CEIL
- CHANGEBYTES
- CHANGECHARS
- CHARLIST

ARCSIN

The ARCSIN function calculates the angle value (in radians) of a specified sine.

Return Value

DECIMAL

Syntax

ARCSIN(*expression*)

Arguments

expression
An expression that contains the decimal value of a sine.

Notes

Invalid Sine Values
When you provide an ineligible value for the sine *expression* (that is, a value greater than `1` or less than `-1`), ARCSIN returns a value of `NA`.

Examples

Example 8–1 Calculating the Arc of a Sine

This example calculates the arc of a sine that has a value of `0.84147`. The statement

```
SHOW ARCSIN(.84147)
```

produces the following result.

```
1.00
```

ARCTAN

The ARCTAN function calculates the angle value (in radians) of a specified tangent.

To retrieve a full-range (`0 - 2` pi) numeric value indicating the arc tangent of a given ratio, use ARCTAN2.

Return Value

DECIMAL

Syntax

ARCTAN(*expression*)

Arguments

expression
An expression that contains the decimal value of a tangent.

Examples

Example 8–2 Calculating the Arc of a Tangent

This example calculates the arc of a tangent that has a value of `1.56`. The statement

```
SHOW ARCTAN(1.56)
```

produces the following result.

```
1.00
```

ARCTAN2

The ARCTAN2 function returns a full-range (`0 - 2` pi) numeric value indicating the arc tangent of a given ratio. The function returns values in the range of -pi to pi, depending on the signs of the arguments. The values are expressed in radians.

To calculate the angle value (in radians) of a specified tangent that is not a ratio, use ARCTAN.

Return Value

NUMBER

Syntax

ARCTAN2 (*n* / *m*)

Arguments

n
A numeric expression that specifies one component of the ratio. The argument *n* can be in an unbounded range.

m
A numeric expression that specifies the other component of the ratio.

Examples

The following example returns the arc tangent of `.3` and `.2`.

```
SHOW ARCTAN2(.3/.2)
```

```
.982793723
```

ARG

Within an OLAP DML program, the ARG function lets you reference arguments passed to a program. The function returns one argument as a text value.

Note: Use an ARGUMENT statement to define arguments in a program and to negate the need for using the ARG function to reference arguments passed to the program. For more information on how to use the ARGUMENT to define arguments that are passed to a program, see "Declaring Arguments that Will be Passed Into a Program" on page 8-20.

Important: When you want to pass NTEXT arguments, be sure to declare them using ARGUMENT instead of using ARG. With ARG, NTEXT arguments are converted to TEXT, and this can result in data loss when the NTEXT values cannot be represented in the database character set.

Return Value

TEXT

Syntax

ARG(*n*)

Arguments

n

The number by position of the argument whose value you want to reference. `ARG(1)` returns the first argument to the program, `ARG(2)` returns the second argument, and so forth. When the program is called with fewer than *n* arguments, ARG returns a null value. ARG also returns a null value when n is zero or negative.

Notes

Argument Requirements

When a program is invoked as a command -- that is, without parentheses around the arguments -- Oracle OLAP counts each word and punctuation mark on the command line as a separate argument. Therefore, you cannot use ARG when the arguments include arithmetic expressions, functions, qualified data references, or IF...THEN...ELSE statements as arguments.

When you want to include any of these types of expressions in the arguments, you can invoke the program in one of the following ways:

- Invoke it as a command. With this method, the program must handle the arguments as a text expression, perhaps using ARGS, and it must use PARSE to interpret the arithmetic expressions, functions, qualified data references, and IF...THEN...ELSE statements.
- Invoke it as a user-defined function or with CALL and enclose the arguments within parentheses. When you use CALL, the return value is discarded.

Arguments Passed by Value

The ARG function is often preceded by an ampersand (&) in a program line to allow the user flexibility in specifying arguments; in other words, to tell Oracle OLAP not to pass the literal contents of ARG into the program, but what the contents point to. Another way to pass arguments by value is to declare them using an ARGUMENT statement instead of referencing them with the ARG function.

ARGS and ARGFR Functions

To reference all the arguments, or a group of arguments, use ARGS or ARGFR.

ARGCOUNT Function

A program can include ARGCOUNT to verify the number of arguments passed to the program.

Commas or Spaces as Delimiters

In most cases, you can use either commas or spaces between arguments. However, arguments may need to be separated with commas when those arguments include parentheses or negative numbers. Without commas, Oracle OLAP might interpret parenthetical expressions as qualified data references and negative signs as subtraction.

CALLTYPE Function

You can use CALLTYPE to determine whether a program was invoked as a function, as a command, or by using a CALL statement.

Examples

Example 8–3 Assigning Arguments

Suppose you have a program that produces a sales report. You want to be able to produce this report for any two periods of months, so you do not want to limit the `month` dimension to any particular month in the program. Instead, you use ARG functions in the LIMIT command so that the starting and ending months for the two periods can be supplied as arguments when the program is run.

Notice the UPCASE function preceding the ARG functions. UPCASE allows the arguments to be specified in upper- or lowercase, even though dimension values in the analytic workspace are in uppercase. A prefixed `&` (ampersand) would have a similar effect since it tells Oracle OLAP to substitute the values of ARG before the LIMIT command is executed -- in this case, a value of the `month` dimension. However, an `&` (ampersand) has the disadvantage of preventing compilation of program lines in which it appears, and slower execution results.

```
DEFINE salesrpt PROGRAM
PROGRAM
PUSH month product district
TRAP ON cleanup
LIMIT month TO UPCASE(ARG(1)) TO UPCASE(ARG(2))
LIMIT product TO 'CANOES'
LIMIT district TO all
REPORT grandtotals DOWN district sales
LIMIT month TO UPCASE(ARG(3)) TO UPCASE(ARG(4))
REPORT grandtotals DOWN district sales
cleanup:
POP month product district
END
```

To run the program, you specify the program name (`salesrpt`) followed by two sets of months to mark the beginning and the end of the two periods of sales to be reported. Then, when the `LIMIT MONTH` statements are executed, Oracle OLAP passes the months specified on the command line as return values for `ARG(1)`, `ARG(2)`, `ARG(3)`, and `ARG(4)` in the LIMIT commands.

```
salesrpt 'Jan95' 'Mar95' 'Jan96' 'Mar96'
```

This statement produces the following output.

```
PRODUCT: Canoes
              ------------SALES--------------
              ------------MONTH--------------
DISTRICT        Jan95       Feb95     Mar95
---------------------------------------------
Boston        66,013.92  76,083.84  91,748.16
Atlanta       49,462.88  54,209.74  67,764.20
Chicago       45,277.56  50,595.75  63,576.53
Dallas        33,292.32  37,471.29  43,970.59
Denver        45,467.80  51,737.01  58,437.11
Seattle       64,111.50  71,899.23  83,943.86
             ----------  ---------  ---------
             303,625.98 341,996.86 409,440.44
             ========== ========== ==========
PRODUCT: Canoes
             ------------SALES---------------
             ------------MONTH---------------
DISTRICT        Jan96      Feb96       Mar96
---------------------------------------------
Boston        70,489.44  82,237.68  97,622.28
Atlanta       56,271.40  61,828.33  77,217.62
Chicago       48,661.74  54,424.94  68,815.71
Dallas        35,244.72  40,218.43  46,810.68
Denver        44,456.41  50,623.19  57,013.01
Seattle       67,085.12  74,834.29  87,820.04
             ----------  ---------  ---------
             322,208.83 364,166.86 435,299.35
             ========== ========== ==========
```

ARGCOUNT

Within an OLAP DML program, the ARGCOUNT function returns the number of arguments that were specified when the current program was invoked.

Return Value

INTEGER

Syntax

```
ARGCOUNT
```

Notes

Argument Declarations

Arguments can be either declared with an ARGUMENT statement or referenced with the ARG function.

Argument Requirements

When a program is invoked as a command -- that is, without parentheses around the arguments -- Oracle OLAP counts each word and punctuation mark on the command line as a separate argument. Therefore, you should not include arithmetic expressions, functions, qualified data references, or IF...THEN...ELSE... statements as arguments.

When you want to include any of these types of expressions as arguments in a program that will be invoked as a command, use PARSE in the program. Alternatively, you can enclose the arguments within parentheses and invoke the program either as a user-defined function or with CALL. When the program is invoked with CALL, the return value is discarded.

CALLTYPE Function

You can use CALLTYPE to determine whether a program was invoked as a function, as a command, or by using CALL.

Examples

Example 8–4 Checking the Number of Arguments

In the following example, the program, a user-defined function, verifies that three arguments are passed. When the number of arguments passed is not equal to 3, the program terminates with `-1` as a return value.

```
DEFINE threearg PROGRAM INTEGER
LD User-defined function expecting three arguments
PROGRAM
ARGUMENT division TEXT
ARGUMENT product TEXT
ARGUMENT month MONTH
IF ARGCOUNT NE 3
   THEN RETURN -1
    ELSE
     DO
     ...
```

ARGFR

Within an OLAP DML program, the ARGFR function lets you reference the arguments that are passed to a program. The function returns a group of one or more arguments, beginning with the specified argument number, as a single text value. You can use ARGFR only within a program that is invoked as a command, not as a user-defined function or with a CALL statement.

Note: Use an ARGUMENT statement to define arguments in a program and to negate the need for using the ARGFR function to reference arguments passed to the program. For more information on how to use the ARGUMENT to define arguments that are passed to a program, see "Declaring Arguments that Will be Passed Into a Program" on page 8-20.

Important: When you want to pass NTEXT arguments, be sure to declare them using ARGUMENT instead of using ARGFR. With ARGFR, NTEXT arguments are converted to TEXT, and this can result in data loss when the NTEXT values cannot be represented in the database character set.

Return Value

TEXT

Syntax

ARGFR(*n*)

Arguments

n

The number by position of the first argument in the group of arguments you want to reference. `ARGFR(1)` returns the first argument and all subsequent arguments, `ARGFR(2)` returns the second argument and all subsequent arguments, and so forth. When there are fewer than *n* arguments, ARGFR returns a null value. ARGFR also returns a null value when *n* is `0` (zero) or negative.

Notes

Ampersand Substitution

The ARGFR function is often preceded by an ampersand (&) in a program line to allow flexibility in specifying arguments; in other words, to tell Oracle OLAP not to pass the literal contents of ARGFR into the program, but what the contents point to. See "Passing Arguments Using ARG and ARGFR" on page 8-14.

Argument Requirements

When a program is invoked as a command -- that is, without parentheses around the arguments -- Oracle OLAP counts each word and punctuation mark on the command line as a separate argument. Therefore, you cannot include arithmetic expressions, functions, qualified data references, or IF...THEN...ELSE... statements as arguments in the program.

When you want to include any of these types of expressions as arguments in a program invoked as a command, you must include a PARSE statement in the program.

ARG and ARGS Functions

To reference a single argument, use ARG, or to reference all the arguments, use ARGS.

ARGCOUNT Function

A program can include ARGCOUNT to verify the number of arguments passed to the program.

Commas or Spaces as Delimiters

In most cases, you can use either commas or spaces between arguments. However, arguments may need to be separated with commas when those arguments include parentheses or negative numbers. Without commas, Oracle OLAP might interpret parenthetical expressions as qualified data references and negative signs as subtraction.

CALLTYPE Function

You can use CALLTYPE to determine whether a program was invoked as a function, as a command, or by using CALL.

Examples

Example 8–5 Passing Arguments Using ARG and ARGFR

Suppose you have a program that produces a sales report. You want to be able to produce this report for any product and any period of months, so you do not want to limit the `product` and `month` dimensions to specific values in the program. Instead, you can use the LIMIT command using ARG for the `product` argument and an ARGFR function for the `month` argument. This way, these items can be specified when the program is run.

When ARGFR is included in the LIMIT command preceded by an ampersand (`&`), Oracle OLAP substitutes the values of `&ARGFR` before the command is executed and, as a result, treats the whole argument as a phrase of the LIMIT command. The `salesreport` program has a LIMIT command that includes `&ARGFR`.

```
DEFINE salesrpt PROGRAM
PROGRAM
PUSH product month district
TRAP ON cleanup
LIMIT product TO UPCASE(ARG(1))
LIMIT month TO &ARGFR(2)
LIMIT district TO ALL
REPORT grandtotals DOWN district sales
cleanup:
POP product month district
END
```

The command line for the `salesrpt` program must include two or more arguments. The first argument is the product for the report, and the second and subsequent arguments are the months. In the `LIMIT month` statement, the `&ARGFR(2)` function returns the months that were specified as arguments on the command line.

The following statement executes the `salesrpt` program, specifying `Jan96`, `Feb96`, `Mar96`, and `Apr96` for the values of `month`.

```
salesrpt 'Canoes' 'Jan96' TO 'Apr96'
```

The statement produces the following output.

```
PRODUCT: CANOES
        -------------------SALES------------------
        -------------------MONTH------------------
DISTRICT    Jan96     Feb96      Mar96     Apr96
------- ---------- ---------- ---------- ---------
Boston  70,489.44  82,237.68  97,622.28 134,265.60
Atlanta 56,271.40  61,828.33  77,217.62 109,253.38
Chicago 48,661.74  54,424.94  68,815.71  93,045.46
Dallas  35,244.72  40,218.43  46,810.68  64,031.28
Denver  44,456.41  50,623.19  57,013.01  78,038.13
Seattle 67,085.12  74,834.29  87,820.04 119,858.56
       ---------- ---------- ---------- ----------
       322,208.83 364,166.86 435,299.34 598,492.41
       ========== ========== ========== ==========
```

The following statement specifies the first three months of 1996.

```
salesrpt 'Tents' quarter 'Q1.96'
```

The statement produces the following output.

```
PRODUCT: TENTS
               -------------SALES-------------
               -------------MONTH-------------
DISTRICT          Jan96     Feb96      Mar96
-------------- ---------- ---------- ---------
Boston         50,808.96  34,641.59  45,742.21
Atlanta        46,174.92  50,553.52  58,787.82
Chicago        31,279.78  31,492.35  42,439.52
Dallas         50,974.46  53,702.75  71,998.57
Denver         35,582.82  32,984.10  44,421.14
Seattle        45,678.41  43,094.80  54,164.06
              ----------  ---------- ---------
              260,499.35 246,469.11 317,553.32
              ========== ========== ==========
```

ARGS

Within an OLAP DML program, the ARGS function lets you reference the arguments that are passed to a program. The function returns all the arguments as a single text value. You can use the ARGS function only within a program that is be invoked as a command, not as a user-defined function or with a CALL statement.

Note: Use an ARGUMENT statement to define arguments in a program and to negate the need for using the ARGS function to reference arguments passed to the program. For more information on how to use the ARGUMENT to define arguments that are passed to a program, see "Declaring Arguments that Will be Passed Into a Program" on page 8-20.

Important: When you want to pass NTEXT arguments, be sure to declare them using ARGUMENT instead of using ARGS. With ARGS, NTEXT arguments are converted to TEXT, and this can result in data loss when the NTEXT values cannot be represented in the database character set.

Return Value

TEXT

Syntax

```
ARGS
```

Notes

No Arguments

When no arguments have been specified for the program, ARGS returns a null value.

Ampersand Substitution

The ARGS function is often preceded by an ampersand (`&`) in a program line to allow flexibility in specifying arguments; in other words, to tell Oracle OLAP not to pass the literal contents of ARGS into the program, but what the contents point to. (See "Passing Arguments Using ARGS" on page 8-17.

ARGFR Function

To reference a single argument use ARG, or to reference a group of arguments beginning with a specified argument use ARGFR.

ARGCOUNT Function

A program can include an ARGCOUNT function to verify the number of arguments passed to the program.

Commas or Spaces as Delimiters

In most cases, either commas or spaces can be used between arguments. However, arguments may need to be separated with commas when those arguments include parentheses or negative numbers. Without commas, Oracle OLAP might interpret parenthetical expressions as qualified data references and negative signs as subtraction.

CALLTYPE Function

You can use CALLTYPE to determine whether a program was invoked as a function, as a command, or by using CALL.

Examples

Example 8–6 Passing Arguments Using ARGS

Assume you have a program that produces a simple sales report. You want to be able to produce this report for any month, so you do not want to limit the `month` dimension to any fixed month in the program. You can use the ARGS function in your LIMIT command so that the months for the report can be supplied as an argument when the program is run.

When ARGS is included in the LIMIT command preceded by an ampersand (`&`), Oracle OLAP substitutes the values of `&ARGS` before the command is executed and, as a result, treats the whole argument as a phrase of the LIMIT command. The `salesreport` program has a LIMIT command that includes `&ARGS`.

```
DEFINE salesrpt PROGRAM
PROGRAM
PUSH month product district
TRAP ON cleanup
LIMIT month TO &ARGS
LIMIT product TO 'CANOES'
LIMIT district TO ALL
REPORT grandtotals DOWN district sales
cleanup:
POP month product district
END
```

When you execute the following statement, the LIMIT command will use the values `Jan96` and `Feb96` for the `month` dimension.

```
salesrpt 'Jan96' 'Feb96'
```

The statement produces the following output.

```
PRODUCT: CANOES
       --------SALES--------
       --------MONTH--------
DISTRICT       Jan96      Feb96
----------------------------------
Boston       70,489.44  82,237.68
Atlanta      56,271.40  61,828.33
Chicago      48,661.74  54,424.94
Dallas       35,244.72  40,218.43
Denver       44,456.41  50,623.19
Seattle      67,085.12  74,834.29
         ---------- ---------- --
            322,208.83 364,166.86
         ========== ========== ==
```

ARGUMENT

The ARGUMENT statement declares an argument that is expected by a program. Within the program, the argument is stored in a structure similar to a variable or valueset. The argument is initialized with the value that was passed when the program was invoked. An argument exists only while the program is running.

The ARGUMENT statement is used only in programs, and it must precede the first executable line in the program. Be careful to distinguish the ARG abbreviation of the ARGUMENT statement from the ARG function.

Syntax

ARGUMENT *name* {*datatype*|*dimension*|VALUESET *dim*}

Arguments

name
The name by which the argument will be referenced in the program. An argument cannot have the same name as a local variable or valueset. You name an argument according to the rules for naming analytic workspace objects (see the DEFINE command).

datatype
The data type of the argument, which indicates the kind of data to be stored. You can specify any of the data types that are listed and described in the DEFINE VARIABLE entry. Also, when you want to the program to be able to receive an argument without converting it to a specific datatype, you can also specify `WORKSHEET` for the data type.

Important: When you declare an argument to be of type NTEXT, and a TEXT value is passed into the program, Oracle OLAP converts the TEXT value to NTEXT. Similarly, when you declare an argument to be of type TEXT, and an NTEXT value is passed into the program, Oracle OLAP converts the NTEXT value to TEXT. Data can be lost when NTEXT is converted to TEXT.

dimension
The name of a dimension, whose value will be contained in the argument. The argument will hold a single value of the dimension. Assigning a value that does not currently exist in the dimension causes an error.

VALUESET *dim*
Indicates that *name* is a valueset. The keyword *dim* specifies the dimension for which the valueset holds values. Argument valuesets can be used within the program in the same way you would use a valueset in the analytic workspace.

Notes

The Life Span of an Argument

An argument exists only while the program in which it is declared is running. When the program terminates, the argument ceases to exist and its value is lost. Therefore, an argument is not an analytic workspace object.

A program can terminate when a RETURN or SIGNAL statement, or at the last line of the program executes. When the program calls a subprogram, the original program is temporarily suspended and the argument still exists when the subprogram ends and control returns to the original program. A program that calls itself recursively has separate arguments for each running copy of the program.

Declaring Arguments that Will be Passed Into a Program

When declaring arguments that are passed into a program special considerations apply.

Arguments Passed by Value Arguments are passed into a program by value. This means that the called program is given only the value of an argument, without access to any analytic workspace object to which it might be related. Therefore, you can change an argument value within the called program without affecting any value outside the program. You can think of an argument variable or valueset as a conveniently initialized local variable or local valueset.

Argument Processing for a Function When a program is invoked either with a CALL statement or as a function, the following two-step process occurs:

1. The specified data types are established. Argument expressions specified by the calling program are evaluated left to right, and their data types are identified. An expression representing a dimension value can be a text (TEXT or ID),

numeric (INTEGER, DECIMAL, and so on), or RELATION value. An error in one argument expression stops the process.

2. Each specified data type is matched with the declared data type. Argument expressions are matched positionally with the declared arguments. The first argument expression is matched with the first declared argument, the second argument expression with the second argument, and so on. Each expression is converted in turn to the declared data type of the declared argument.

When an argument is declared as a dimension value, the matching value passed from the calling program can be TEXT or ID (representing a value of the specified dimension), numeric (representing a logical dimension position), or RELATION (representing a physical dimension position). The RELATION method is the way Oracle OLAP passes along dimension values that are the result of evaluating a dimension name or relation name used as the matching value. When the matching value is a noninteger numeric value (for example, DECIMAL), it is rounded to the nearest integer to represent a logical dimension position.

When an argument is declared as something other than a dimension value, and the matching value from the calling program is a RELATION value, an error will occur. When you want to pass a RELATION value and receive it as a TEXT argument, use CONVERT to convert the value in the program's argument list.

When an argument is declared as a valueset of a dimension, only the name of a valueset of that dimension will be accepted as an argument.

When an error occurs in either the first or second step, the program is not executed.

Argument Processing for a Command When a program is invoked as a standalone command with its arguments not enclosed by parentheses, the arguments are matched positionally with the declared arguments. The called program can reference the specified arguments either as declared arguments or through the ARG (*n*), ARGS, and ARGFR (*n*) functions. In this situation, the arguments are passed as text strings, not by value.

Extra Arguments When the calling program specifies more arguments than there are declarations in the called program, the extra arguments are ignored. When the calling program specifies fewer arguments than there are declarations in the called program, the extra arguments are given `NA` values.

Duplicate Names

Ordinarily, when you give an argument the same name as an analytic workspace object, the argument (not the analytic workspace object) will be referenced within the program. Exceptions to this rule occur only when the statement in which the reference is made requires an analytic workspace object as an argument. The following OLAP DML statements require an analytic workspace object to be specified as an argument.

CONSIDER
COPYDFN
DEFINE
DELETE
DESCRIBE
EXPORT
IMPORT
LOAD
MOVE
POP
PUSH
RENAME
STATUS

CALLTYPE Function

You can use the CALLTYPE function to determine whether a program was invoked as a function, as a command, or by using a CALL statement.

No Arguments in Formulas

You cannot use declared arguments in a FORMULA.

Examples

Example 8–7 Passing an Argument to a User-Defined Function

Sometimes verifying user input to the GET function can become complicated. The usual method involves a line of code such as the following one.

```
SHOW GET(INT VERIFY VALUE GT 0 AND VALUE LT 100 -
   IFNOT 'The value must be between 1 and 100')
```

You can create a user-defined function to make the GET expression simpler. For example, the following program can be used as a function to check for values between `0` and `100`.

```
DEFINE verit PROGRAM BOOLEAN
PROGRAM
  ARGUMENT uservalue INT
  TRAP ON haderror NOPRINT
  IF uservalue GT 100
     THEN SIGNAL toobig 'The value must be 100 or smaller.'
  ELSE IF uservalue LT 0
     THEN SIGNAL toosmall 'The value must be 0 or greater.'
  RETURN TRUE
haderror:
  RETURN FALSE
END
```

The following GET expression uses the `verit` function.

```
SHOW GET(INT VERIFY VERIT(VALUE) IFNOT ERRORTEXT)
```

Example 8–8   Passing Multiple Arguments

Suppose, in the `product.rpt` program, that you want to supply a second argument that specifies the column width for the data columns in the report. In the `product.rpt` program, you would add a second ARGUMENT statement to declare the INTEGER argument to be used in setting the value of the COLWIDTH option.

```
ARGUMENT natext TEXT
ARGUMENT widthamt INTEGER
NASPELL = natext
COLWIDTH = widthamt
```

To specify eight-character columns, you could run the `product.rpt` program with the following statement.

```
CALL product.rpt ('Missing' 8)
```

When the `product.rpt` program also requires the name of a product as a third argument, then in the `product.rpt` program you would add a third ARGUMENT statement to handle the product argument, and you would set the status of the `product` dimension using this argument.

```
ARGUMENT natext TEXT
ARGUMENT widthamt INTEGER
ARGUMENT rptprod PRODUCT
NASPELL = natext
COLWIDTH = widthamt
LIMIT product TO rptprod
```

You can run the `product.rpt` program with the following statement.

```
CALL product.rpt ('Missing' 8 'TENTS')
```

In this example, the third argument is specified in uppercase letters with the assumption that all the dimension values in the analytic workspace are in uppercase letters.

Example 8–9 Using the ARGUMENT Statement

Suppose you are writing a program, called `product.rpt`. The `product.rpt` program produces a report, and you want to supply an argument to the report program that specifies the text that should appear for an NA value in the report. In the `product.rpt` program, you can use the declared argument `natext` in an ARGUMENT statement to set the NASPELL option to the value provided as an argument.

```
ARGUMENT natext TEXT
NASPELL = natext
```

To specify `Missing` as the text for NA values, you can execute the following statement.

```
CALL product.rpt ('Missing')
```

In this example, literal text enclosed in single quotes provides the value of the text argument. However, any other type of text expression works equally well, as shown in the next example.

```
DEFINE natemp VARIABLE TEXT TEMP
natemp = 'Missing'
CALL product.rpt (natemp)
```

Example 8–10 Passing the Text of an Expression

Suppose you have a program named `custom.rpt` that includes a REPORT statement, but you want to be able to use the program to present the values of an expression, such as `sales - expense`, as well as single variables.

```
custom.rpt 'sales - expense'
```

Note that you must enclose the expression in single quotation marks. Because the expression contains punctuation (the minus sign), the quotation marks are necessary to indicate that the entire expression is a single argument.

In the `custom.rpt` program, you could use the following statements to produce a report of this expression.

```
ARGUMENT rptexp TEXT
REPORT &rptexp
```

For an example of using ampersand substitution to pass multiple dimension values, see "Using Ampersand Substitution with LIMIT" on page 16-17.

Example 8–11 Passing Workspace Object Names and Keywords

Suppose you design a program called `sales.rpt` that produces a report on a variable that is specified as an argument and sorts the `product` dimension in the order that is specified in another argument. You would run the `sales.rpt` program by executing a statement like the following one.

```
sales.rpt units d
```

In the `sales.rpt` program, you can use the following statements.

```
ARGUMENT varname TEXT
ARGUMENT sortkey TEXT
SORT product &sortkey &varname
REPORT &varname
```

After substituting the arguments, these statements are executed in the `sales.rpt` program.

```
SORT product D units
REPORT units
```

ASCII

The ASCII function returns the decimal representation of the first character of an expression.

Return Value

INTEGER.

Syntax

ASCII (*text-exp*)

Arguments

text-exp
A text expression.

Notes

Returning EBCDIC Values

When your database character set is 7-bit ASCII, then this function returns an ASCII value. When your database character set is EBCDIC Code, then this function returns an EBCDIC value. There is no corresponding EBCDIC character function

Examples

The following example returns the ASCII decimal equivalent of the letter "Q":

```
SHOW ASCII('Q')
81
```

AVERAGE

The AVERAGE function calculates the average of the values of an expression.

Return Value

DECIMAL

Syntax

AVERAGE(*expression* [[STATUS] *dimensions*])

Arguments

expression
The expression whose values are to be averaged.

STATUS
Can be specified when one or more of the dimensions of the result of the function are not dimensions of the expression. (See the description of the *dimensions* argument.) When you specify the STATUS keyword when this is not the case, Oracle OLAP produces an error.

In cases where one or more of the dimensions of the result of the function are not dimensions of the expression, the STATUS keyword might be required in order for Oracle OLAP to process the function successfully, or the STATUS keyword might provide a performance enhancement. See "The STATUS Keyword" on page 8-28.

dimensions
The dimensions of the result. By default, AVERAGE returns a single value. When you indicate one or more dimensions for the results, AVERAGE calculates an average for each value of the dimensions that are specified and returns an array of values. Each dimension must be either a dimension of *expression* or related to one of its dimensions. When it is a related dimension, you can specify the name of the relation instead of the dimension. This makes it possible for you to choose which relation is used when there is more than one.

Notes

NA Values

AVERAGE is affected by the NASKIP option. When NASKIP is set to `YES` (the default), AVERAGE ignores `NA` values and returns the average of the values that are not `NA`. When NASKIP is set to `NO`, AVERAGE returns `NA` when any value of the expression is `NA`. When all the values of the expression are `NA`, AVERAGE returns `NA` for either setting of NASKIP.

Averaging Over a Dimension of Type DAY, WEEK, MONTH, QUARTER, or YEAR

When *expression* is dimensioned by a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR, you can specify any other dimension that has one of these types as a related dimension. Oracle OLAP uses the implicit relation between the two dimensions. To control the mapping of one of these types of dimensions to another (for example, from weeks to months), you can define an explicit relation between the two dimensions and specify the name of the relation as the dimension argument to the AVERAGE function.

For each time period in the related dimension, Oracle OLAP averages the data for all the source time periods that end in the target time period. This method is used regardless of which dimension has the more aggregate time periods. To control the way in which data is aggregated or allocated between the periods of two dimensions, you can use the TCONVERT function.

The STATUS Keyword

When one or more of the dimensions of the result of the function are not dimensions of the expression, Oracle OLAP creates a temporary variable to use while processing the function. When you specify the STATUS keyword, Oracle OLAP uses the current status instead of the default status of the related dimensions for calculating the size of this temporary variable.

When the size of the temporary variable for the results of the function would exceed 2 gigabytes, you must specify the STATUS keyword in order for Oracle OLAP to execute the function successfully. When the dimensions of the expression are limited to a few values and are physically fragmented, you can specify the STATUS keyword to improve the performance of the function.

When you use the AVERAGE function with the STATUS keyword for an expression that requires going outside of the status for results (for example, with the LEAD or LAG functions or with a qualified data reference), the results outside of the status will be returned as `NA`.

Examples

Example 8–12 Calculating Average Monthly Sales

This example shows how to calculate the average monthly sales of sportswear for each sales district.

```
LIMIT product TO 'SPORTSWEAR'
REPORT W 14 HEADING 'Average Sales' AVERAGE(sales district)
```

The preceding statements produce the following output.

```
DISTRICT    Average Sales
----------- --------------
Boston       69,150.41
Atlanta     151,192.36
Chicago      95,692.99
Dallas      162,242.89
Denver       88,892.72
Seattle      54,092.32
```

You might also want to see the average monthly sales for each region. Since the `region` dimension is related to the `district` dimension, you can specify `region` instead of `district` as a dimension for the results of AVERAGE.

AW command

The syntax of the AW command varies depending on the task that you want to perform.

AW ALIASLIST
AW ALLOCATE
AW ATTACH
AW CREATE
AW DELETE
AW DETACH
AW LIST
AW SEGMENTSIZE

Notes

Triggering Program Execution When an AW Statement Executes

When a program named TRIGGER_AW exists in the analytic workspace, the execution of any AW ATTACH, AW CREATE, AW DELETE, or AW DETACH statement automatically executes that program. See "Trigger Programs" on page 1-14 and TRIGGER_AW, for more information.

When an AW ATTACH statement executes Oracle OLAP checks for other programs as well. See "Programs Executed When Attaching Analytic Workspaces" on page 8-39 for more information.

AW ALIASLIST

The AW ALIASLIST command assigns or deletes one or more workspace alias for the specified attached workspace or, when no workspace is specified, for the current workspace. ALIAS indicates that the alias or aliases should be assigned, and UNALIAS indicates that the alias or aliases should be deleted.

Syntax

AW ALIASLIST [*workspace*] {ALIAS|UNALIAS} *alias1*, *alias2*, ...

Arguments

workspace
The name of the analytic workspace. You can specify either a workspace name or a workspace alias, depending on the keywords you are using.

ALIAS
Assigns one or more workspace alias for the specified attached workspace or, when no workspace is specified, for the current workspace. ALIAS indicates that the alias or aliases should be assigned, and UNALIAS indicates that the alias or aliases should be deleted.

All aliases for a given workspace are automatically deleted when you detach a workspace. Therefore, each time you attach an unattached workspace, you must reassign its aliases.

UNALIAS
Deletes one or more workspace alias for the specified attached workspace or, when no workspace is specified, for the current workspace.

alias1
alias2
When you assign an alias, keep in mind the following rules for workspace aliases: Aliases can contain only letters, numerals, and underscores; they cannot begin with a numeral; they cannot be reserved words in the OLAP DML; and they can be no more than 26 characters in length. (Use RESERVED to identify reserved words.) All characters must come from the database character set.

Notes

Duration of Alias

All aliases for a given workspace are automatically deleted when you detach a workspace. Therefore, each time you attach an unattached workspace, you must reassign its aliases.

No Current Workspace

Your session does not have to have a current workspace. When you start Oracle OLAP without specifying a workspace name, the `EXPRESS` workspace will be first on the list, but there is no current workspace until you specify one with the AW ATTACH statement. (You can make the `EXPRESS` workspace current by specifying its name in the AW ATTACH statement.)

EXPRESS Workspace

When your database is installed with the OLAP option, the `EXPRESS` workspace is always attached in read-only mode in your session. It never automatically becomes the current workspace, even when it is the first or only workspace in your workspace list, because it is for internal use by Oracle OLAP. You can make the `EXPRESS` workspace the current workspace by explicitly attaching it, but this is not recommended. You cannot detach the `EXPRESS` workspace.

Examples

Example 8–13 Assigning an Alias

The following statement assigns `sdemo` as an alias for the `demo` workspace, which was created by a user named `scott`. The full name of the workspace is specified because the current user is not `scott`.

```
AW ALIASLIST scott.demo ALIAS sdemo
```

In the following statement, the user named `scott` assigns `mydemo` as an alias for the same workspace. This statement can specify only the name of the work space (not the full name), because the current user is `scott`.

```
AW ALIASLIST demo ALIAS mydemo
```

AW ALLOCATE

The AW ALLOCATE command allocates space for your workspace.

Syntax

AW ALLOCATE *n* [K, M, or G] [*workspace*]

Arguments

n
The amount of space to allocate.

workspace
The name of the analytic workspace. You can specify either a workspace name or a workspace alias.

AW ATTACH

The AW ATTACH command attaches a workspace to your session. Oracle OLAP makes the specified workspace the current one. Previously attached workspaces move down in the list of attached workspaces to make room for the new current one at the top of the list.

When you attach more than one workspace, the code and data in all the attached workspaces are available during your session. The current workspace is first on the workspace list, which Oracle OLAP keeps for your session.

> **Note:** When an AW ATTACH statement executes, it can trigger the execution of several other programs. See "Programs Executed When Attaching Analytic Workspaces" on page 8-39 for more information.

Syntax

AW ATTACH *workspace* -

[ONATTACH [*progname*]|NOONATTCH] -

[RO|RW|RWX|MULTI] [WAIT|NOWAIT] -

[AUTOGO [*progname*]|NOAUTOGO] -

[AFTER *workspace*|BEFORE *workspace*|LAST|FIRST] -

[PASSWORD *password*]

Arguments

workspace

The name of the analytic workspace. When you use the ATTACH keyword to attach a workspace that is not already attached, you must specify the workspace name. Again this is because no alias is assigned. However, when you use the ATTACH keyword on an already attached workspace (for example, in order to change its position in the workspace list), you can use an assigned alias

ONATTCH [*progname*]
An Onattach program automatically executes when the workspace is started:

- When you specify the ONATTCH keyword without following it with a program name, Oracle OLAP looks in the workspace for a program named `ONATTACH` and executes it if it exists. This syntax is provided for clarity in your programs. You can get the same results by *not* specifying ONATTACH.
- When you specify the ONATTCH keyword and you follow it with a program name, Oracle OLAP looks in the workspace for a program of that name. When it exists, Oracle OLAP executes that program, even when a program named `ONATTACH` exists in the workspace. See "Programs Executed When Attaching Analytic Workspaces" on page 8-39 for more information.

NOONATTACH
Specifying NOONATTACH indicates that when a program named `ONATTACH` exists in the workspace, Oracle OLAP should not execute that program.

AUTOGO *progname*
When you do not specify progname, the AUTOGO clause automatically runs the program specified a program named AUTOGO if one exists in the attached workspace. When you do specify *progname*, the AUTOGO clause automatically runs the specified program in the attached program. See "Programs Executed When Attaching Analytic Workspaces" on page 8-39 for more information.

NOAUTOGO
Specifying NOAUTOGO indicates that there is no Autogo program. This syntax is provided for clarity in your programs. You can get the same results by *not* specifying AUTOGO *progname*.

RO
Specifies that the workspace is attached in read-only access mode. (Default) Users can make private changes to the data in the workspace to perform what-if analysis but cannot commit any of these changes.

A workspace that is attached read-only can be accessed simultaneously by several sessions. The read-only attach mode is compatible with the read/write and multiwriter access mode. A user can attach an analytic workspace in read-only mode when other users have the workspace attached in either read/write and multiwriter access mode. Likewise, a user cannot attach an analytic workspace in read/write exclusive mode when another user has it attached in read-only mode. When you attach a workspace with read-only access, Oracle OLAP executes a program called PERMIT_READ, when it finds one in the workspace.

RW

Specifies that the workspace is attached in read/write access mode. Only one user can have an analytic workspace open in read/write at a time. The user has to commit either all or none of the changes made to the workspace.

A workspace that is attached read/write non-exclusive can be accessed simultaneously by several sessions. The read/write non-exclusive attach mode is only compatible with the read-only access mode. A user can attach an analytic workspace in read/write mode when other users have the workspace attached in read-only mode; however, a user not attach an analytic workspace in read/write mode when another user has it attached in any other mode. Likewise, a user cannot attach an analytic workspace in any mode other than read-only when another user has it attached in read/write non-exclusive mode. When you attach a workspace with read/write access, Oracle OLAP executes a program called PERMIT_WRITE, when it finds one in the workspace. See "Permission Programs" on page 8-40.

RWX

Specifies that the workspace is attached in read/write exclusive access mode. Only one user can have an analytic workspace open in read/write exclusive at a time. The user has to commit either all or none of the changes made to the workspace.

A workspace that is attached read/write exclusive cannot be accessed by any other sessions. The read/write exclusive attach mode is not compatible with any other access modes. A user cannot attach an analytic workspace in read/write exclusive mode when another user has it attached in any mode. Likewise, a user cannot attach an analytic workspace in any other mode when another user has it attached in read/write exclusive mode. When you attach a workspace with read/write access, Oracle OLAP executes a program called PERMIT_WRITE, when it finds one in the workspace. See "Permission Programs" on page 8-40.

MULTI

Specifies that the workspace is attached in multiwriter access mode. A workspace that is attached in multiwriter mode can be access simultaneously by several sessions. In multiwriter mode, users can simultaneously modify the same analytic workspace in a controlled manner by specifying specify the attachment mode (read-only or read/write) for individual variables, relations, valuesets, and dimensions.

> **See:** Table 8–1, " Statements for Managing Objects When Attached in Multiwriter Mode" on page 8-38 for a list of the OLAP DML statements that you use to manipulate objects in an analytic workspace that is attached in multiwriter mode,.

The multiwriter attach mode is only compatible with read-only and multiwriter modes. A user cannot attach an analytic workspace in multiwriter mode when another user has it attached in read/write or exclusive modes. Likewise, a user cannot attach an analytic workspace in read/write or exclusive mode when another user has it attached in multiwriter mode.

WAIT
NOWAIT
Specifies whether Oracle OLAP waits for a workspace to become available for access when you request access to a workspace that is being used with read/write exclusive access or when you request read/write access to a workspace that is already being used with read/write non-exclusive access. NOWAIT (the default) causes Oracle OLAP to produce an error message indicating that the workspace is unavailable. When you specify WAIT, Oracle OLAP will wait for the workspace to become available for access. The number of seconds that Oracle OLAP will wait for access depends on the value of the Oracle OLAP option AWWAITTIME. For more information, see AWWAITTIME and "Workspace Sharing" on page 8-41.

FIRST
Makes the workspace you are attaching the current workspace in the workspace list. (Default)

LAST
Puts the workspace after the current workspace in the workspace list and before the `EXPRESS` workspace. When there are other workspaces attached before the `EXPRESS` workspace, the specified workspace is attached after them. When there are no workspaces before the `EXPRESS` workspace, LAST makes the specified workspace the current one. LAST ignores any workspaces after the `EXPRESS` workspace.

AFTER *workspace*
BEFORE *workspace*
Let you specify the position in the workspace list of the newly attached workspace relative to a workspace that is already attached. Use AFTER, rather than LAST, to attach a workspace after the `EXPRESS` workspace. When specifying BEFORE puts the workspace first, the workspace becomes the current one.

The order of the workspace list determines the order in which workspaces will be searched when Oracle OLAP looks for programs or objects named in programs.

PASSWORD *password*

Specifies a password to be checked in a permission program in order to give or deny access to the workspace being attached. See "Permission Programs" on page 8-40.

Notes

Attaching Many Workspaces

Attaching more than one workspace to your session provides access to programs and data in all of the attached workspaces. You can look at and change data or edit programs in any of the workspaces. As long as the workspace is not attached read-only, you can update your changes.

Attaching Many Workspaces: Naming Objects Naming objects requires more care when you attach more than one workspace. When you request an object by name, either with a DESCRIBE statement or by referring to it in a statement or program, Oracle OLAP searches all the active workspaces in order until it finds the named object. When you intend to use several workspaces together, do not give the same name to objects in different workspaces, unless you are prepared to use qualified object names.

Attaching Many Workspaces: LIST Keyword The names of all attached workspaces are kept on the workspace list. You can view the list using AW LIST.

Attaching in Multiwriter Mode

When you are attached in multiwriter mode, you use the OLAP DML statements listed in Table 8–1 to manipulate analytic workspace objects.

Table 8–1 Statements for Managing Objects When Attached in Multiwriter Mode

Statement	Description
ACQUIRE	When attached in multiwriter mode, acquires and (optionally) resynchronizes the specified objects so that their changes can be updated and committed.
RELEASE	When attached in multiwriter mode, changes the access mode of the specified variables, relations, valuesets, or dimensions from read/write (acquired) access to read-only access.
RESYNC	When attached in multiwriter mode, drops private changes for the specified variables, relations, valuesets, and dimensions and promotes them so that the user now sees the data from the latest visible generations.

Table 8–1 (Cont.) Statements for Managing Objects When Attached in Multiwriter

Statement	Description
REVERT	When attached in multiwriter mode, drops all changes made to the specified objects since they were last updated, resynchronized, or acquired, or since the analytic workspace was attached .

Programs Executed When Attaching Analytic Workspaces

When you attach a workspace, Oracle OLAP looks for and executes the following programs in the order indicated:

1. Onattach program. A program that Oracle OLAP looks for and executes when you attach an analytic workspace using an AW ATTACH statement in either of the following situations:

   - When you attach an analytic workspace that contains a program named ONATTACH and you do not include the NOONATTCH keyword in the AW statement or when the AW statement includes an ONATTACH clause that does not specify a program name, Oracle OLAP executes the `ONATTACH` program.

   - When the AW statement includes an ONATTCH clause that specifies a program name, Oracle OLAP looks in the workspace for a program of that name. When it exists, Oracle OLAP executes that program.

2. Permission programs. Programs that Oracle OLAP looks for and executes varies depending on the attachment mode specified in the AW ATTACH statement:

   a. When you request that an analytic workspace be attached in read-only mode, Oracle OLAP checks for a program named PERMIT_READ.

   b. When you request that an analytic workspace be attached in exclusive or non-exclusive read/write mode, Oracle OLAP checks for a program named PERMIT_WRITE.

3. Autogo program. A program that Oracle OLAP looks for and executes when you attach an analytic workspace using an AW ATTACH statement with the AUTOGO clause.

4. TRIGGER_AW program. A trigger program that you create and that Oracle OLAP checks for by name when an AW command executes.

Using ATTACH on an Already-Attached Workspace

Reattaching an attached workspace with a AW ATTACH *workspace* statement does not cause Oracle OLAP to bring a new copy of the workspace into working memory. Instead, Oracle OLAP takes the following actions:

1. Makes the workspace the current workspace.
2. Runs an Autogo program, when you specify the AUTOGO keyword

However, when you have made any changes to data during the session, they are *not* discarded when you reattach an active workspace. Furthermore, current aliases for the workspace are *not* changed.

Conflicts between Workspace Names and Aliases

You cannot attach a workspace that is in your schema and whose name is the same as an assigned alias. Similarly, you cannot assign an alias that duplicates the name of an attached workspace that is in your schema. Furthermore, you cannot assign the same alias to two attached workspaces.

In an AW DELETE statement, when you specify a workspace name (for a workspace that is not attached) and the name is the same as an assigned alias, Oracle OLAP interprets the name as an alias and reports an error.

EXPRESS Workspace

When your database is installed with the OLAP option, the `EXPRESS` workspace is always attached in read-only mode in your session. It never automatically becomes the current workspace, even when it is the first or only workspace in your workspace list, because it is for internal use by Oracle OLAP. You can make the `EXPRESS` workspace the current workspace by explicitly attaching it, but this is not recommended. You cannot detach the `EXPRESS` workspace.

Permission Programs

Keep the following points in mind when working with permission programs.

Specifying Permission to Access Workspace Objects You can specify permission to access workspace objects with PERMIT statements. You can specify PERMIT statements, and the values of the permission conditions on which permission is based, in the workspace permission programs PERMIT_READ and PERMIT_WRITE. All the objects referred to in the workspace permission programs or in the permission expressions must exist within the same workspace. (See PERMIT.)

Permission Programs: Naming You create the workspace permission programs as user-defined Boolean functions in the workspace to which you want to control access. PERMIT_READ must be the name of the program for attaching read-only. PERMIT_WRITE must be the name of the program for attaching read/write. When a workspace permission program executes, it must return `YES` in order for the workspace to be attached.

Permission Programs: In Different Workspaces When you have workspace permission programs defined in workspaces that are currently attached, Oracle OLAP executes the one in the workspace that you are attaching. However, when you have workspace permission programs in more than one currently attached workspace, you must take special care when you edit them or use them in any other way, to ensure that you access the appropriate version.

Permission Programs: Running When you specify a password when attaching the workspace, it is passed as an argument to the workspace permission program.

Permission Programs: Copying to and from Analytic Workspaces When you export PERMIT_READ or PERMIT_WRITE programs which are hidden, they are empty when imported. Additionally, when you outfile PERMIT_READ or PERMIT_WRITE programs which are hidden, then they are empty when infiled.

> **Tip:** Rename PERMIT_READ and PERMIT_WRITE programs before using EXPORT (to EIF) or OUTFILE After copying the programs to an analytic workspace using IMPORT (from EIF) or INFILE.

Read-Only Workspaces

To protect a workspace from inadvertent changes, you can specify RO access when attaching it. You can use a read-only workspace in the same way as an ordinary workspace; you can even make changes to it during your session. However, you cannot save the changes in your session by updating. The UPDATE will have no effect. This protects data you are sure you do not want to change.

Workspace Sharing

Unless the workspace is already attached exclusive and your user ID has the appropriate access rights, you can get read-only access to a workspace, no matter how many other users are using it. When another user has read/write access and uses the UPDATE and COMMIT statements, your view of the workspace does not change. However, you can access their committed changes by detaching the workspace and attaching it again.

Examples

Example 8–14 Startup Programs

Assume that you have created an analytic workspace named `awtest` that contains five programs named `PERMIT_READ`, `PERMIT_WRITE`, `ONATTACH`, `MYATTACH`, and `AUTOGO` that have the following definitions.

```
DEFINE PERMIT_READ PROGRAM BOOLEAN
PROGRAM
SHOW 'permit_read program executing'
AW LIST
RETURN YES
END

DEFINE PERMIT_WRITE PROGRAM BOOLEAN
PROGRAM
SHOW 'permit_write program executing'
AW LIST
RETURN YES
END

DEFINE ONATTACH PROGRAM BOOLEAN
PROGRAM
SHOW 'onattach program executing'
AW LIST
RETURN YES
END

DEFINE MYATTACH PROGRAM BOOLEAN
PROGRAM
SHOW 'myattach program executing'
AW LIST
RETURN YES
END

DEFINE AUTOGO PROGRAM
PROGRAM
SHOW 'autogo program executing'
AW LIST
END
```

The programs that execute when you attach `awtest` vary depending on the attachment mode and keywords in the AW ATTACH statement:

- When you attach `awtest` in read/write mode using the following statements.

  ```
  AW DETACH awtest
  AW ATTACH awtest RW
  ```

  First the `PERMIT_WRITE` program executes, and then the `ONATTACH` program executes.

  ```
  permit_write program executing
  AWTEST    R/W CHANGED   XUSER.AWTEST
  EXPRESS   R/O UNCHANGED SYS.EXPRESS
  onattach program executing
  AWTEST    R/W CHANGED   XUSER.AWTEST
  EXPRESS   R/O UNCHANGED SYS.EXPRESS
  ```

- When you attach `awtest` in read-only mode using the following statements.

  ```
  AW DETACH axuserwtest
  AW ATTACH awtest NOONATTACH RO
  ```

  Only the `PERMIT_READ` program executes.

  ```
  permit_read program executing
  AWTEST    R/O UNCHANGED XUSER.AWTEST
  EXPRESS   R/O UNCHANGED SYS.EXPRESS
  ```

- When you attach `awtest` in read-only mode using the following statements.

  ```
  AW DETACH awtest
  AW ATTACH awtest RO
  ```

  First the `PERMIT_READ` program executes, and then the `ONATTACH` program executes.

  ```
  permit_read program executing
  AWTEST    R/O CHANGED   XUSER.AWTEST
  EXPRESS   R/O UNCHANGED SYS.EXPRESS
  onattach program executing
  AWTEST    R/O CHANGED   XUSER.AWTEST
  EXPRESS   R/O UNCHANGED SYS.EXPRESS
  ```

- When you attach `awtest` in read-only mode using the following statements.

  ```
  AW DETACH awtest
  AW ATTACH awtest ONATTACH myattach RO
  ```

  First the `PERMIT_READ` program executes, and then the `MYATTACH` program executes.

  ```
  permit_read program executing
  AWTEST    R/O CHANGED   XUSER.AWTEST
  EXPRESS   R/O UNCHANGED SYS.EXPRESS
  myattach program executing
  AWTEST    R/O CHANGED   XUSER.AWTEST
  EXPRESS   R/O UNCHANGED SYS.EXPRESS
  ```

- When you attach `awtest` in multi mode using the following statements.

  ```
  AW DETACH awtest
  AW ATTACH awtest MULTI
  ```

  First the `PERMIT_WRITE` program executes, and then the `ONATTACH` program executes.

  ```
  permit_write program executing
  AWTEST    R/M CHANGED   XUSER.AWTEST
  EXPRESS   R/O UNCHANGED SYS.EXPRESS
  onattach program executing
  AWTEST    R/M CHANGED   XUSER.AWTEST
  EXPRESS   R/O UNCHANGED SYS.EXPRESS
  ```

- When you attach `awtest` in read-only mode using the following statements.l

  ```
  AW DETACH awtest
  AW ATTACH awtest AUTOGO
  ```

  First the `PERMIT_WRITE` program executes. Secondly, the `ONATTACH` program executes. Finally, the `AUTOGO` program executes.

  ```
  permit_write program executing
  AWTEST    R/O UNCHANGED XUSER.AWTEST
  EXPRESS   R/O UNCHANGED SYS.EXPRESS
  onattach program executing
  AWTEST    R/O UNCHANGED XUSER.AWTEST
  EXPRESS   R/O UNCHANGED SYS.EXPRESS
  autogo program executing
  AWTEST    R/O UNCHANGED XUSER.AWTEST
  EXPRESS   R/O UNCHANGED SYS.EXPRESS
  ```

Example 8–15 Attaching an Analytic Workspace Using an ONATTACH Program

Suppose you have two workspaces of sales data, one for `expenses` and one for `revenue`. You have a third workspace called `analysis` contains programs to analyze the data. Your `analysis` workspace has the following `ONATTACH` program to attach the other two.

```
DEFINE onattach PROGRAM
PROGRAM
AW ATTACH expenses RW AFTER analysis
AW ATTACH revenues RW AFTER analysis
END
```

To run the `ONATTACH` program, attach the `analysis` workspace with the following statement.

```
AW ATTACH analysis
```

When you issue an AW LIST statement, you can see from the following output, that all three of your analytic workspaces are attached.

```
ANALYSIS  R/W CHANGED   XUSER.ANALYSIS
REVENUE   R/W UNCHANGED XUSER.REVENUES
EXPENSES  R/W UNCHANGED XUSER.EXPENSES
EXPRESS   R/O UNCHANGED SYS.EXPRESS
```

AW CREATE

The AW CREATE command creates a new workspace and make it the current workspace in your session. It is important to note that Oracle OLAP automatically executes a COMMIT as part of its procedure for creating a workspace. Previously attached workspaces move down in the list of attached workspaces to make room for the new one at the top of the list. Oracle suggests that you use the TABLESPACE argument to create your workspace in a tablespace that has been prepared for this purpose. Ask your DBA which tablespace you should use.

Note: When a program named TRIGGER_AW exists in the analytic workspace, the execution of an AW CREATE statement automatically executes that program.

Syntax

AW CREATE *workspace* [*position*] [UNPARTITIONED|<u>PARTITIONS</u> *n*] -

    [SEGMENTSIZE *n* [K, M, or G]] [TABLESPACE *tblspname*]

where *position* specifies the workspace's position in the workspace list and is one of the following values. (FIRST is the default.)

AFTER *workspace*
BEFORE *workspace*
LAST
FIRST

Arguments

workspace
The name of the analytic workspace. You must specify the name. You cannot specify an alias because no alias is assigned when you are creating. When you create a workspace, keep in mind the following rules for workspace names: Workspace names can contain only letters, numerals, and underscores; they cannot begin with a numeral; they cannot be reserved words in the DML; and they can be no more than 26 characters in length. (Use RESERVED to identify reserved words.) All characters must come from the database character set.

FIRST
Makes the workspace you are attaching the current workspace. (Default)

LAST
Puts the workspace after the current workspace and before the `EXPRESS` workspace. When there are other workspaces attached before the `EXPRESS` workspace, the specified workspace is attached after them. When there are no workspaces before the `EXPRESS` workspace, LAST makes the specified workspace the current one. LAST ignores any workspaces after the `EXPRESS` workspace.

AFTER
BEFORE
Specify the position of the newly attached workspace relative to a workspace that is already attached. Use AFTER, rather than LAST, to attach a workspace after the `EXPRESS` workspace. When specifying BEFORE puts the workspace first, the workspace becomes the current one.

The order of the workspace list determines the order in which workspaces will be searched when Oracle OLAP looks for programs or objects named in programs.

UNPARTITIONED
Specifies that the relational table that is the analytic workspace is not a partitioned table.

partitionS *n*
Specifies that the relational table that is the analytic workspace is a hash partitioned table with *n* partitions. Specifying a value of 0 (zero) for *n* is the same as specifying UNPARTITIONED. The default value of *n* is 8.

SEGMENTSIZE *n* [K, M, or G]
With the CREATE keyword, this argument sets the maximum size of each segment for the workspace being created. When you do not specify `K`, `M`, or `G`, the value you specify for *n* is interpreted as bytes. When you specify `K`, `M`, or `G` after the value *n*, the value is interpreted as kilobytes, megabytes, or gigabytes, respectively.

TABLESPACE *tblspname*
Specifies the name of an Oracle Database tablespace in which the analytic workspace is created.

Notes

Security and Permissions

You can add security to analytic workspaces at several levels:

- At the relational table level.
- At the analytic workspace level and workspace object level as described in "Restricting Accessing Using Analytic Workspace Attachment Modes" on page 8-49.
- At the analytic workspace object level and value level as described in "Restricting Access Using OLAP DML Permission Programs" on page 8-49.

Granting Access Using SQL Statements An analytic workspace is a multidimensional data source that is stored as a relational table of LOBs. The name of the relational table that is the analytic workspace is `AW$` followed by the OLAP DML name of the analytic workspace.

When you first create an analytic workspace using an OLAP DML AW CREATE statement, you are the only user who has access that workspace. When you want others to use the workspace, you must give them access to the relational table that is the analytic workspace use an SQL `GRANT` statement:

- To give read access to another user, execute a statement like the following one in SQL. In this example, the workspace name is `demo` and the user's name is `Scott`.

  ```
  GRANT SELECT ON aw$demo TO Scott
  ```

- To give write access to another user, execute a SQL statement like the following one.

  ```
  GRANT UPDATE ON aw$demo TO Scott
  ```

As in any SQL `GRANT` statement, you can specify a group or role instead of a user.

Restricting Accessing Using Analytic Workspace Attachment Modes When you attach an analytic workspace using the AW ATTACH statement, the mode that you attach it in determines the access that have to the analytic workspace objects:

- **Read-only mode** — When an analytic workspace is attached in read-only access mode, users can make private changes to the data in the workspace to perform what-if analysis, but cannot commit any of these changes.
- **Read/write nonexclusive mode**—Only one user can have an analytic workspace attached in read/write nonexclusive mode at a time. The user has to commit either all or none of the changes made to the workspace.
- **Read/write exclusive mode**—Only one user can have an analytic workspace attached in read/write exclusive at a time. The user has to commit either all or none of the changes made to the workspace.
- **Multiwriter mode** —A workspace that is attached in multiwriter mode can be accessed simultaneously by several sessions. In multiwriter mode, users can simultaneously modify the same analytic workspace in a controlled manner by specifying the attachment mode (read-only or read/write) for individual objects.

  When users first attach an analytic workspace in multiwriter mode, all objects in the workspace are read-only. Users can make private changes to the data in the workspace to perform what-if analysis but you cannot update or commit any of these changes. Using the OLAP DML statements described in Table 8–1, " Statements for Managing Objects When Attached in Multiwriter Mode" on page 8-38, users manipulate the attachment mode of individual objects (for example, change the a variable from read-only to write access).

Restricting Access Using OLAP DML Permission Programs Permission programs are programs that you write that give permission to users to access workspace data. Permission programs do not exist within an analytic workspace unless you define and write them as described in "Permission Programs" on page 1-12.

When a user attaches an analytic workspace, Oracle OLAP checks to see if a permission program that is appropriate for the attachment mode exists. (The permission program for each attachment mode must have a particular name as outlined in Table 8–2, " Names of Permission Programs for Different Attachment Modes".) When an appropriate permission program exists, Oracle OLAP executes the program. When a user specifies a password when attaching the analytic workspace, then the password is passed as an argument to the permission program for processing.

Table 8–2 Names of Permission Programs for Different Attachment Modes

Attachment Modes	Name of Program
Multiwriter, Read-only, and Read/write	ONATTACH
Read-only	PERMIT_READ
Read/write	PERMIT_WRITE

Note: A dimension surrogate has the access permissions of its dimension. Use a PERMIT on a dimension to grant or deny permission to access the values of a dimension surrogate for that dimension.

Permission programs are not the only programs that are executed when a user attaches to an analytic workspace. For more information, see "Startup Programs" on page 1-11.

Examples

Example 8–16 Creating and Starting a Workspace

You can use the AW command with the CREATE keyword to create and start a new workspace.

```
AW CREATE mywork
```

AW DELETE

The AW DELETE command deletes the specified workspace from the database. It is important to note that Oracle OLAP automatically executes a COMMIT as part of its procedure for deleting a workspace. The DELETE keyword executes successfully only when no user has the workspace attached. Therefore, detach the workspace before executing this statement.

Note: When a program named TRIGGER_AW exists in the analytic workspace, the execution of an AW DELETE statement automatically executes that program.

Syntax

AW DELETE *workspace*

Arguments

workspace
The name of the analytic workspace. You must specify the name; you cannot specify an alias.

Notes

Deleting an Unattached Workspace

When you attempt to delete an unattached workspace and the name is the same as an assigned alias, Oracle OLAP interprets the name as an alias and reports an error.

Examples

Example 8–17 Deleting a Workspace

You can use the AW command with the DELETE keyword to delete a workspace.

```
AW DELETE mywork
```

AW DETACH

The AW DETACH command removes a workspace from the workspace list. When you remove the first workspace, the second workspace becomes the current workspace (unless it is the `EXPRESS` workspace). When you detach a workspace, changes that were made before an UPDATE was issued remain in the database and become permanent with the next COMMIT. When changes were made after the UPDATE was issued, they are discarded.

Note: When a program named TRIGGER_AW exists in the analytic workspace, the execution of an AW DETACH statement automatically executes that program.

Syntax

AW DETACH *workspace*

Arguments

workspace
The name of the analytic workspace. You can specify either a workspace name or a workspace alias, depending on the keywords you are using.

Notes

EXPRESS Workspace

You cannot detach the `EXPRESS` workspace.

Examples

Example 8–18 Detaching a Workspace

You can use the AW command with the DETACH keyword to detach a workspace.

```
AW DETACH expense
```

AW LIST

The AW LIST command sends to the current outfile a list of the active workspaces, along with their update status.

Syntax

```
AW LIST
```

Notes

Output Produced by AW LIST

The first workspace in the list is the current workspace, unless you do not have a current workspace. The meaning of the update status, CHANGED or UNCHANGED, depends on whether the workspace is attached with read/write or read-only access and whether or not the workspace is being shared with other users. The update status displayed by AW LIST is as follows:

- An unshared workspace in read/write mode -- The update status is CHANGED when you have made changes since attaching the workspace or since your last update.
- An unshared workspace in read-only mode -- The status is always UNCHANGED because you cannot update it.
- A shared workspace in read/write mode -- The status is CHANGED when you have made changes since attaching the workspace or since your last update. This is the same as for an unshared workspace in read/write mode.
- A shared workspace in read-only mode -- The status is CHANGED when another user has updated it since you accessed it. To access the new objects or data, you must detach and reattach the workspace after the other user commits his or her changes. As long as you keep the workspace attached, your view of the workspace remains unchanged.

Current Workspace

The name of the current workspace is first on the workspace list and is the name returned by the AW(NAME) function. (See AW function for details.) The NAME dimension includes only the objects in the current workspace. Programs such as AWDESCRIBE and LISTBY list only objects in the current workspace. When a workspace is active but not current, you can change and update its data, edit and run its programs, and modify its objects.

EXPRESS Workspace

When your database is equipped with the OLAP option, the `EXPRESS` workspace is always attached in read-only mode in your session. It never automatically becomes the current workspace, even when it is the first or only workspace in your workspace list, because it is for internal use by Oracle OLAP. You can make the `EXPRESS` workspace the current workspace by explicitly attaching it, but this is not recommended. You cannot detach the `EXPRESS` workspace.

Examples

Assume that you have just connected to Oracle OLAP using the OLAP Worksheet. You issue an `AW LIST` statement that returns a value showing that the only attached analytic workspace is `EXPRESS`.

```
AW LIST
EXPRESS   R/O UNCHANGED SYS.EXPRESS
```

Now you create a new analytic workspace and issue another `AW LIST` statement. You can see that both the `EXPRESS` analytic workspace and the newly created analytic workspace are attached.

```
AW CREATE myaw
AW LIST
MYAW      R/W UNCHANGED MYNAME.MYAW
EXPRESS   R/O UNCHANGED SYS.EXPRESS
```

AW SEGMENTSIZE

The AW SEGMENTSIZE command sets up a workspace for multiple segments.

Syntax

AW SEGMENTSIZE *n* [K, M, or G] [*workspace*]

Arguments

workspace
The name of the analytic workspace. You can specify either a workspace name or a workspace alias, depending on the keywords you are using.

SEGMENTSIZE *n* [K, M, or G] [*workspace*]
Sets the maximum size of each segment for a specified workspace or, when no workspace is specified, for the current workspace.

When the current workspace already has several segments, setting SEGMENTSIZE affects only the most recent one and has no effect on previous ones. Previous segments may have various sizes, determined by the SEGMENTSIZE setting at the time each one was created. When you do not specify `K`, `M`, or `G`, the value you specify for *n* is interpreted as bytes. When you specify `K`, `M`, or `G` after the value *n*, the value is interpreted as kilobytes, megabytes, or gigabytes, respectively.

AW function

The AW function returns information about currently attached workspaces.

Return Value

The return value depends on the keyword you specify, as described in Table 8–3, " Keywords for AW Function" on page 8-56.

Syntax

AW(*keyword* [*workspace*])

Arguments

keyword

Indicates the specific information you want. The keywords that you can use with the AW function are listed in Table 8–3, " Keywords for AW Function" with the data type of the value they return and the meaning of the information.

Table 8–3 Keywords for AW Function

Keyword	Type	Information Returned
ACQUIRED	TEXT	When an analytic workspace is attached in multiwriter mode, returns the names of any acquired variables and dimensions in the analytic workspace
AGGMAP	TEXT	A list of all aggmap objects in the workspace. When there are several, Oracle OLAP returns a multiline text value with each object name on a separate line.
ALIASLIST	TEXT	A list of currently assigned aliases for the workspace. When there are several, Oracle OLAP returns a multiline text value with each alias on a separate line.
ATTACHED	BOOLEAN	Indicates whether the specified workspace is attached. The workspace argument is required.
CHANGED	BOOLEAN	When you have read/write access to the workspace, indicates whether you have made changes since the last time the workspace was updated. When you have read-only access to the workspace, indicates whether another user has updated the workspace and committed the changes since you attached it.

Table 8–3 (Cont.) Keywords for AW Function

Keyword	Type	Information Returned
COMPOSITE	TEXT	A list of all named composite objects in the specified workspace.
DATE	DATE	The date of your most recent update in the current session. When you have not updated in the current session, it returns the date of the most recent commit before you attached the workspace. When you have attached a shared workspace as read-only, DATE does not take into account any updates or commits that have occurred since you attached the workspace.
DIMENSION	TEXT	A list of all the dimensions defined in the workspace. When there are several dimensions, Oracle OLAP returns a multiline text value with each dimension name on a separate line.
EXISTS	BOOLEAN	Indicates whether the specified analytic workspace has been defined in the Oracle Database.
FORMULA	TEXT	A list of all the formulas defined in the workspace. When there are several formulas, Oracle OLAP returns a multiline text value with each formula name on a separate line.
FULLNAME	TEXT	The full name of the specified workspace. The full name includes the schema that contains the workspace.
ISUPDATED	TEXT	When the specified analytic workspace is not attached in multiwriter mode, returns TRUE when the workspace is updated but not committed. When he specified analytic workspace is attached in multiwriter mode, returns TRUE when at least one variable or dimension belonging to the workspace is updated but not committed.
LIST	TEXT	A list of all currently attached workspaces. Each line of the multiline text value contains the name of a workspace.
LISTNAMES	TEXT	A list of all the objects defined in the workspace. Each line of the multiline text value contains the name of a workspace object.
MODEL	TEXT	A list of all the models defined in the workspace. When there are several models, Oracle OLAP returns a multiline text value with each model name on a separate line.

Table 8–3 (Cont.) Keywords for AW Function

Keyword	Type	Information Returned
MULTI	TEXT	Indicates if you have multi-writer access to the analytic workspace.
NAME	TEXT	The name of the current workspace.
OPTION	TEXT	A list of all the Oracle OLAP options defined in the `EXPRESS` workspace. When the workspace is not EXPRESS, `AW(OPTION)` returns `NA`, because options are defined only in the `EXPRESS` workspace. For the `EXPRESS` workspace, `AW(OPTION)` returns a multiline text value with each option name on a separate line.
PROGRAM	TEXT	A list of all the programs defined in the workspace. When there are several programs, Oracle OLAP returns a multiline text value with each program name on a separate line.
RELATION	TEXT	A list of all the relations defined in the workspace. When there are several relations, Oracle OLAP returns a multiline text value with each relation name on a separate line
RO	TEXT	Indicates whether you have read-only access to the workspace.
RW	TEXT	Indicates whether you have read/write access to the workspace.
SEGMENTSIZE	DECIMAL	The current maximum segment size for the workspace. It is the most recent value specified using an `AW SEGMENTSIZE` statement.
SHARED	BOOLEAN	Indicates whether the workspace is being shared by other users.
TIME	ID	The time of your most recent update in the current session. When you have not updated in the current session, it returns the time of the most recent commit before you attached the workspace. When you have attached a shared workspace as read-only, TIME does not take into account any updates or commits that have occurred since you attached the workspace.
VALUSET	TEXT	A list of all the valuesets that are defined in the workspace. When there are several valuesets, Oracle OLAP returns a multiline text value with each valueset name on a separate line.

Table 8–3 (Cont.) Keywords for AW Function

Keyword	Type	Information Returned
VARIABLE	TEXT	A list of all the variables defined in the workspace. When there are several variables, Oracle OLAP returns a multiline text value with each variable name on a separate line.
WORKSHEET	TEXT	A list of all the worksheet objects defined in the workspace. When there are several worksheets, Oracle OLAP returns a multiline text value with each worksheet name on a separate line. This keyword will not be available in the Oracle 10i release and later.

workspace

A text expression that indicates the name of the workspace for which you want information. When you do not specify this argument, the AW function ordinarily returns information about the current workspace. The ATTACHED, LIST, and NAME keywords are exceptions to this rule.

Notes

Status Information

You can use the SHARED, CHANGED, RO, and RW keywords to get information about the current status of a shared workspace. You can check if SHARED, RO, and CHANGED are `TRUE` to find out if another user has updated a workspace since you attached it.

Examples

Example 8–19 Ascertaining the Active Workspace

The following program line checks which workspace is currently active so the program can choose the appropriate data to report. With this method, you can use the same report program in several workspaces, each containing different data.

```
REPORT IF AW(NAME) EQ 'mysales' THEN mysales ELSE gensales
```

AWDESCRIBE

The AWDESCRIBE program sends information about the current analytic workspace to the current outfile. After a summary page, it provides a report in two parts:

- An alphabetic list of analytic workspace objects showing name, type, and description.
- A list of object definitions by object type. Each definition includes the information you would see when you used the DESCRIBE statement. It also includes a "Referenced By" list, which indicates any programs or other compilable objects that call or access the object. In addition, compilable objects have a "References To" list, indicating the analytic workspace objects that they call or access.

Syntax

```
AWDESCRIBE
```

Notes

Information in Referenced By List

The AWDESCRIBE command does not provide information in the "Referenced By" and "References To" list for implicit references. For example: When a program contains a LIMIT command to limit a dimension by a related dimension, AWDESCRIBE does not list the relation for those dimensions in the "References To" list for that program.

Examples

Example 8–20 Describing a Workspace

The following example shows a portion of the output of AWDESCRIBE for a workspace named demo.

```
                             DEMO Workspace Listing
                             =====================

Last updated:  25Jun96     Time: 09:46:50
Print date:    27Aug96     Time: 10:30:11
DEMO contains:
   11 DIMENSIONS
   19 VARIABLES
    1 PROGRAM
    4 RELATIONS
    2 VALUESETS

This report is in two parts:
    - Object Listing: An alphabetic list of workspace objects,
      beginning on the next page.
    - Object Descriptions: Detailed descriptions of all workspace
      objects, sorted by object type and alphabetically by name.

Object List                                                        Page 2
Workspace: DEMO         Updated: 25Jun96   At: 09:46:50             ACTUAL

NAME              TYPE       DESCRIPTION

____              ____       ___________
ACTUAL            VARIABLE   Actual $ Financials
ADVERTISING       VARIABLE   Total Advertising Dollars
BUDGET            VARIABLE   Budgeted $ Financials
CHOICE            DIMENSION  List of choices
CHOICEDESC        VARIABLE   Description line for the choices
DEMOVER           VARIABLE   DEMO Workspace Version
DISTRICT          DIMENSION
DIVISION          DIMENSION  Division
DIVISION.PRODUCT  RELATION   DIVISION for each PRODUCT
EXPENSE           VARIABLE   Total Production & Distribution Cost
FCST              VARIABLE   Forecasted $ Financials
INDUSTRY.SALES    VARIABLE   Total Industry Sales Revenue
LINE              DIMENSION  Lineitem
MARKET            DIMENSION  Geography Dim with Embedded Totals
MARKET.MARKET     RELATION   Self-relation for the Market Dim
```

```
MARKETLEVEL       DIMENSION  Geography Level
MLV.MARKET        RELATION
MONTH             DIMENSION
NAME.LINE         VARIABLE   Lineitem Names for Reporting
NAME.PRODUCT      VARIABLE   Product Names for Reporting Purposes
NATIONAL.SALES    VARIABLE   Projected Total U.S. Dollar Sales
NOT.IMPLEMENTED   PROGRAM
PRICE             VARIABLE   Wholesale Unit Selling Price
PRODUCT           DIMENSION  Sporting Goods Products
PRODUCT.MEMO      VARIABLE   Product Analysis Memo
PRODUCTSET        VALUESET   Valueset for Sporting Goods Products
QUARTER           DIMENSION
QUARTERSET        VALUESET
REGION            DIMENSION  Sales Region
REGION.DISTRICT   RELATION   REGION for each DISTRICT
SALES             VARIABLE   Sales Revenue
SALES.FORECAST    VARIABLE   Forecasted Unit Sales
SALES.PLAN        VARIABLE   Budgeted Sales Revenue
SHARE             VARIABLE   Market Share (Based on Dollar Sales)
UNITS             VARIABLE   Actual Unit Shipments
UNITS.M           VARIABLE
YEAR              DIMENSION

Description of DIMENSIONS                                       Page 3
Workspace: DEMO        Updated: 25Jun96   At: 09:46:50           CHOICE

DEFINE CHOICE DIMENSION TEXT
LD List of choices
     Referenced By:
         NONE

DEFINE DISTRICT DIMENSION TEXT
     Referenced By:
         NONE

DEFINE DIVISION DIMENSION TEXT
LD Division
     Referenced By:
         NONE
...
```

AWWAITTIME

The AWWAITTIME option holds the number of seconds that a AW ATTACH command with the WAIT keyword waits for an analytic workspace to become available for access. The default value of AWWAITTIME is 20 seconds.

Data type

INTEGER

Syntax

AWWAITTIME = *seconds*

Arguments

seconds
The number of seconds to wait for an analytic workspace to be available.

Notes

Workspace Sharing

When your user ID has the appropriate access rights and that no user has read/write exclusive access to the workspace, you can get read-only access to a workspace, no matter how many other users are using it. When another user has read/write access and commits the workspace, your view of the workspace does *not* change; you must detach and reattach the workspace to see the changes.

AW WAIT or NOWAIT

When attaching a workspace using the AW ATTACH command, you can specify the WAIT or NOWAIT keyword. When you request access to a workspace that is being used by another session, NOWAIT (the default) causes an error message to be produced indicating that the workspace is unavailable. To wait for the workspace to become available for access, use WAIT. The number of seconds to wait is determined by the value of the AWWAITTIME option.

BACK

The BACK function returns the names of all currently executing programs, listed one name on each line in a multiline text value. When more than one program is executing, this means that one program has called another in a series of nested executions.

The first name in the return value is that of the program containing the call to BACK. The last name is that of the initial program, which made the first call to another program.

BACK can only be used in a program.

Return Value

TEXT

Syntax

```
BACK
```

Notes

BACK Function and BACK Command

In previous releases, the OLAP DML included a command named BACK, which worked only in the interactive debugging facility through OLAP Worksheet. This BACK command is not currently available, but the BACK function (documented here) is available.

Examples

Example 8–21 Debugging a Program Using the BACK Function

The following example uses three programs. `program1` calls `program2`, and `program2` calls `program3`.

```
DEFINE program1 PROGRAM
PROGRAM
SHOW 'This is program number 1'
CALL program2
END
DEFINE program2 PROGRAM
PROGRAM
SHOW 'This is program number 2'
CALL program3
END
DEFINE program3 PROGRAM
PROGRAM
SHOW 'This is program number 3'
SHOW 'These programs are currently executing:'
SHOW BACK
END
```

Executing `program1` produces the following output.

```
This is program number 1
This is program number 2
This is program number 3
These programs are currently executing:
PROGRAM3
PROGRAM2
PROGRAM1
```

BADLINE

When a program, model, or input file is executing, the BADLINE option controls whether Oracle OLAP records, in the current outfile, the line that caused an error.

See also: PROGRAM, MODEL, and INFILE.

Data type

BOOLEAN

Syntax

BADLINE = {YES|NO}

Arguments

YES
When an error occurs during the execution of a program, model, or input file, Oracle OLAP records in the current outfile the name of the program, model, or file in which the error occurred and the line that caused the error. When an error message is included in the output, the BADLINE information appears immediately after the error message.

NO
When an error occurs in a program, model, or input file, Oracle OLAP does not record the error in the current outfile. (Default)

Examples

Example 8–22 Using the BADLINE Option

In a simple program called `test`, the variable `myint1` is divided by zero.

```
DEFINE test PROGRAM
PROGRAM
VARIABLE myint1 INTEGER
VARIABLE myint2 INTEGER
myint1 = 0
myint2 = 250/myint1
END
```

When you run the program when the `DIVIDEBYZERO` option is set to `NO`, then an error occurs because division by zero is not allowed. When `BADLINE` is set to `YES`, the following messages are recorded in the current outfile.

```
ERROR: (MXXEQ01) A division by zero was attempted. Set DIVIDEBYZERO to
YES if you want NA to be returned as the result of division by zero.
In DEMO!TEST PROGRAM:
myint2 = 250/myint1
```

Example 8–23 Finding Errors in Program Lines

In a simple program called `test`, the variable `myint1` is divided by `0` (zero).

```
DEFINE test PROGRAM
PROGRAM
VARIABLE myint1 INTEGER
VARIABLE myint2 INTEGER
myint1 = 0
myint2 = 250/myint1
END
```

When you run the program, an error occurs because division by zero is not allowed (that is, when DIVIDEBYZERO is set to `NO`).

When BADLINE is set to `NO` only the error is recorded in the current outfile.

```
ERROR: (MXXEQ01) A division by zero was attempted.  (If you want NA to
be returned as the result of a division by zero, set the DIVIDEBYZERO
option to YES.)
```

When BADLINE is set to `YES`, the line that causes the error and the name of the program in which the error occurred are recorded in the current outfile.

```
ERROR: (MXXEQ01) A division by zero was attempted.  (If you want NA to
be returned as the result of a division by zero, set the DIVIDEBYZERO
option to YES.)
In TESTBAD PROGRAM:
myint2 = 250/myint1
In EDDE.RUNCMD PROGRAM:
```

BASEDIM

The BASEDIM function returns the name of the dimension from which the current value of a concat dimension comes.

Return Value

```
TEXT
```

Syntax

BASEDIM(*concatdim* [LEAF])

Arguments

concatdim
Specifies the concat dimension for which you want the names of the base or component dimensions. The data type of the values returned is TEXT.

LEAF
The LEAF keyword causes BASEDIM to return the names of the component dimensions of the *concatdim* dimension. See "Using the LEAF Keyword" on page 8-68.

Notes

Using the LEAF Keyword

The *base* dimensions of a concat dimension are the simple, conjoint, or other concat dimensions that you specify with the *basedimlist* argument when you define the concat. Simple dimensions and conjoint dimensions are the bottom-level components, or leaves, of a concat dimension. When you specify a concat dimension as a base dimension when defining a concat, then the base dimensions of that inner concat are *component* dimensions of the outer concat. Using the LEAF keyword results in BASEDIM returning the names of the component simple and conjoint dimensions of the inner concat dimension.

When the base dimensions are all simple dimensions or conjoint dimensions, then the base dimensions are the bottom-level components and therefore BASEDIM returns the names of those dimensions whether or not you use the LEAF keyword.

Looping Over the Dimension

BASEDIM is dimensioned by the *concatdim* dimension so the function automatically loops over the dimension.

Examples

Example 8–24 Returning Base Dimension Names

In this example the `product` dimension is limited to two values, the `district` dimension is limited to its first three values and the `region` dimension has only three values. The example defines a nonunique concat dimension with `region` and `district` as its base dimensions and then defines another nonunique concat dimension with `product` and the first concat dimension as its base dimensions. The example then gets the names of the base dimensions of the outer concat.

```
LIMIT district TO 'Boston' TO 'Chicago'
LIMIT product TO 'Tents''Canoes'
DEFINE region.district DIMENSION CONCAT(region district)
DEFINE product.region.district DIMENSION CONCAT(product region.district)
REPORT BASEDIM(product.district.region)
```

The preceding statements return the following.

```
PRODUCT
PRODUCT
REGION.DISTRICT
REGION.DISTRICT
REGION.DISTRICT
REGION.DISTRICT
REGION.DISTRICT
REGION.DISTRICT
```

Example 8–25 Returning Component Dimension Names

This example uses the same objects as the previous example. It gets the names of the component dimensions of the concat dimension.

```
REPORT BASEDIM(product.region.district LEAF)
```

The preceding statement returns the following.

```
PRODUCT
PRODUCT
REGION
REGION
REGION
DISTRICT
DISTRICT
DISTRICT
```

BASEVAL

The BASEVAL function returns the values of the base dimensions of a concat dimension. When a base dimension is a concat dimension, then the values of its base dimensions are returned, also.

Return Value

The following are the rules that determine the data types of the values returned by BASEVAL:

- The data type of the return value is NTEXT when any of the component dimensions of *concatdim* is of type NTEXT, or when any component dimension is a conjoint that uses a simple dimension of type NTEXT.
- The data type of the return value is the data type of the component dimensions when all of the component dimensions have the same data type and when none of the component dimensions is a conjoint.
- The data type of the return value is TEXT in all other cases.

Syntax

BASEVAL(*concatdim*)

Arguments

concatdim

Specifies the concat dimension for which you want the base values. The data types of the values returned depend on the data types of the base dimensions of the concat dimension.

Notes

Looping Over the Dimension

BASEVAL is dimensioned by the *concatdim* dimension so the function automatically loops over the dimension.

Examples

Example 8–26 Returning NTEXT Values

The following example creates two simple dimensions and a nonunique concat dimension, then gets the values of the concat dimension.

```
DEFINE textdim DIMENSION TEXT
DEFINE ntextdim DIMENSION NTEXT
MAINTAIN textdim ADD 'v1' 'v2'
MAINTAIN ntextdim ADD 'n1' 'n2'
DEFINE concatdim DIMENSION CONCAT(textdim ntextdim)
REPORT w 18 BASEVAL(concatdim)
```

The preceding statement returns the following.

```
CONCATDIM            BASEVAL(CONCATDIM)
-------------------- ------------------
<textdim: v1>        v1
<textdim: v2>        v2
<ntextdim: n1>       n1
<ntextdim: n2>       n2
```

The data type of the returned values is NTEXT. The BASEVAL function converted the `v1` and `v2` TEXT values into NTEXT values before returning them.

Example 8–27 Returning the Base Values of a Base Concat Dimension

This example defines the simple dimensions `state` and `city` and adds values to them. It defines a nonunique concat dimension, `statecity`, with `state` and `city` as the bases and then defines another nonunique concat dimension, `geog`, with `region`, `district`, and `statecity` as its bases. Finally, the REPORT statement returns the values returned by the BASEVAL function.

```
DEFINE city DIMENSION TEXT
DEFINE state DIMENSION TEXT
MAINTAIN city ADD 'Boston' 'Worcester' 'Portsmouth' 'Portland' -
  'Burlington' 'Hartford' 'New York' 'Albany'
MAINTAIN state ADD 'MA' 'NH' 'ME' 'VT' 'CT' 'NY'
DEFINE statecity DIMENSION CONCAT(state city)
DEFINE geog DIMENSION CONCAT(region district statecity)
LCOLWIDTH = 20
REPORT W 16 BASEVAL(geog)
```

The preceding statement returns the following.

```
GEOG                 BASEVAL(GEOG)
-------------------- ----------------
<region: East>       East
<region: Central>    Central
<region: West>       West
<district: Boston>   Boston
<district: Atlanta>  Atlanta
<district: Chicago>  Chicago
<district: Dallas>   Dallas
<district: Denver>   Denver
<district: Seattle>  Seattle
<state: MA>          MA
<state: NH>          NH
<state: ME>          ME
<state: VT>          VT
<state: CT>          CT
<state: NY>          NY
<city: Boston>       Boston
<city: Worcester>    Worcester
<city: Portsmouth>   Portsmouth
<city: Portland>     Portland
<city: Burlington>   Burlington
<city: Hartford>     Hartford
<city: New York>     New York
<city: Albany>       Albany
```

BEGINDATE

The BEGINDATE function returns the beginning date of the first time period for which an expression has a non-`NA` value.

> **Note:** You can only use this function with dimensions of type DAY, WEEK, MONTH, QUARTER, or YEAR. You can*not* use this function for time dimensions that are implemented as hierarchical dimensions of type TEXT.

Return Value

DATE

Syntax

BEGINDATE(*expression*)

Arguments

expression
The expression must have exactly one dimension that has a type of DAY, WEEK, MONTH, QUARTER, or YEAR.

Notes

NA Values

When all the values of the expression are `NA`, BEGINDATE returns `NA`.

How BEGINDATE Works

BEGINDATE returns the first date of the first time period in dimension status for which the expression has a non-`NA` value. For example, assume that an expression is dimensioned by `month`, and that `Jan97` is the first dimension value for which the expression has a non-`NA` value. In this case, BEGINDATE returns the date `January 1, 1997`.

Format of the Date

When you display the result returned by BEGINDATE, Oracle OLAP formats the date according to the date template in the DATEFORMAT option. When the day of the week or the name of the month is used in the date template, Oracle OLAP uses the day names specified in the DAYNAMES option and the month names specified in the MONTHNAMES option. You can use the result returned by BEGINDATE anywhere that a DATE value is expected.

DATE-to-TEXT Conversion

You can also use the result where a text value is expected. Oracle OLAP automatically converts the date to a text value, using the current template in the DATEFORMAT option to format the text value. When you want to override the current DATEFORMAT template, you can convert the date result to text by using the CONVERT function with a date-format argument.

Retrieving the Last Valid Date

The ENDDATE function, which returns the last date for which an expression has a non-NA value.

Examples

Example 8–28 Finding the Beginning Date

The following statements limit the values in the `month`, `product`, and `district` dimensions, then send the first date for which the `units` variable contains a non-`NA` value for unit sales of tents in the Chicago district to the current outfile.

```
LIMIT month TO ALL
LIMIT product TO 'TENTS'
LIMIT district TO 'CHICAGO'
SHOW BEGINDATE(units)
```

These statements produce the following output.

```
01JAN95
```

BITAND

BITAND computes an AND operation on the bits of two integers. This function is commonly used with the DECODE function.

Return Value

INTEGER

Syntax

BITAND (*argument1* , *argument2*)

Arguments

argument1
A nonnegative INTEGER.

argument2
A nonnegative INTEGER.

BLANK

The BLANK command sends one or more blank lines to the current outfile. BLANK is normally used only in programs. For example, in a report program, BLANK is commonly used to insert blank lines that separate headings from data or that separate groups of data from one another.

Syntax

BLANK [*n*]

Arguments

n
An INTEGER expression with a value of `0` (zero) or higher, that specifies how many blank lines should be inserted. When you omit *n*, Oracle OLAP inserts one blank line. `NA` produces an error.

Examples

Example 8–29 Inserting Blank Lines

This example inserts two blank lines between the title of a report and the column headings. The following lines are from a report program.

```
LSIZE = 50
HEADING WIDTH LSIZE CENTER 'Quarterly Sales Report'
BLANK 2
ROW WIDTH 20 'Unit Sales' ACROSS month -
   'Jan96' TO 'Mar96': month
```

The program produces the following output.

```
              Quarterly Sales Report


Unit Sales           Jan96    Feb96    Mar96
```

BLANKSTRIP

The BLANKSTRIP function removes leading or trailing blank spaces from text values. BLANKSTRIP is useful for such purposes as removing unwanted blank spaces from fixed-length fields in an imported worksheet.

Return Value

TEXT or NTEXT

Syntax

BLANKSTRIP(*text-expression* [TRAILING|LEADING|BOTH])

Arguments

text-expression
A text expression from which to remove blank spaces. When you specify a TEXT expression, the return value is TEXT. When you specify an NTEXT expression, the return value is NTEXT.

TRAILING
Removes blank spaces at the end of the text.

LEADING
Removes blank spaces at the beginning of the text.

BOTH
Removes both leading and trailing spaces.

Examples

Example 8–30 Stripping Leading and Trailing Blanks

In this example, we remove both leading and trailing blank spaces from the field `prodlabel` in an imported worksheet and store the results in a variable called `product`.

```
product = BLANKSTRIP(prodlabel, BOTH)
```

BMARGIN

The BMARGIN option defines the number of blank lines for the bottom margin of output pages. BMARGIN is meaningful only when PAGING is set to `YES` and only for output from statements such as REPORT and DESCRIBE. The BMARGIN option is usually set in the initialization section of report programs.

Data type

INTEGER

Syntax

BMARGIN = *n*

Arguments

n

An INTEGER expression that specifies the number of lines that you want to set aside for the bottom margin in a report. The default is `1`.

Notes

Output to the Default Outfile

When you set BMARGIN for the default outfile, the new value remains in effect until you reset it, regardless of intervening OUTFILE statements that send output to a file. That is, the value of BMARGIN is automatically saved for the default outfile.

Output to a File

To set BMARGIN for a file, first make the file your current outfile by specifying its name in an OUTFILE statement, then set BMARGIN to the desired value. The new value remains in effect until you reset it or until you use an OUTFILE statement to direct output to a different outfile. When you direct output to a different outfile, BMARGIN returns to its default value of `1` for the file.

Examples

Example 8–31 Setting the Bottom Margin of a Report Page

Suppose you want to be able to make notes on the bottom of a report page. You can set a large bottom margin of 5 lines. Here is the statement that you would include in the initialization section of your report program.

```
BMARGIN = 5
```

BREAK

The BREAK command transfers program control from within a SWITCH, FOR, or WHILE statement to the statement immediately following the DOEND associated with SWITCH, FOR, or WHILE. You can use BREAK only within OLAP DML programs and only with the SWITCH, FOR, or WHILE statements.

Syntax

BREAK

Notes

TEMPSTAT Statement and BREAK Statement

Within a FOR loop of a program, when a DO ... DOEND phrase follows TEMPSTAT, status is restored when the DOEND, BREAK, or GOTO is encountered.

Examples

Example 8–32 Using BREAK with SWITCH

The following lines from a program include a SWITCH statement with two case labels. The last statement under each case label is BREAK, which ensures that execution does not continue from one set of case statements to the next. Each BREAK statement transfers control to the statement that follows DOEND.

```
SWITCH userchoice
    DESCRIPTION 'MARKET REPORT\NFINANCE REPORT\NNO REPORT')
    DO
         CASE 'market':
            ...
            BREAK
         CASE 'finance':
            ...
            BREAK
         DEFAULT:
            ...
            BREAK
    DOEND
cleanup:
      ...
```

CALENDARWEEK

The CALENDARWEEK option determines whether weeks should be aligned with the actual calendar year.

> **Note:** You can only use this function with dimensions of type WEEK. You can*not* use this function for time dimensions that are implemented as hierarchical dimensions of type TEXT.

Data type

BOOLEAN

Syntax

CALENDARWEEK = {<u>YES</u>|NO}

Arguments

YES
Specifies that weeks are aligned with the calendar year. For example, if you have defined a dimension of type WEEK, Oracle OLAP numbers its values so that the first week in the calendar year is week 1, the second week in the calendar year is week 2, and so on. Weeks are aligned with the calendar year regardless of any beginning or ending date specified in the WEEK dimension definition.

NO
Specifies that weeks are not aligned with the calendar year. Instead, weeks are numbered so that they are aligned with the date specified in the dimension definition. For example, if you have defined a dimension of type WEEK with a beginning or ending date, its values are numbered so that the week corresponding to the date in the dimension definition is week `1`, the following week is week `2`, and so on.

Notes

Fiscal Years

Setting CALENDARWEEK to `NO` causes weeks to be numbered so that the number 1 is assigned to the week beginning or ending on the date specified in the DEFINE DIMENSION statement. This week is then assigned to a fiscal year, which is the calendar year of the first January 1 on or after the week's starting date. For example, if you define a dimension of type WEEK with a starting date of `02Jan1996` (or, equivalently, an ending date of `08Jan1996`), the week starting `02Jan1996` will be considered week `1` of fiscal year `1997`. If, by contrast, you had given the dimension a starting date between `02Jan1995` and `01Jan1996`, then the week starting on that date would be week `1` of fiscal year `1996`.

Examples

Example 8–33 Aligning Weeks with the Calendar Year

The following statements define a dimension of type WEEK, define its ending date, add values to the dimension, and produce a report.

```
DEFINE week dimension WEEK ENDING '18Jan97'
MAINTAIN week ADD '21Dec96' '25Jan97'
REPORT W 22 CONVERT(week DATE)
```

These statements produce the following output.

```
WEEK                 CONVERT(WEEK DATE)
-------------- --------------------
w51.96         21Dec96
w52.96         28Dec96
w1.97          04Jan97
w2.97          11Jan97
w3.97          18Jan97
w4.97          25Jan97
```

Example 8–34 Aligning Weeks with a Specified Ending Date

The following statements set the CALENDARWEEK option to `NO`, which aligns the weeks with the ending date that is specified in the definition of the `week` dimension in "Aligning Weeks with the Calendar Year" on page 8-83.

```
CALENDARWEEK = NO
REPORT W 22 CONVERT(week date)
```

These statements produce the following output.

```
WEEK              CONVERT(WEEK DATE)
-------------- --------------------
w50.97         21Dec96
w51.97         28Dec96
w52.97         04Jan97
w53.97         11Jan97
w1.98          18Jan97
w2.98          25Jan97
```

CALL

The CALL command invokes a program. When the program has arguments, which are always enclosed in parentheses, it passes these arguments to the called program.

Syntax

CALL *program-name* [(*arg* ...)]

Arguments

program-name

The name of the program to be called.

arg

One or more optional arguments expected by the called program. These arguments can be declared in the called program with ARGUMENT, or they can be referenced in the program with ARCTAN2. When the program uses the ARGUMENT statement, when you use CALL to invoke the program, specify the arguments so that they match the positions of the arguments declared in the called program.

Notes

Dimension Arguments

When you pass a dimension value or dimension name as an argument, you must enclose the exact text value in single quotes, for example, `'Jan96'`. When the program arguments are declared with the ARGUMENT statement, you can pass a text expression that evaluates to a text value.

Program Return Values

When you use CALL to invoke a program that returns a value, the return value is discarded. A program can use the CALLTYPE function to determine whether it was invoked as a function, as a command, or by using CALL.

ARGUMENT Command or ARG Function

The called program can process arguments using either the ARGUMENT statement or the ARG function. In a program that has been invoked with CALL or as a function, the ARGS and ARGFR functions always return `NA`.

When CALL invokes a program whose arguments are not declared with the ARGUMENT statement, the arguments passed can be referenced with the ARG function. However, the ARG function is a text function and, as a result, interprets all arguments passed as text values. When you want to pass NTEXT arguments, be sure to declare them using ARGUMENT instead of using ARG. With ARG, NTEXT arguments are converted to TEXT, and this can result in data loss when the NTEXT values cannot be represented in the database character set.

ARGUMENT Statement Processing

When a program is invoked with CALL or as a function, the following two-step process occurs. When an error occurs in either step, the program is not executed.

1. The specified data types are established. Argument expressions specified by the calling program are evaluated left to right, and their data types are identified. Any expression representing a dimension value can be a text (TEXT or ID), numeric (INTEGER, DECIMAL, and so on), or RELATION value. An error in one argument expression stops the process.
2. Each specified data type is matched with the declared data type. Argument expressions are matched by position with the declared arguments in the called program. The first argument expression is matched with the first declared argument variable, the second argument expression is matched with the second declared argument variable, and so on. Each expression is converted in turn to the declared data type of the argument variable.

When an argument variable is declared as a dimension value, the matching value passed from the calling program can be TEXT or ID (representing a value of the specified dimension), numeric (representing a logical dimension position), or RELATION (representing a physical dimension position).When the matching value is a non-integer numeric value (for example, DECIMAL), it is rounded to the nearest INTEGER to represent a logical dimension position.

When an argument variable is declared as something other than a dimension value, and the matching value from the calling program is a RELATION value, an error will occur. When you want to pass a RELATION value that will be received as a TEXT argument, use the CONVERT function to convert the value in the program's argument list.

ARGUMENT Statement with Extra Arguments

When the calling program specifies more arguments than are declared in the called program, the extra arguments are ignored. When the calling program specifies fewer arguments than are declared in the called program, the extra argument variables are given `NA` values.

ARGUMENT Statement Passing by Value

When arguments are declared with the ARGUMENT statement, they are passed *by value* to a program. As a result, the called program is given only the *value* of an argument, without access to any analytic workspace object to which it might be related. However, when the name of an analytic workspace object is specified as an argument enclosed in single quotes, the value of the analytic workspace object is not passed. Instead, the name of the object is passed as a text string. See Example 8–35, "Calling a Program or Function" on page 8-87.

Examples

Example 8–35 Calling a Program or Function

This example illustrates how two programs, `roundup.p` and `roundup.f`, are used in different ways to evaluate data and produce output.

`roundup.p` accepts the name of a decimal variable as a text string and produces a report of that variable's values rounded to the nearest INTEGER. `roundup.f` also accepts the name of a decimal variable. However, instead of passing the name of the variable as a text string, the variable's value is passed as an argument. `roundup.f` does not produce a report. Instead, it returns each of the values of the decimal variable, rounded to the nearest INTEGER.

`roundup.p` is invoked using CALL and includes a REPORT statement. In contrast, `roundup.f` is invoked as a user-defined function whose return value is then used as an argument to a REPORT statement.

The `roundup.p` program uses ARGUMENT to declare a text argument. When invoked, `roundup.p` uses the argument as the name of a decimal variable. The calling program passes the name of the variable in order to give the called program access to *all* the values of the dimensioned variable. When the calling program passed the variable itself, instead of its name, only a *single* value would have been accessible to the called program. This program does not return a value; it produces a report.

```
DEFINE roundup.p PROGRAM INTEGER
PROGRAM
ARGUMENT varname TEXT
Report Down Line Across Month: Heading 'VARNAME' -
   IF INTPART(&varname) EQ &varname -
   THEN &varname ELSE INTPART(&varname) + 1
END
```

The following statements

```
LIMIT division TO 1
LIMIT month TO 1 TO 4
DECIMALS = 0
CALL roundup.p('actual')
```

produce the following report.

```
DIVISION: CAMPING
               ----------------- Varname------------------
               -------------------MONTH-------------------
LINE             Jan95      Feb95      Mar95      Apr95
-------------- ---------- ---------- ---------- ----------
revenue           533,363    572,797    707,198    968,858
cogs              360,811    400,902    478,982    641,716
gross.margin      172,553    171,895    228,217    327,143
marketing          37,370     38,867     51,224     69,439
selling            89,008     86,458    102,233    139,567
r.d                24,308     23,400     39,943     57,186
opr.income         21,868     23,171     34,819     60,952
taxes              15,971     16,320     23,030     27,584
net.income          5,898      6,851     11,789     33,368
```

Another way to produce the same report is to write a user-defined function that can be used as an argument to the REPORT statement as illustrated in the following program named `roundup.f`.

```
DEFINE roundup.f PROGRAM INTEGER
PROGRAM
ARGUMENT realval DECIMAL
IF realval EQ INTPART(realval)
THEN RETURN INTPART(realval)
ELSE RETURN INTPART(realval) + 1
END
```

The following statements

```
LIMIT division TO 1
LIMIT month TO 1 TO 4
DECIMALS = 0
REPORT DOWN line ACROSS month: roundup.f(actual)
```

produce the following report.

```
DIVISION: CAMPING
               ------------ ROUNDUP.F(ACTUAL)-------------
               -------------------MONTH-------------------
LINE             Jan95      Feb95      Mar95      Apr95
-------------- ---------- ---------- ---------- ----------
revenue           533,363    572,797    707,198    968,858
cogs              360,811    400,902    478,982    641,716
gross.margin      172,553    171,895    228,217    327,143
marketing          37,370     38,867     51,224     69,439
selling            89,008     86,458    102,233    139,567
r.d                24,308     23,400     39,943     57,186
opr.income         21,868     23,171     34,819     60,952
taxes              15,971     16,320     23,030     27,584
net.income          5,898      6,851     11,789     33,368
```

(Compare the `roundup.f` program with the `roundup.p` program. `roundup.f` returns a value; it does not produce a report.)

CALLTYPE

Within an OLAP DML program, the CALLTYPE function indicates whether a program was invoked as a function, as a command, by using a CALL statement, or triggered by the execution of an OLAP DML statement.

The return value of CALLTYPE is either `FUNCTION`, `COMMAND`, or `CALL`. CALLTYPE is for use only within programs.

Return Value

TEXT

The return value of CALLTYPE is:

- `FUNCTION` when the program was invoked as a function that returns a value.
- `COMMAND` when the program was invoked as a a command.
- `CALL` when the program was invoked using a CALL statement.
- `TRIGGER` when the program is a trigger program (that is, when a TRIGGER command associated the program with an object event) was invoked in response to a OLAP DML statement.

Syntax

CALLTYPE

Notes

Handling Arguments

The CALLTYPE function is helpful when you want to find out how your program has been called, so that you can handle appropriately any arguments that have been passed. See ARGUMENT for information on the differences in argument handling that depend on how a program is invoked. See CALL for information on calling programs.

Examples

Example 8–36 Determining the Calling Method

This sample program, called `myprog`, demonstrates how CALLTYPE returns different values depending on how the program is invoked.

```
DEFINE myprog PROGRAM
PROGRAM
SHOW CALLTYPE
RETURN('This is the return value')
END
```

The following statements invoke `myprog`: 1) as command; 2) with a CALL statement; 3) as a function.

```
myprog
CALL myprog
SHOW myprog
```

The three statements send the following output to the current outfile. Note that the return value of myprog appears only when the program is called as a function.

```
COMMAND
CALL
FUNCTION
This is the return value
```

CATEGORIZE

The CATEGORIZE function groups the values of a numeric expression into categories. You define the categories by specifying a series of increasing numeric values. The result that CATEGORIZE returns is dimensioned by all the dimensions of *expression*. For each cell in *expression*, CATEGORIZE returns one of the following: the category in which the number falls, zero (`0`) for a value below the range of the first category, minus one (`-1`) for a value above the range of the last category, or `NA` for an `NA` value.

Return Value

DECIMAL

Syntax

CATEGORIZE(*expression* {*values*|*group-expression*})

where:

values
has the following syntax:

bottom-value [*next-lowest-break-value*] *top-value*

Arguments

expression
The numeric expression whose values are to be categorized.

bottom-value
A number that specifies the lowest number in the series and sets the bottom limit of category 1.

next-lowest-break-value
A number that specifies the beginning of the range of the next category.

top-value
A number that specifies the highest number in the series and sets the upper limit of the highest category.

group-expression
A one-dimensional numeric expression that defines the break values for the categories.

Examples

Example 8–37 Specifying Category Range Values

These examples use the following `geography` and `items` dimensions and `sales2` variable.

```
DEFINE geography DIMENSION TEXT
MAINTAIN geography ADD 'g1' 'g2' 'g3'
DEFINE items DIMENSION TEXT
MAINTAIN items ADD 'Item1' 'Item2' 'Item3' 'Item4' 'Item5'
DEFINE sales2 DECIMAL <geography items>
```

Assume the `sales2` variable has the following data values.

```
               -------------SALES2-------------
               -----------GEOGRAPHY------------
ITEMS              g1         g2         g3
-------------- ---------- ---------- ----------
Item1               30.00      15.00      12.00
Item2               10.00      20.00      18.00
Item3               15.00      20.00      24.00
Item4               30.00      25.00      25.00
Item5                  NA       7.00      21.00
```

This statement reports the result of categorizing the `sales2` variable.

```
REPORT CATEGORIZE(sales2 10 15 20 25)
```

The preceding statement produces the following output.

```
               -CATEGORIZE(SALES2 10 15 20 25)-
               -----------GEOGRAPHY------------
ITEMS              g1         g2         g3
-------------- ---------- ---------- ----------
Item1               -1.00       2.00       1.00
Item2                1.00       3.00       2.00
Item3                2.00       3.00       3.00
Item4               -1.00       3.00       3.00
Item5                  NA       0.00       3.00
```

Example 8–38 Specifying a Group-Expression

These statements define a `groups` dimension and a `groupval` variable.

```
DEFINE groups DIMENSION TEXT
MAINTAIN groups ADD 'Grp1' 'Grp2' 'Grp3' 'Grp4'
DEFINE groupvals DECIMAL <groups>
groupvals(groups 'Grp1') = 10
groupvals(groups 'Grp2') = 15
groupvals(groups 'Grp3') = 20
groupvals(groups 'Grp4') = 25
```

This statement reports the result of calling the CATEGORIZE function with the `sales` variable as the *expression* argument and the `groupvals` variable as the *group-expression* argument of the call.

```
REPORT CATEGORIZE(sales, groupvals)
```

The preceding statement produces the same output as the statement in the "Specifying Category Range Values" on page 8-93.

CDA

With the CDA command, you can identify or change the current directory object for your session.

With an established current directory object, you can specify a file identifier in a DML file access statement without including the name of the directory object. Some examples of file access statements are FILECOPY, FILEMOVE, FILEDELETE, EXPORT, and IMPORT.

Syntax

CDA [*directory-alias*]

Arguments

directory-alias
A text expression that specifies the directory object that you want to be the current one for your session.

When you do not specify this argument, CDA sends the name of the current directory object to the current outfile. When there is no current directory object, the statement reports that fact.

Notes

Specifying a File Identifier with an Established Current Directory Object
The following statement moves the file `log.txt` from your session's current directory object to file `oldlog.txt` in a directory object called `backup`.

```
FILECOPY 'log.txt' 'backup/oldlog.txt'
```

Setting Up a Directory Object
A database administrator must set up a directory object and give you access to it.

Examples

Example 8–39 Specifying the Current Directory Object

The following statement identifies `mydir` as the current directory object.

```
CDA 'mydir'
```

Example 8–40 Obtaining the Current Directory Object

The following statement causes the current directory object to be sent to the current outfile.

```
CDA
```

This statement produces the following output.

```
The current directory is MYDIR.
```

CEIL

The CEIL function returns the smallest whole number greater than or equal to a specified number.

Return Value

NUMBER

Syntax

CEIL(*n*)

Arguments

n
A whole number (NUMBER datatype) that you specify.

Examples

Example 8–41 Displaying the Smallest Integer Greater Than or Equal to a Number

The following statements show results returned by CEIL.

The statement

```
SHOW CEIL(15.7)
```

produces the following result

```
16
```

The statement

```
SHOW CEIL(-6.457)
```

produces the following result.

```
-6
```

CHANGEBYTES

The CHANGEBYTES function changes one or more occurrences of a specified string in a text expression to another string.

Return Value

TEXT

Syntax

CHANGEBYTES(*text-expression oldtext newtext* [*number*])

Arguments

text-expression
The expression in which bytes are to be changed. When *text-expression* is a multiline TEXT value, CHANGEBYTES preserves the line breaks in the returned value.

oldtext
A text expression that contains one or more bytes that will be changed.

newtext
A text expression that contains one or more bytes that will replace *oldtext*.

number
An INTEGER that represents the number of times *oldtext* should be replaced with *newtext* when *oldtext* appears more than once in *text-expression*. The default is to change all occurrences of *oldtext*.

Notes

Single-Byte Characters

When you are using a single-byte character set, you can use CHANGECHARS instead of CHANGEBYTES.

NTEXT Data Type

This function does not accept NTEXT arguments, because it is oriented toward byte-manipulation instead of character manipulation. It always returns values of type TEXT. When you must use this function on NTEXT values, use the CONVERT or TO_CHAR function to convert the NTEXT value to TEXT.

Replacing Bytes in a Text Value

You can use REPLBYTES to replace bytes in a text value beginning at a certain byte.

Examples

Example 8–42 Changing Text Values Using Bytes

This example shows how to change one instance of a portion of a text value.

The statement

```
SHOW CHANGEBYTES('Hello there, Joe\nHello there, Jane',
   'there', - 'to you', 1)
```

produces the following output.

```
Hello to you, Joe
Hello there, Jane
```

CHANGECHARS

The CHANGECHARS function changes one or more occurrences of a specified string in a text expression to another string.

See also: The following related functions:

- CHANGEBYTES which you can use instead of CHANGECHARS when you are using a multibyte character set
- REPLCHARS which you can use to replace characters in a text value beginning at a certain character

Return Value

TEXT or NTEXT

This function accepts TEXT values and NTEXT values as arguments. The data type of the return value depends on the data type of the values specified for the arguments:

- When all arguments are TEXT values, the return value is TEXT.
- When all arguments are NTEXT values, the return value is NTEXT.
- When the arguments include both TEXT and NTEXT values, the function converts all TEXT values to NTEXT before performing the function operation, and the return value is NTEXT.

Syntax

CHANGECHARS(*text-expression oldtext newtext* [*number*] [UPCASE])

Arguments

text-expression
The expression in which characters are to be changed. When *text-expression* is a multiline text value, CHANGECHARS preserves the line breaks in the returned value.

oldtext
A text expression that contains one or more characters that will be changed.

newtext
A text expression that contains one or more characters that will replace *oldtext*.

number
An INTEGER that represents the number of times *oldtext* should be replaced with *newtext* when *oldtext* appears more than once in *text-expression*. The default is to change all occurrences of *oldtext*.

UPCASE
Specifies that CHANGECHARS should uppercase *text-expression* and *oldtext* before trying to find a match. CHANGECHARS does not uppercase the return value.

Examples

Example 8–43 Changing the Values of Text Characters

This example shows how to change one instance of a portion of a text value.

The statement

```
SHOW CHANGECHARS('Hello there, Joe\nHello there, Jane',
   'there', - 'to you', 1)
```

produces the following output.

```
Hello to you, Joe
Hello there, Jane
```

CHARLIST

The CHARLIST function transforms an expression into a multiline text value with a separate line for each value of the original expression.

Return Value

TEXT or NTEXT

The data type of the return value depends on the data type of the expression:

- When the expression is TEXT, the return value is TEXT.
- When the expression is NTEXT, the return value is NTEXT.
- When the expression has a data type other than TEXT or NTEXT, CHARLIST automatically converts its values to TEXT to create the result.

Syntax

CHARLIST(*expression* [*dimensions*])

Arguments

expression
The expression to be transformed into a multiline text value.

dimensions
The dimensions of the return value. When you do not specify a dimension, CHARLIST returns a single value. When you provide one or more dimensions for the return value, CHARLIST returns a multiline text value for each value in the current status list of the specified dimension. Each dimension must be an actual dimension of the expression; it cannot be a related or base dimension.

Notes

Creating Lists of Workspace Objects

You can use CHARLIST with the NAME dimension to create lists of workspace objects. Such lists are useful when you want to perform a task on a group of objects. To create a list of objects, limit NAME to the names of the objects in which you are interested (for example, worksheets). You can then use CHARLIST to loop over the

NAME dimension and perform the task on each item in this group. You can use CHARLIST in this way with any statement that can take a list of names as its argument. See "Deleting Workspace Objects" on page 8-103.

Empty Composites

CHARLIST cannot return values of a variable dimensioned by a composite when you have not assigned any values to the variable. In this case, the variable and the composite are considered to be empty, and CHARLIST returns NA.

Examples

Example 8–44 Deleting Workspace Objects

Suppose you want to delete all objects of a certain type in your workspace, for example, all worksheets. You can use CHARLIST and an ampersand (&) to do this.

```
LIMIT NAME TO OBJ(TYPE) EQ 'WORKSHEET'
DELETE &CHARLIST(NAME)
```

Example 8–45 Creating a List of Top Sales People

Assume you have stored the names of the sales people who sold the most for each product in `product.memo`, a text variable with the dimensions of `product` and `month`. You then want to create a list of top sales people broken out by product. To do this, you can create a variable dimensioned by `product` and then use CHARLIST with the `product` dimension to create a separate list of all the top sales people for each product.

```
DEFINE topsales VARIABLE TEXT <product>
topsales = CHARLIST(product.memo product)
```

9 CHGDFN to DDOF

This chapter contains the following OLAP DML statements:

- CHGDFN
- CHGDIMS
- CLEAR
- COALESCE
- COLVAL
- COLWIDTH
- COMMAS
- COMMIT
- COMPILE
- COMPILEMESSAGE
- COMPILEWARN
- CONSIDER
- CONTEXT command
- CONTEXT function
- CONTINUE
- CONVERT
- COPYDFN
- CORRELATION
- COS

CHGDFN

The CHGDFN command enables you to change certain aspects of the definitions of analytic workspace objects.

Before you can use CHGDFN to change the definition of an object, use CONSIDER to make that object definition the current definition.

Note: You cannot use CHGDFN to change definitions of objects that are in an analytic workspace that is attached in multiwriter mode.

Syntax

CHGDFN *desired-change*

where *desired-change* is one of the following:

varname SEGWIDTH *length-dim*...

partitioned-varname {DROP | ADD } (*partition-instance*...)

partition-template {DEFINE | DELETE [CLEAR] } (*partition-instance*...)

partition-template RENAME PARTITION *old-name new-name*

{*conjoint* | *composite*} {HASH | BTREE | NOHASH}

concat BASE ADD *dimensionlist*

conjoint COMPOSITE

composite DIMENSION

dwmqy-dimname { {BEGINNING | ENDING} *phase* | {EARLIER | LATER} *n*}

concat [NOT] UNIQUE

Arguments

varname
The name of the variable whose segment size you want to set.

SEGWIDTH
Indicates explicit sizing of a variable's segments. A segment is contiguous disk space reserved for a portion of the total number of values a variable holds. The number of segments in a variable affects the performance of data loading and data accessing. See "Using CHGDFN SEGWIDTH" on page 9-7.

The segment size that you specify is used not only for the variable you designate as *varname,* but also for all other variables and relations that are defined with the same combination of dimensions and composites in the same order. The DEFINE command sets the SEGWIDTH at the time it creates a variable or relation. Changing the SEGWIDTH affects any new variable or relation that you subsequently create. The changed SEGWIDTH setting does not apply to previously existing variables or relations.

The time it takes to do data loads on a variable depends on how many pages are brought into memory and then written back out. This number can be affected by how a variable is divided into segments. Too many segments (thousands to millions) can degrade performance. See "Reducing the Number of Segments" on page 9-8.

The number of segments also affects data access. The time it takes to report a variable depends on how many values are brought into memory. You decide how many segments your variable should have based on your data loading and data accessing patterns.

DEFINE provides default segments. In most cases, you can use the default segments so that you do not have to use `CHGDFN SEGWIDTH` to manually control the size of segments. However, you may be able to improve performance by specifying the segment size instead of using the defaults.

When you are not sure what your segment size should be, use the maximum anticipated number of values for each dimension or composite as the length arguments to SEGWIDTH. Then only one segment will be created for the variable.

partitioned-varname
Specifies the name of a partitioned variable whose partitions you want to modify.

DROP *partition-instance*
ADD *partition-instance*
Removes or adds the specified partitions from the partitioned variable. See DEFINE VARIABLE for a complete description of the *partition-instance* argument.

DEFINE *partition*
DELETE [CLEAR] *partition-instance*
Removes or adds the specified partitions from the partition template object. See DEFINE PARTITION TEMPLATE for a complete description of the *partition-instance* argument.

When you include the optional CLEAR keyword, Oracle OLAP also drops any corresponding partitions in the variables that are partitioned using the partition template object. In other words, including CLEAR is the same as issuing an additional CHGDFN statements to DROP the partition from the variables partitoned by it.

RENAME PARTITION *old-name new-name*
Renames the specified partitions in the partition template object.

BASE ADD *dimensionlist*
Adds the dimension or dimensions specified by *dimensionlist* to the base dimensions of the concat dimension.

length-dim...
Segment width is specified as the maximum number of values in each segment for each dimension or composite in the variable's dimension list. The first *length-dim* is the number of values for the dimension or composite in the first position of the dimension list in the variable's definition (that is, the fastest-varying dimension or composite), the second *length-dim* is the number of values for the dimension or composite in the second position in the dimension list, and so on. See "Using CHGDFN SEGWIDTH" on page 9-7.

conjoint
composite
For the *index* syntax, the name of the conjoint dimension or composite whose index algorithm you want to change. For the *conjoint-to-composite* syntax, the name of the conjoint dimension you want to change to a composite. For the *composite-to-dim* syntax, the name of the composite you want to change to a conjoint dimension. You cannot change a conjoint dimension to a composite when the conjoint is a dimension of a formula.

HASH
BTREE
NOHASH
Indicates the index algorithm used to load and access values of your conjoint dimension or composite without losing data in objects defined with the conjoint or

composite. A composite cannot be changed to NOHASH. A conjoint can be changed to NOHASH only when it was originally defined as HASH. See "Changing Conjoints to NOHASH" on page 9-10.

HASH, NOHASH, and BTREE are different index algorithms used to load and access the values of a conjoint dimension or composite. HASH is the default for conjoints. The default for composites is determined by the SPARSEINDEX option, which has a default value of BTREE. The index algorithm affects the performance of loading and accessing large conjoints or composites. Performance varies depending on your machine configuration, the organization of your data, and the design of your application. You can do performance testing to determine which algorithm provides the best performance for your situation. See "When to Use HASH" on page 9-9, "When to Use NOHASH" on page 9-9, and "When to Use BTREE" on page 9-9.

COMPOSITE
Indicates changing a conjoint dimension into a named composite. There are some restrictions on changing conjoint dimensions to composites; when a conjoint has the NOHASH index algorithm or when it has permissions, you cannot change it to a composite.

DIMENSION
Indicates changing a named composite into a conjoint dimension.

dwmqy-dimname
Specifies or changes the phase of a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR.

BEGINNING *phase*
ENDING *phase*
Specifies the beginning phase or ending phase of a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR. You must specify the phase as a date, giving the month, day, and year, enclosed in single quotes, using any of the input styles that are valid for variable values with a data type of DATE. When you specify a date with an ambiguous meaning (such as `'03 05 97'`), the date is interpreted according to the current setting of the DATEORDER option. For more information about specifying dates, see DATEORDER.

EARLIER *n*
LATER *n*
n is an INTEGER expression that increments or decrements the period on which the DAY, WEEK, MONTH, QUARTER, or YEAR dimension's phase begins or ends. For example, for a WEEK dimension whose current begin phase is Monday, specify `LATER 2` to change the phase to Wednesday.

[NOT] UNIQUE
When you include `NOT`, changes a unique concat dimension to a nonunique concat. When you do not include `NOT`, changes a nonunique concat dimension to a unique concat dimension. See DEFINE DIMENSION CONCAT for more information on concat dimensions.

Notes

Using CHGDFN SEGWIDTH

Use the SEGWIDTH keyword with the CHGDFN command to specify the segment size of a variable. For example, when you create a variable dimensioned by `month` and `product` and set the SEGWIDTH of `month` to `150` and `product` to `90,000`, each segment will hold up to 150 x 90,000 values of the variable.

Suppose you have a variable called `d.sales` that is dimensioned by `month` and by a composite with the base dimensions `market` and `product`. The definition of `d.sales` looks like the following.

```
DEFINE d.sales VARIABLE DECIMAL <month SPARSE<market product>>
```

Suppose you want to have only one segment in the `d.sales` variable. You estimate that the `month` dimension will eventually have 150 values and the composite will have 100,000. The following statement will create one segment for the `d.sales` variable.

```
CHGDFN d.sales SEGWIDTH 150 100000
```

When you wish, you can set the SEGWIDTH of one or more dimensions to 0. The value of 0 for *length-dim* has a special meaning: Oracle OLAP will grow segments in that dimension as needed, minimizing the number of segments but not changing any existing segments.

When you do not use `CHGDFN SEGWIDTH`, the default behavior is to assign a segwidth of `0` (zero) on non-composite dimensions and a large value for composites that are not the slowest-varying in the dimension set. This allows new dimension and composite values to be added in most situations without greatly increasing the number of segments and degrading performance.

You must specify a number, `0` (zero), or nonzero, for every dimension and composite.

You can always specify `0` (zero for the slowest-varying dimension, because the data for any values that are later added to that dimension will be appended to the existing data in the variable's last segment. For example, a better way to specify segment size for `d.sales` is to specify 0 for the slowest-varying dimension.

```
CHGDFN d.sales SEGWIDTH 150 0
```

Any composite that is not the slowest-varying dimension should have a nonzero value. For example, suppose you have a variable called `f.costs` with the following definition.

```
DEFINE f.costs VARIABLE DECIMAL <geog SPARSE<product channel> time>
```

You estimate the `geog` dimension will have 100 values and the composite will have 300,000. You do not have to estimate the number of values for the `time` dimension, because it is the slowest-varying dimension. The following statement will create one segment for the `f.costs` variable.

```
CHGDFN f.costs SEGWIDTH 100 300000 0
```

Reducing the Number of Segments

You can use OBJ (NUMSEGS) to find out if you have too many segments for objects that have a particular dimension set. When you find that you do, you can reduce the number of segments by following these steps:

1. Export the variables and relations that use this dimension set to an EIF file.
2. Execute a MAINTAIN DELETE ALL statement for one of the dimensions in the dimension set.
3. Optimally, execute a CHGDFN command for one of the variables or relations with this dimension set, and increase the value of the length arguments to the SEGWIDTH keyword.
4. From the EIF file, import all the values you exported in Step 1.

Adding Base Dimensions to a Concat Dimension

When you add one or more dimensions as base dimensions of a concat, then Oracle OLAP appends the dimensions to the existing list of base dimensions of the concat. Objects that are dimensioned by the concat, or objects that are dimensioned by a concat that has the altered concat as a base dimension, gain additional NA values. You cannot add as a base dimension a dimension that is already a component of the concat dimension.

Changing Index Algorithms

Changing the index algorithm of a large conjoint dimension or composite from one algorithm type to another may take a considerable amount of time. The CHGDFN command cannot be interrupted.

When to Try Different Algorithms CHGDFN gives you an easy way to test using different index algorithms in order to determine how each affects performance. You can use CHGDFN to try using different algorithms when a data load for a conjoint dimension takes longer than expected. For example, suppose a data load executes well at first, then slows down drastically. Use CHGDFN to change the index algorithm from BTREE to NOHASH. Try the data load again to determine whether or not using NOHASH improves performance. You can then use CHGDFN to change the index algorithm back to BTREE.

When to Use BTREE BTREE is a standard indexing method that is recommended for composites and conjoint dimensions. Use BTREE as the default unless you are an advanced user and have a special need that requires HASH or NOHASH. BTREE tends to group similar values together, which results in better locality of access.

When to Use HASH HASH is a standard indexing method that can be used for composites or conjoint dimensions that have only 2 or 3 base dimensions. One advantage to using HASH is that it results in a small amount of code. However, HASH is generally not recommended. Using HASH results in a very large index table, which can be too large to fit into memory.

When to Use NOHASH You can use NOHASH with conjoint dimensions only. It can be advantageous to use NOHASH when there is little memory available and the conjoint dimension has only 2 or 3 base dimensions.

Also, you can use NOHASH when you load a very large initial amount of data. When you use NOHASH, the data will be loaded in a way that makes it easy to access that data after it has been loaded. Once the data is loaded, change the definition of the conjoint dimension back to BTREE to ensure good performance.

Otherwise, performance is likely to suffer, especially when the conjoint dimension has 4 or more base dimensions. See "Changing Conjoints to NOHASH" on page 9-10.

Changing Conjoints to NOHASH When you need to change a conjoint dimension that was originally defined with the BTREE algorithm to a NOHASH conjoint, you can use the following method:

1. Export the conjoint dimension and all the objects dimensioned by it to an EIF file.
2. Delete all the objects dimensioned by the conjoint dimension, and then delete the conjoint itself.
3. Redefine the conjoint as a NOHASH conjoint.
4. Import the conjoint dimension and the objects dimensioned by it from the EIF file. The NOHASH attribute on the definition at the time of the import will cause the conjoint dimension to be read in as a NOHASH conjoint.

Changing Unnamed Composites to Conjoints

When you want to change an unnamed composite into a conjoint dimension, you can use the RENAME command to change the unnamed composite into a named composite, and then use CHGDFN to change the named composite into a conjoint dimension.

Examples

Example 9–1 Adding an External Partition to a Variable

Assume that your analytic workspace has a sales variable with two external partitions—one partition for sales in 2002 and another partition for sales in 2003. The following definitions are used to define the sales variable.

```
DEFINE YEAR_2003 DIMENSION TEXT
DEFINE YEAR_2002 DIMENSION TEXT
DEFINE PRODUCT DIMENSION TEXT
DEFINE SALES_2003 VARIABLE DECIMAL <YEAR_2003 PRODUCT>
DEFINE SALES_2002 VARIABLE DECIMAL <YEAR_2002 PRODUCT>
DEFINE TIME DIMENSION CONCAT (YEAR_2003 YEAR_2002 YEAR_2004) UNIQUE
DEFINE PART_TEMP_SALES_BY_YEAR PARTITION TEMPLATE <TIME PRODUCT> -
   PARTITION BY CONCAT (TIME) -
    (PARTITION PARTITION_2002 <YEAR_2002 PRODUCT> -
     PARTITION PARTITION_2003 <YEAR_2003 PRODUCT>)
DEFINE SALES VARIABLE DECIMAL <PART_TEMP_SALES_BY_YEAR <TIME PRODUCT>> -
    (PARTITION PARTITION_2002 EXTERNAL SALES_2002 -
     PARTITION PARTITION_2003 EXTERNAL SALES_2003)
```

Assume that you want to add data for the year 2004 to `sales`. Before you can add the data, you need to add an external partition to `sales` to hold data. To add an external partition to `sales`, you take the following steps:

1. Issue the following DEFINE statements to add a definitions for a dimension for the values for 2004 and a `sales` variable to hold 2004 data.

   ```
   DEFINE YEAR_2004 DIMENSION TEXT
   DEFINE SALES_2004 VARIABLE DECIMAL <YEAR_2004 PRODUCT>
   ```

2. Issue the following CHGDFN statements to add the `year_2004` dimension to the time dimension, a partition for 2004 to the partition template used by `sales` and to the `sales` variable, itself.

   ```
   CHGDFN time BASE ADD year_2004
   CHGDFN part_temp_sales_by_year -
      DEFINE(PARTITION partition_2004 <year_2004 product>)
   CHGDFN sales ADD (PARTITION partition_2004 EXTERNAL sales_2004)
   ```

Now `time`, `part_temp_sales_by_year`, and `sales` have the following definitions.

```
DEFINE TIME DIMENSION CONCAT (YEAR_2003 YEAR_2002 YEAR_2004) UNIQUE
DEFINE PART_TEMP_SALES_BY_YEAR PARTITION TEMPLATE <TIME PRODUCT> -
   PARTITION BY CONCAT (TIME) -
    (PARTITION PARTITION_2002 <YEAR_2002 PRODUCT> -
     PARTITION PARTITION_2003 <YEAR_2003 PRODUCT> -
     PARTITION PARTITION_2004 <YEAR_2004 PRODUCT>)
DEFINE SALES VARIABLE DECIMAL <PART_TEMP_SALES_BY_YEAR <TIME PRODUCT>> -
    (PARTITION PARTITION_2002 EXTERNAL SALES_2002 -
     PARTITION PARTITION_2003 EXTERNAL SALES_2003 -
     PARTITION PARTITION_2004 EXTERNAL SALES_2004)
```

3. After you populate the `year_2004` dimension, you issue the following REPORT statement. You can see that the sales variable has a partition for 2004 data.

```
REPORT DOWN PARTITION(part_temp_sales_by_year) time product sales

PARTITION(PART_TEMP_SALES_BY_YEAR)     TIME     PRODUCT     SALES
---------------------------------- ---------- ---------- ----------
PARTITION_2002                     01Jan2002  00001           14.44
...
PARTITION_2003                     01Jan2003  00001           10.00
...
PARTITION_2004                     01Jan2004  00001              NA
...
PARTITION_2004                     Jan2004    00001              NA
...
PARTITION_2004                     2004       00001              NA
PARTITION_2004                     01Jan2004  00002              NA
...
PARTITION_2004                     2004       00002              NA
```

Example 9–2 Changing the Phase of a YEAR Dimension

The following statements first create a dimension of type YEAR for a fiscal year, then use CHGDFN to switch to a new time phase for the fiscal year.

```
DEFINE fiscal DIMENSION year BEGINNING '06 01 96'
CHGDFN fiscal BEGINNING '01 01 97'
```

Example 9–3 Adding a Base Dimension to a Concat Dimension

The following statements create a nonunique concat dimension named `reg.dist.ccdim` that has the `region` and `district` dimensions as its base dimensions and report the values of the concat.

```
DEFINE reg.dist.ccdim DIMENSION CONCAT(region district)
REPORT W 22 reg.dist.ccdim
```

The preceding statement produces the following output.

```
REG.DIST.CCDIM
--------------------
<region: East>
<region: Central>
<region: West>
<district: Boston>
<district: Atlanta>
<district: Chicago>
<district: Dallas>
<district: Denver>
<district: Seattle>
```

The following statements add the `store_id` dimension as a base to the concat dimension and then report the values of the concat again.

```
CHGDFN reg.dist.ccd BASE ADD store_id
REPORT W 22 reg.dist.ccd
```

The preceding statement produces the following output.

```
REG.DIST.CCD
----------------------
<region: East>
<region: Central>
<region: West>
<district: Boston>
...
<district: Seattle>
<store_id: 10>
<store_id: 20>
<store_id: 30>
<store_id: 100>
...
<store_id: 500>
<store_id: 510>
```

CHGDIMS

The CHGDIMS function changes the dimensionality of an expression or changes the dimension status during the evaluation of expression.

Return Value

Data type of the original expression.

Syntax

CHGDIMS(*expression* { LIMIT *valueset-list* | TO *dimension-list* | ADD *dimension-list* })

Arguments

expression
The expression you want to modify.

LIMIT *valueset-list*
Sets the current status list of the dimensions of *expression* to the values specified by *valuelist* while Oracle OLAP evaluates *expression*.

TO *dimension-list*
Specifies that Oracle OLAP evaluate the expression as though the dimensions of *expression* are the dimensions specified by *dimension-list*.

ADD *dimension-list*
Specifies that Oracle OLAP evaluate the expression as though the dimensions of *expression* are the dimensions of *expression* plus the dimensions specified by *dimension-list*

Examples

Assume that you have the following objects in your analytic workspace.

```
DEFINE PRODUCT DIMENSION TEXT
DEFINE GEOG DIMENSION TEXT
DEFINE SALES VARIABLE INTEGER <PRODUCT GEOG>
```

Assume, also, that the `sales` variable has the following values.

```
               -------------------SALES-------------------
               ------------------PRODUCT------------------
GEOG            Trousers    Skirts    Dresses     Shoes
-------------- ---------- ---------- ---------- ----------
USA                    13         20         32         18
Canada                 17         32         15         28
```

The following lines of code show how the value returned by a `TOTAL(sales)` expression varies depending on how you qualify that expression.

```
"Total over all dims with standard status
SHOW TOTAL(sales)
175

"Total over all dims using new status for product
SHOW CHGDIMS(TOTAL(sales) LLIMIT LIMIT(product TO FIRST 2))
82

"Total just over product
SHOW TOTAL(CHGDIMS(sales TO product))
83
```

CLEAR

The CLEAR command deletes the data that you specify for one or more variables.

Syntax

CLEAR [STATUS | {ALL [CACHE]}] [VALUES | {*aggdata* [USING *aggmapname*]}] FROM *varname*...

where *aggdata* is one or more of the following keywords that identifies the type of aggregated data that you want deleted from the variable.

```
AGGREGATE
LEAVES
PRECOMPUTES
NONPRECOMPUTES
```

Arguments

STATUS
Specifies that only the data that is currently in status will be taken into consideration. (Default)

ALL
Specifies that all of a variable's data will be taken into consideration regardless of the current status.

CACHE
Empties the session cache (see "What is an Oracle OLAP Session Cache?" on page 21-54 for details).

VALUES
Deletes all of a variable's stored data and replaces each deleted data value with an `NA` value. (Default)

AGGREGATE
Deletes the data in all cells populated by the execution of an AGGREGATE command or an AGGREGATE function.

PRECOMPUTES
For all variables *except* those dimensioned by a compressed composite, deletes any data that was calculated when an AGGREGATE command executed and replaces that data with `NA` values.

NONPRECOMPUTES
Deletes any data that was calculated on the fly when a AGGREGATE function executed and replaces that data with `NA` values.

LEAVES
Deletes the detail-level data, meaning, the "leaf" data.

FROM *varname*
Specifies the name of the variable or variables from which data will be deleted. When you specify more than one variable, then every variable must have exactly the same dimensions in exactly the same order in its definition. In other words, when you include multiple variables in one command, those variables must be identical in their dimensionality.

USING *aggmapname*
Specifies the name of the aggmap that should be used. When you include this phrase, the dimensionality of every variable included in the CLEAR command must be identical to the dimensionality of the aggmap. In other words, every variable definition must have the same dimensions in the same order as those in the definition of the aggmap.

Examples

Example 9–4 Clearing a Variable's Data

The CLEAR command gives you an easy way to delete all of a variable's stored data. Suppose you have defined a `sales` variable and loaded data into it. You then find out that much of this data has changed. It will be more efficient to clear the `sales` variable and reload all of the data than it would be to change the existing data. You can do so with the following statement.

```
CLEAR ALL FROM sales
```

In this example, the VALUES keyword is assumed by default. Therefore, all of the `sales` data is deleted and replaced with `NA` values.

Example 9–5 Clearing Aggregated Data

Suppose you have aggregated data for your `sales` and `units` variable, and you have specified that all other data should be calculated on the fly.

The `sales` and `units` variables are defined with the same dimensions in the same order: `time`, `product`, and `geography`. Therefore, they have been aggregated with the `sales.agg` aggmap, which has the following definition.

```
DEFINE sales.agg AGGMAP <time, product, geography>
```

The `sales.agg` aggmap has the following contents.

```
RELATION time.r PRECOMPUTES (time ne 'YEAR99')
RELATION product.r PRECOMPUTES (product ne 'ALL')
RELATION geography.r
```

After aggregating both `sales` and `units`, you learn that there are certain geographic regions that none of your users will access. Because `geography` is the slowest-varying dimension, you can probably reduce the number of pages needed to store data by deleting data for the geographic regions that no one will need. This can reduce the size of your analytic workspace and possibly improve performance.

1. Set the status for each dimension. The only geographic regions that users will need are New England, Europe, and Australasia. The following statements put all time periods and all products for every geographic region in the current status, except for the geographic regions that users need. In other words, the following statements put all of the data that users do not need to access in status.

   ```
   LIMIT time TO ALL
   LIMIT product TO ALL
   LIMIT geography COMPLEMENT 'NewEngland' 'Europe' 'Australasia'
   ```

2. Use the following statement to delete the unneeded data.

   ```
   CLEAR STATUS PRECOMPUTES FROM sales units USING sales.agg
   ```

Example 9–6 Clearing Cached Data

Data is cached when an aggmap specifies calculation on the fly and contains a CACHE SESSION statement.

For example, suppose the `sales.agg` aggmap has the following contents.

```
RELATION time.r PRECOMPUTES (time ne 'YEAR99')
RELATION product.r PRECOMPUTES (product ne 'ALL')
RELATION geography.r
CACHE SESSION
```

Note that the `sales.agg` contains a CACHE SESSION command. This means that Oracle OLAP calculates some of the data at the time a user requests it, and then stores it in the session cache. To clear this data from the `sales` variable, use the following statement.

```
CLEAR ALL CACHE FROM sales
```

COALESCE

The COALESCE function returns the first non-`NA` expression in a list of expressions, or `NA` when all of the expressions evaluate to `NA`.

Return Value

Data type of the first argument.

Syntax

COALESCE (*expr* [, *expr*]...)

Arguments

expr
An expression.

COLVAL

The COLVAL function returns a numeric value from a column to the left of the current column in the same row of a report. COLVAL can only be used in the ROW command and the REPORT command.

Return Value

DECIMAL

Syntax

COLVAL(*n*)

Arguments

n

The number of the column in the current row whose value you want; *n* can be any INTEGER expression.

Use a positive number to identify an *absolute* column number (counting left to right from the left margin of the report). For example, `COLVAL(2)` identifies the second column from the left margin of the report.

Use a negative number to identify a *relative* column number (counting right to left from the current column). For example, `COLVAL(-2)` identifies the column that is two columns to the left of the current column.

Notes

Absolute Column Numbers

In figuring an absolute column number, you must count all columns shown in the report. For example, this means that when you are using a REPORT command that produces a column of labels down the left side of the report, you count this column of labels as column 1.

TEXT or ID Data

When the selected column (*n*) contains only a TEXT or ID value, COLVAL returns `NA`.

Error Conditions

An error occurs when you specify the current column, a column to the right of the current column, or a nonexistent column.

Examples

Example 9–7 Performing Column Calculations in a Report

Suppose in a report you want to show actual sales and planned sales, along with the difference between the two. You can use the COLVAL function to calculate this difference.

```
LIMIT month TO 'Jun96'
LIMIT district TO 'Boston'
FOR product
   ROW product sales sales.plan COLVAL(2)-COLVAL(3)
```

These statements produce the following output.

```
Tents           95,120.83  80,138.18  14,982.65
Canoes         157,762.08 132,931.39  24,830.69
Racquets        97,174.44  84,758.46  12,415.98
Sportswear      79,630.20  73,568.52   6,061.68
Footwear       153,688.02 109,219.15  44,468.87
```

COLWIDTH

The COLWIDTH option controls the default width of data columns in report output. For output from the ROW command and HEADING command, COLWIDTH affects all columns except the first column. For output from REPORT, COLWIDTH affects all data columns, as well as the label columns for a composite or a conjoint dimension.

Data type

INTEGER

Syntax

COLWIDTH = *n*

Arguments

n

An INTEGER expression that specifies the desired column width in number of characters. You can use an INTEGER literal or an expression that returns an INTEGER value. The default is `10`.

Notes

Label Columns in REPORT

By default, the REPORT command produces a column of dimension values labeling the rows down the left side of the report. The default width of this label column is controlled by the LCOLWIDTH option, which has a default value of 14 characters. However, when the DOWN phrase in a REPORT command specifies a composite or a conjoint dimension, Oracle OLAP creates a separate column for each base dimension. The default width of these base dimension columns is controlled by the COLWIDTH option.

Maximum Column Width

You can set COLWIDTH to any value from `1` to `4000`.

Note: The maximum width of a line in a report is 4000 characters. Therefore, the combined width of all the columns of a report cannot be greater than 4000 characters.

Overriding COLWIDTH

For an individual column, the COLWIDTH value is always overridden by a WIDTH attribute in a HEADING, REPORT, or ROW command.

Examples

Example 9–8 Setting the Default Column Width in a Report

Suppose you want to look at unit sales for six months. Since the data values are not large, you do not need a width of 10 characters for your data columns. You can set COLWIDTH to provide a narrower default column.

```
LIMIT district TO 'Atlanta'
LIMIT month TO 'Oct95' TO 'Mar96'
COLWIDTH = 6
REPORT ACROSS month: units
```

These statements produce the following output.

```
DISTRICT: ATLANTA
               ------------------UNITS------------------
               ------------------MONTH------------------
PRODUCT        Oct95  Nov95  Dec95  Jan96  Feb96  Mar96
-------------- ------ ------ ------ ------ ------ ------
Tents             503    345    259    279    305    356
Canoes            317    282    267    281    309    386
Racquets        1,365  1,270  1,357  1,125  1,304  1,263
Sportswear      3,065  2,327  1,955  2,591  2,829  3,137
Footwear        3,445  3,247  2,831  3,089  3,282  3,475
```

COMMAS

The COMMAS option controls the use of a character to separate thousands and millions in numeric output. COMMAS affects all commands that produce output, including the ROW command as well as HEADING, REPORT, and SHOW.

Data type

BOOLEAN

Syntax

COMMAS = {NO|YES}

Arguments

NO
Numeric output does not contain a character that separates thousands, millions, and so on.

YES
Numeric output contains a character that separates thousands, millions, and so on. (Default)

Notes

Overriding COMMAS
You can use the COMMA and NOCOMMA attributes of a HEADING, REPORT, or ROW command to override the COMMAS setting.

Setting the Thousands Separator
The character that separates thousands and millions in numeric output is normally a comma. However, it might be different depending on your NLS_TERRITORY setting. The THOUSANDSCHAR option records the character that is currently being used for separating thousands. The COMMAS option controls whether the character appears in numeric output.

Examples

Example 9–9 Showing Numerical Data Without Commas

Suppose you want to look at the cost of goods sold, without commas in the data values. You can set COMMAS to `NO` before producing your report.

```
COMMAS = NO
LIMIT line TO 'Cogs'
LIMIT month TO 'Jan96' 'Feb96'
REPORT DOWN division ACROSS month: DECIMAL 0 actual
```

These statements produce the following output.

```
LINE: COGS
               -----ACTUAL------
               ------MONTH------
DIVISION       Jan96      Feb96
-------------- -------- ----------
Camping          368044     385120
Sporting         287558     315299
Clothing         567767     610727
```

COMMIT

The COMMIT command executes a SQL `COMMIT` command. All changes made in your database session are committed, whether they were made through Oracle OLAP or through another form of access (such as SQL) to the database.

When you want changes that you have made in a workspace to be committed when you execute the COMMIT command, then you must first update the workspace using the UPDATE command. UPDATE moves changes from a temporary work area to the database table in which the workspace is stored. Changes that have not been moved to the table are not committed.

The COMMIT command only affects changes in workspaces that you have attached in read/write access mode. After the command returns, all committed changes are visible to other users who subsequently attach the workspace.

Syntax

```
COMMIT
```

Notes

Unsaved Changes

When you do not use the UPDATE and COMMIT commands, changes made to an analytic workspace during your session are discarded when you end your Oracle session.

Note: You can detach and reattach a workspace without losing updated changes, even though they are not committed. This is because the detaching and reattaching occur within a single database session.

SQL COMMIT Statement

When you execute a SQL `COMMIT` statement in your session outside Oracle OLAP, this statement commits all updated changes in workspaces that you have attached with read/write access.

Automatic COMMIT

Many users execute DML statements using SQL*Plus® or OLAP Worksheet. Both of these tools automatically execute a COMMIT statement when you end your session.

Shared Workspaces

When you have attached a shared workspace and another user has read/write access, that user's UPDATE and COMMIT commands do not affect your view of the workspace. Your view of the data remains the same as when you attached the workspace. When you want access to the changes, you can detach the workspace and reattach it.

Examples

Example 9–10   Saving All Changes to an Analytic Workspace

The following statements permanently save all analytic workspace changes made so far in your session. The COMMIT command also saves database changes made in your session outside Oracle OLAP.

```
UPDATE
COMMIT
```

COMPILE

The COMPILE command generates compiled code for a compilable object, such as a program, formula, model, or aggmap without running it and saves the compiled code in the analytic workspace. During compilation, COMPILE checks for format errors, so you can use COMPILE to help debug your code before running it. COMPILE records the errors in the current outfile.

However, you are not required to use the COMPILE command before running a compilable object. When you do not use the COMPILE command, Oracle OLAP automatically compiles a compilable object the first time you run it after entering or changing its contents. This automatic compilation is unnoticeable except for a slight delay while it is happening.

Whether you compile an object explicitly with COMPILE or automatically through running it, the code executes faster whenever you subsequently run the object during the same session, because the code is already compiled. When you update and commit your analytic workspace, the compiled code is saved as part of your analytic workspace and can be used in later sessions. The code thus executes faster the first time it is run in each later session.

Using the COMPILE command to compile code without running a compilable object is especially useful when you are writing code that will be part of a read-only analytic workspace (that is, a analytic workspace that people can use but not update).

Syntax

COMPILE *object-name*

Arguments

object-name
The name of a compilable object that you want to compile.

Notes

Compilation Options

A number of options effect compilation. These options are listed in Table 9–1, " Compilation Options" on page 9-30.

Table 9–1 Compilation Options

Statement	Description
COMPILEMESSAGE	An option that specifies whether you want Oracle OLAP to send to the current outfile non-fatal messages during execution of the COMPILE command.
COMPILEWARN	An option that controls whether Oracle OLAP records a warning message in the current outfile when a compilable object, such as an OLAP DML program or a model, is being compiled automatically.
THIS_AW	A read-only option that is the value of the workspace name that Oracle OLAP uses when it replaces occurrences of the THIS_AW keyword to create a qualified object name.

Deleted Objects

When you delete or rename an object in your analytic workspace, Oracle OLAP automatically invalidates the compiled code for every statement in a program and every formula and model that depends on that object. When you try to execute code that refers to the deleted or renamed object, Oracle OLAP tries to compile the code again. Unless you have defined a new object with the same name, you will receive an error message at this time.

When you run a program that contains invalidated code, it is compiled and executed one statement at a time. To save compiled code for the entire program, use the COMPILE command to explicitly compile it.

Multiple Errors in a Line

When a single statement has more than one error, COMPILE finds only the first error. However, COMPILE continues checking for format errors in subsequent statements.

Advantages of Compiling

Explicit compilation using the COMPILE command offers several advantages over automatic compilation:

- For any compilable object, COMPILE generates compiled code without executing the code in the object.
- In a program or model, automatic compilation diagnoses an error only in the first statement that contains an error. It then displays the error message and

halts the execution of a program or the analysis of a model. So each time a program or model is automatically compiled, only a single error message is displayed. In contrast, COMPILE checks every statement in a program or model for correct format, and generates multiple error messages, one for each statement that contains an error. (In programs, some types of statements cannot be compiled, so they are exceptions. See "Errors COMPILE Does Not Catch" on page 9-31.) Since COMPILE shows you every statement that contains at least one error, this minimizes the number of times you must edit the code to correct all errors.

- For a model, you may want to examine the results of the compilation or set options for handling simultaneous equations before you run the model.

Errors COMPILE Does Not Catch

Because the COMPILE command does not actually execute code, it can compile code that, for reasons unrelated to format errors, might not be successfully executed when the object were actually run. In a program, for example, you can compile the following statement, even though 'joplin' is not a district.

```
LIMIT district TO 'joplin'
```

Although the statement compiles successfully, you will get an error message at runtime.

Commands Not Compiled

In programs, certain statements cannot be compiled at all, and are therefore interpreted each time they are executed. These include statements that contain ampersand substitution, statements involving analytic workspace operations, and any statement that calls a program as a command. (Statements that call a program as a function or with the CALL command are compiled.)

PRGTRACE Option

You can use the PRGTRACE option to check which statements in a program have been compiled. When you set PRGTRACE to `YES` and run a program, each statement is recorded in the current outfile before it is executed. A compiled statement is identified with an equal sign.

```
(PRG= program-name) statement
```

An uncompiled statement is identified with a colon.

```
(PRG: program-name) statement
```

Compiling Models

You can use the COMPILE command to compile a model. When you do not use the COMPILE command before you run the model, Oracle OLAP automatically compiles it before solving it. You can use the OBJ function with the ISCOMPILED choice to test whether a model is compiled.

```
SHOW OBJ(ISCOMPILED 'myModel')
```

When you compile a model, Oracle OLAP checks for problems that are unique to models. You receive an error message when any of the following occurs:

- The model contains any statements other than DIMENSION (in models), INCLUDE, and assignment (SET) statements.
- The model contains both a DIMENSION command and an INCLUDE command.
- A DIMENSION or INCLUDE command is placed after the first equation in the model.
- The dimension values in a single dimension-based equation refer to two or more different dimensions.
- An equation refers to a name that the compiler cannot identify as an object in any attached analytic workspace. When this error occurs, it may be because an equation refers to the value of a dimension, but you have neglected to include the dimension in a DIMENSION command. In addition, a DIMENSION command may appear to be missing when you are compiling a model that includes another model and the other model fails to compile. When a root model (the innermost model in a hierarchy of included models) fails to compile, the parent model is unable to inherit any DIMENSION commands from the root model. In this case the compiler may report an error in the parent model when the source of the error is actually in the root model. See INCLUDE for additional information.

Resolving Names in Equations The model compiler examines each name in an equation to determine the analytic workspace object to which the name refers. Since you can use a variable and a dimension value in exactly the same way in a model equation (basing calculations on it or assigning results to it), a name might be the name of a variable or it might be a value of any dimension listed in a DIMENSION (in models) statement.

To resolve each name reference, the compiler searches through the dimensions listed in explicit or inherited DIMENSION statements, in the order they are listed, to determine whether the name matches a dimension value of a listed dimension. The search concludes as soon as a match is found.

Therefore, when two or more listed dimensions have a dimension value with the same name, the compiler assumes that the value belongs to the dimension named earliest in a DIMENSION statement.

Similarly, the model compiler might misinterpret the dimension to which a literal integer value belongs. For example, the model compiler will assume that the literal value `'200'` belongs to the first dimension that contains either a value at position 200 or the literal dimension value 200.

To avoid an incorrect identification, you can specify the desired dimension and enclose the value in parentheses and single quotes. See "Formatting Ambiguous Dimension Values" on page 21-64.

When the compiler finds that a name is not a value of any dimension specified in a DIMENSION statement, it assumes that the name is the name of an analytic workspace variable. When a variable with that name is not defined in any attached analytic workspace, an error occurs.

Code for Looping Over Dimensions The model compiler determines the dimensions over which the statements will loop. When an equation assigns results to a variable, the compiler constructs code that loops over the dimensions (or bases of a composite) of the variable.

When you run a model that contains dimension-based equations, the solution variable that you specify can be dimensioned by more dimensions than are listed in DIMENSION (in models) statements.

Evaluating Program Arguments When you specify the value of a model dimension as an argument to a user-defined program, the compiler recognizes a dependence introduced by this argument.

For example, an equation might use a program named `weight` that tests for certain conditions and then weights and returns the `Taxes` line item based on those conditions. In this example, a model equation might look like the following one.

```
Net.Income = Opr.Income - weight(Taxes)
```

The compiler correctly recognizes that `Net.Income` depends on `Opr.Income` and `Taxes`. However, when the `weight` program refers to any dimension values or variables that are not specified as program arguments, the compiler does not detect any hidden dependencies introduced by these calculations.

Dependencies Between Equations The model compiler analyzes dependencies between the equations in the model. A dependence exists when the expression on the right-hand side of the equal sign in one equation refers to the assignment target of another equation. When an equation indirectly depends on itself as the result of the dependencies among equations, a cyclic dependence exists between the equations.

The model compiler structures the model into blocks and orders the equations within blocks and the blocks themselves to reflect dependencies. When you run the model, it is solved one block at a time. The model compiler can produce three types of solution blocks:

- **Simple Solution Blocks**—Simple blocks are one of the three types of solution blocks that the model compiler can produce. Simple blocks include equations that are independent of each other and equations that have dependencies on each other that are non-cyclic.

  For example, when a block contains equations that solve for values `A`, `B`, and `C`, a non-cyclic dependence can be illustrated as `A>B>C`. The arrows indicate that `A` depends on `B`, and `B` depends on `C`.

- **Step Solutions Blocks**—Step blocks are one of the three types of solution blocks that the model compiler can produce. Step blocks include equations that have a cyclic dependence that is a one-way dimensional dependence. A dimensional dependence occurs when the data for the current dimension value depends on data from previous or later dimension values. The dimensional dependence is one-way when the data depends on previous values only or later values only, but not both.

  Dimensional dependence typically occurs over a time dimension. For example, it is common for a line item value to depend on the value of the same line item or a different line item in a previous time period. When a block contains equations that solve for values `A` and `B`, a one-way dimensional dependence can be illustrated as `A>B>LAG(A)`. The arrows indicate that `A` depends on `B`, and `B` depends on the value of `A` from a previous time period.

- **Simultaneous Solution Blocks**—Simultaneous blocks are one of the three types of solution blocks that the model compiler can produce.

  When a model contains a block of simultaneous equations, COMPILE gives you a warning message. In this case, you may want to check the settings of the options that control simultaneous solutions before you run the model. Table 17–1, "Model Options" on page 17-23 lists these options.

Simultaneous blocks include equations that have a cyclic dependence that is other than one-way dimensional. The cyclic dependence may involve no dimensional qualifiers at all, or it may be a *two-way dimensional* dependence.

An example of a cyclic dependence that does not depend on any dimensional qualifiers can be illustrated as `A>B>C>A`. The arrows indicate that `A` depends on `B`, `B` depends on `C`, and `C` depends on `A`.

An example of a cyclic dependence that is a two-way dimensional dependence can be illustrated as `A>LEAD(B)>LAG(A)`. The arrows indicate that `A` depends on the value of `B` from a future period, while `B` depends on the value of `A` from a previous period.

Order of Simultaneous Equations The solution of a simultaneous block of equations is sensitive to the order of the equations. In general, you should rely on the model compiler to determine the optimal order for the equations. In some cases, however, you may be able to encourage convergence by placing the equations in a particular order.

To force the compiler to leave the simultaneous equations in each block in the order in which you place them, set the MODINPUTORDER option to `YES` before compiling the model. (MODINPUTORDER has no effect on the order of equations in simple blocks or step blocks.)

One-Way Dimensional Dependence

When dependence between equations is introduced through any of the following structures, a one-way dimensional dependence occurs:

- A one-way dimensional dependence can occur when you use a LAG or LEAD function and when the argument for the number of time periods is a number. (Otherwise, there may be a two-way dependence, involving both previous and future dimension values, and the compiler assumes that a simultaneous solution is required.) The following example illustrates the use of LAG.

  ```
  Opr.Income = Gross.Margin - (Marketing + Selling + R.D)
  Marketing = LAG(Opr.Income, 1, month)
  ```

- A one-way dimensional dependence also can occur when you use a MOVINGAVERAGE, MOVINGMAX, MOVINGMIN, or MOVINGTOTAL function, when that the start and stop arguments are nonzero numbers, and when both the start and top arguments are positive *or* both are negative. (Otherwise, two-way dimensional dependence is assumed.)

```
Opr.Income = Gross.Margin - (Marketing + Selling + R.D)
Marketing = MOVINGAVERAGE(Opr.Income, -4, -1, 1, month)
```

Two-Way Dimensional Dependence

When dependence is introduced through any of the following structures, the model compiler assumes that two-way dimensional dependence occurs:

- A two-way dimensional dependence can occur when you use an aggregation function, such as AVERAGE, TOTAL, ANY, or COUNT.

  ```
  Opr.Income = Gross.Margin -
     (TOTAL(Marketing + Selling + R.D))
  Marketing = LAG(Opr.Income, 1, month)
  ```

- A two-way dimensional dependence can occur when you use a time-series function that requires a time-period argument, such as CUMSUM or LAG (except for the specific functions and conditions described in "One-Way Dimensional Dependence" on page 9-35.
- A two-way dimensional dependence also can occur when you use a financial function, such as DEPRSL or NPV.

  A cyclic dependence across a time dimension that you introduce through a loan or depreciation function may cause unexpected results. The loan functions include FINTSCHED, FPMTSCHED, VINTSCHED, and VPMTSCHED. The depreciation functions include DEPRDECL, DEPRDECLSW, DEPRSL, and DEPRSOYD.

Obtaining Analysis Results After compiling a model, you can use the following tools to obtain information about the results of the analysis performed by the compiler:

- The MODEL.COMPRPT program produces a report that shows how model equations are grouped into blocks. For step blocks and for simultaneous blocks with a cross-dimensional dependence, the report lists the dimensions involved in the dependence.
- The MODEL.DEPRT program produces a report that lists the variables and dimension values on which each model equation depends. When a dependence

is dimensional, the report gives the name of the dimension.

- The INFO function lets you obtain specific items of information about the structure of the model.

Multiple Analytic Workspaces

When you compile a compilable object that uses objects in another analytic workspace, the second analytic workspace must be attached to your current Oracle OLAP session. You can then run the compilable object with that analytic workspace or another analytic workspace with objects of the same name and type attached. Oracle OLAP checks that the objects have the same name, type (variable, dimension, and so on), data type (INTEGER, TEXT, and so on), and dimensions as the objects used to compile the compilable object.

When you have more than one active analytic workspace, *do not* have objects of the same name in both analytic workspaces. For example, when you have an analytic workspace of programs and two analytic workspaces with data about the products TEA and COFFEE, both product analytic workspaces can have a MONTH dimension and the programs can refer to MONTH. However, during your session, attach only one product analytic workspace at a time so that there is only one MONTH dimension.

OBJ Function

Use the OBJ function with the ISCOMPILED keyword to obtain information about the compilation status of a compilable object.

COMPILEWARN Option

To have Oracle OLAP display a message when it compiles an object automatically, you can set the COMPILEWARN option to `YES`.

COMPILEMESSAGE Option

You use the COMPILEMESSAGE option to specify whether you want Oracle OLAP to record non-fatal messages (those messages that indicate errors that do not prevent a program from compiling) during execution of the COMPILE command.

Memory Use

In order for code to compile, all variables referenced in a program (with the exception of variables in lines containing ampersand substitution) must be loaded into memory. This means Oracle OLAP reads the definition of every variable you use and stores it in a portion of available memory that is dedicated for storing object definitions. When the compilation tries to bind a large variable, this may use a large amount of memory and create a large `EXPTEMP` file. When the compilation tries to bind a large number of large variables, it may fail and Oracle OLAP will record an error message such as 'Insufficient Main Memory'. See LOAD for more information about loading an object's definition into memory.

Compiling Aggregation Specifications

Compiling the aggmap object is important for aggregation performed at run-time using the AGGREGATE function. Unless the compiled version of the aggmap has been saved, the aggmap is recompiled by each session that uses it.

There are two ways you can compile an aggmap objects:

- Issue a COMPILE statement.

  A COMPILE statement is the only way to compile an aggmap object that will be used by an AGGREGATE function. Explicitly compiling an aggmap is also useful for finding syntax errors in the aggmap before attempting to use it to generate data. The following statement compiles the `sales.agg` aggmap.

  ```
  COMPILE gpct.aggmap
  ```

- When you aggregate the data using an AGGREGATE command, include the FUNCDATA phrase in the statement.

  When you use the FUNCDATA phrase in an AGGREGATE command, Oracle OLAP compiles the aggmap before it aggregates the data. For example, this statement compiles and precalculates the aggregate data.

  ```
  AGGREGATE sales USING gpct.aggmap FUNCDATA
  ```

Important: When some of the data is calculated on the fly, then you must compile and save the aggmap *after* executing the AGGREGATE command.

Examples

Example 9–11 Compiling a Program

The following is an example of a `COMPILE` command that compiles the `myprog` program.

```
COMPILE myprog
```

Suppose you misspell the dimension `month` in a LIMIT command in the `myprog` program.

```
LIMIT motnh TO LAST 6
```

When the `COMPILE` command encounters this command, it produces the following message.

```
ERROR: (MXMSERR00) Analytic workspace object MOTNH does not exist.
In DEMO!MYPROG PROGRAM:
limit month to last 6
```

You can edit the program to correct the error and then try to compile it again.

Example 9–12 Finding Program Errors

This example shows a program called `salesrpt` that contains two errors.

```
DEFINE salesrpt PROGRAM
PROGRAM
ROW WIDTH 80 CENTER Monthly Report
BLANK 2
ROWW 'Total Sales' TOTAL(sales)
END
```

You can compile the program with the following statement.

```
COMPILE salesrpt
```

Oracle OLAP identifies both errors and records the following messages.

```
ERROR: You provided extra input starting at 'REPORT'.
In SALESRPT PROGRAM:
ROW WIDTH 80 CENTER Monthly Report
ERROR: ROWW is not a command.
In SALESRPT PROGRAM:
roww 'Total Sales' TOTAL(sales)
```

You can now edit the program to correct these errors, enclosing `'Monthly Report'` in single quotes and correcting the spelling of `ROWW`. Then you can compile the program again, and save the compiled code as part of your analytic workspace.

COMPILEMESSAGE

You use the COMPILEMESSAGE option to specify whether you want Oracle OLAP to send to the current outfile non-fatal messages during execution of the COMPILE command. Non-fatal messages are those indicating errors that do not prevent a program from compiling.

> **See also:** For more information about compiling objects, see COMPILE.

Data type

BOOLEAN

Syntax

COMPILEMESSAGE = {<u>YES</u>|NO}

Arguments

YES
Indicates that Oracle OLAP should record non-fatal messages during execution of the COMPILE command. (Default)

NO
Indicates that Oracle OLAP should suppress non-fatal messages during execution of the COMPILE command.

Examples

Example 9–13 Suppressing Error Messages During Compilation

The following statement specifies that Oracle OLAP should suppress non-fatal messages during execution of the COMPILE command.

```
COMPILEMESSAGE = NO
```

COMPILEWARN

The COMPILEWARN option controls whether Oracle OLAP records a warning message in the current outfile when a compilable object, such as a program or a model, is being compiled automatically. When a compilable object has been changed since the last time it was compiled or run, Oracle OLAP automatically compiles it when you execute it.

Data type

BOOLEAN

Syntax

```
COMPILEWARN = {YES|NO}
```

Arguments

YES
Oracle OLAP records a message warning you that a compilable object is being compiled automatically. The message explains why the compilation was necessary.

NO
Oracle OLAP does not record a message warning you that an object is being compiled automatically. (Default)

Notes

Slower Response Time

Developing an Oracle OLAP application involves repeated editing of objects that must be recompiled each time you test them. The compile warning lets you know that the slower response of the application is because it is compiling code, and not because of problems with the application. In deeply nested applications, you may not even be aware that an object with new or revised code has been called.

Conditions for Automatic Compilation

A compilable object will be automatically compiled in the following cases:

- The first time it is executed after being edited.
- The first time it is executed in a session when it was compiled in a previous session after the last time the analytic workspace was updated and committed.
- After an analytic workspace object referred to in the code has been renamed or deleted. When the object name in the code has not been redefined, you will receive an error message.
- When the code refers to objects in another analytic workspace and the objects in the currently attached analytic workspace do not have the same object type (variable, relation, and so on), data type (INTEGER, TEXT, and so on), or dimensions as the objects available when the code was previously compiled.

Updating Your Analytic Workspace

When you receive the compile warning, you should update and commit your analytic workspace so the compiled code is saved as part of your analytic workspace and can be used in later sessions.

COMPILE Command

When you use the COMPILE command to compile an object, Oracle OLAP does not display the COMPILEWARN message.

OBJ Function

Use the OBJ function with the ISCOMPILED keyword to obtain information about the compilation status of a compilable object.

Examples

Example 9–14 Specifying That You Want Compiler Warnings

When COMPILEWARN is set to `YES`, when you run the `do_report` program just after editing it, Oracle OLAP places the following message in your current outfile before the `do_report` output.

```
DO_REPORT is being automatically compiled.
```

CONSIDER

The CONSIDER command identifies a definition as the current definition. This enables you to add a description, value name format, formula, program, model, permission, or property to the definition with an LD, VNF, EQ, PROGRAM, MODEL, PERMIT, or PROPERTY command.

Syntax

CONSIDER *name*

Arguments

name
The name of a definition in the current workspace or in an attached workspace.

Notes

Replacing a Definition Component

When you use an LD, VNF, EQ, PROGRAM, MODEL, or PERMIT command to add a component to the current definition, any existing value for that component is discarded and replaced by the new value you specify. For the PROPERTY command, the value is replaced only when you specify a new value for an existing property name. Definitions can have more than one property.

Unsuccessful CONSIDER Commands

When the CONSIDER command you issue is unsuccessful, subsequent LD, VNF, EQ, PROGRAM, MODEL, PERMIT, or PROPERTY commands produce an error.

Implicit CONSIDER Commands

The DEFINE, COPYDFN, and RENAME command automatically issue an implicit CONSIDER command.

Examples

Example 9–15 Adding a Description to an Analytic Workspace Object

This example adds a description (LD) to the definition for `district`. To add the LD, you must first use CONSIDER to make `district` the current definition. The statements

```
CONSIDER district
LD Sales Districts
DESCRIBE district
```

produce the following definition.

```
DEFINE district DIMENSION TEXT
LD Sales Districts
```

CONTEXT command

The CONTEXT command lets you create and use a context during your Oracle OLAP session. A context is a means of preserving object values. After you create a context, you can save the current status of dimensions and the values of options, single-cell variables, valuesets, and single-cell relations in the context. You can then restore some or all of the object values from the context.

You can use the CONTEXT function to obtain information about a context.

The CONTEXT command and function provide an alternative to the PUSH and POP commands. With contexts, you can access and update the saved object values, whereas PUSH and POP simply allow you to save and restore values.

Syntax

CONTEXT *context-name* [CREATE | APPLY | DISCARD | {SAVE |DROP|RESTORE} *objects*]

Arguments

context-name
A text expression that contains the name of the context.

CREATE
Creates a context with the name specified by *context-name,* which must be unique.

SAVE
Stores the values of the objects specified in *objects* in the context.

APPLY
Sets the appropriate objects to the values of all corresponding objects saved in the context.

DISCARD
Deletes the context.

SAVE
Stores the values of the objects specified in *objects* in the context.

DROP
Drops the values of the objects specified in *objects* from the context.

RESTORE
Sets whatever objects you specify in *objects* to the values of the corresponding objects saved in the context.

objects
One or more object names. Each object name must be separated by a space.

Notes

Persistence of a Context
A context exists only for the duration of an Oracle OLAP session. It is not an analytic workspace object and therefore cannot be saved as part of any analytic workspace.

Deleted Objects
When you delete an Oracle OLAP object during the session, it is also removed from the context.

Detached Analytic Workspaces
When a context contains saved values for objects in a particular analytic workspace, and you detach that analytic workspace, Oracle OLAP removes those objects from the context. That context retains any saved values for Oracle OLAP options, as well as objects from other analytic workspaces that are still attached.

NAME Dimension
You cannot use the CONTEXT command to save the values of the NAME dimension. When you include NAME in the list of *name(s)* that you specify with the SAVE keyword, Oracle OLAP produces an error message.

Long Lines
When you are listing several *name(s)* that will not fit on a single line, you may use the continuation character to continue the CONTEXT command on additional lines.

Dimensioned Objects
You may save the values of single-cell variables and relations in a context. When you try to save a dimensioned variable or relation, Oracle OLAP produces an error message.

Naming Convention

A suggested programming practice is to name the context after the analytic workspace with which it is associated.

Examples

Example 9–16 Saving Dimension Status

This example shows how you can use the CONTEXT command to save and restore the status of a dimension. The following statements create a context that includes a subset of the values in the `product` dimension.

```
LIMIT product TO 'Tents' 'Canoes'
CONTEXT 'democontext1' CREATE
CONTEXT 'democontext1' SAVE product
```

The following statements limit `product` to all its values and produce a report that lists them all.

```
LIMIT product TO ALL
REPORT product
```

This is the report.

```
PRODUCT
-----------
Tents
Canoes
Racquets
Sportswear
Footwear
```

The following statements apply the saved context and produce a report that lists only the values included in the context.

```
CONTEXT 'democontext1' APPLY
REPORT product
```

This is the new report.

```
PRODUCT
-----------
Tents
Canoes
```

CONTEXT function

The CONTEXT function lets you obtain information about object values that are saved in a context. You must first create the context with the CONTEXT command.

Return Value

The data type of the return value of the CONTEXT function depends on the arguments you provide. When you use the CONTEXT function without supplying any arguments, it returns a multiline text value that contains the names of all the contexts in the current session.

Syntax

CONTEXT ([*context-name* [UPDATE|*name*]])

Arguments

context-name
A text expression that contains the name of the context. Using the CONTEXT function with only the context-name returns a multiline text value that contains the names of all the objects saved in that context.

UPDATE
When you specify UPDATE with the CONTEXT function, the return value is the number of times values have been saved or dropped from the context.

name
The name of an object whose value is saved in the context. When you specify *name* with the CONTEXT function, the return value is the saved status or value of that object.

Notes

Persistence of a Context

A context exists only for the duration of an Oracle OLAP session. It is not an analytic workspace object and therefore cannot be saved as part of any analytic workspace.

Detached Analytic Workspaces

When a context contains saved values for objects in a particular analytic workspace, and you detach that analytic workspace, then Oracle OLAP removes those objects from the context. That context retains any saved values for Oracle OLAP options, as well as objects from other analytic workspaces that are still attached.

Examples

Example 9–17 Listing Context Names

In the following statement, the CONTEXT function returns the name of the only context in the current session. This is the same context used in "Saving Dimension Status" on page 9-48.

```
SHOW CONTEXT
```

The statement produces the following output.

```
democontext1
```

Example 9–18 Listing Objects in a Context

In the following statement, the CONTEXT function returns the name of the only object included in the context named `democontext1`.

```
SHOW CONTEXT('democontext1')
```

The statement produces the following output.

```
PRODUCT
```

Example 9–19 Listing Saved Values

In the following statement, the CONTEXT function returns the values of the `product` dimension that are saved in the context named `democontext1`.

```
SHOW CONTEXT('democontext1' product)
```

The statement produces the following output.

```
Tents
Canoes
```

CONTINUE

The CONTINUE command transfers program control to the end of a FOR or WHILE loop (just before the DO/DOEND command), allowing the loop to repeat. You can use CONTINUE only within programs and only with FOR or WHILE.

For more information on controlling program execution, see also BREAK, FOR, SWITCH, and WHILE.

Syntax

CONTINUE

Examples

Example 9–20 Skipping Over Code in a FOR Loop

In the following lines from a program, an IF command is used to test whether total sales for a district exceed 5,000,000. When sales are more this amount, the program goes on to produce a report for that district. However, when a district's sales are less than the amount, the CONTINUE command is used to transfer control to the end of the FOR loop (just before the DOEND command). No lines are produced for that district, and the program goes on to test the next district in the status list.

```
 ...
FOR district
    DO
    IF TOTAL(sales, district) LT 5000000
      THEN CONTINUE
     ... "(report commands for districts with total sales above 5,000,000)
    DOEND
 ...
```

CONVERT

The CONVERT function converts values from one type of data to another. CONVERT is primarily useful for changing values from a numeric or DATE data type to a text data type, or vice versa.

Return Value

The return value depends on the value of the *type* argument.

Syntax

CONVERT(*expression*, *type* [*argument...*])

Arguments

expression
The expression or variable to be converted.

type
The type of data to which you want to convert *expression.* The keywords that represent the types are described in Table 9–2, " Keywords for the type Argument of the CONVERT Function".

Table 9–2 Keywords for the type Argument of the CONVERT Function

Keyword	Description
TEXT	Conversion to standard Oracle OLAP data types. Corresponds to CHAR and VARCHAR2 data types in the Oracle relational database. A TEXT character is encoded in the database character set.
NTEXT	Conversion to standard Oracle OLAP data types. Corresponds to the NCHAR and NVARCHAR2 data types of the relational database. An NTEXT character is encoded in `UTF8` Unicode. This encoding might be different from the NCHAR character set of the database, which can be `UTF16`. A conversion from NTEXT to TEXT can result in data loss when the NTEXT value cannot be represented in the database character set.
ID	Conversion to standard Oracle OLAP data types.
DATE	Conversion to standard Oracle OLAP data types.

Table 9–2 (Cont.) Keywords for the type Argument of the CONVERT Function

Keyword	Description
NUMBER	Conversion to standard Oracle OLAP data types.
BOOLEAN	Conversion to standard Oracle OLAP data types.
INTEGER	Conversion to standard Oracle OLAP data types.
SHORTINTEGER	Conversion to standard Oracle OLAP data types.
LONGINTEGER	Conversion to standard Oracle OLAP data types.
DECIMAL	Conversion to standard Oracle OLAP data types.
SHORTDECIMAL	Conversion to standard Oracle OLAP data types.
DATETIME	Conversion to standard Oracle OLAP data types.
BYTE	Converts a single character into an ASCII integer value in the range 0 to 255. Or BYTE converts an INTEGER within this range into a character. An INTEGER outside this range is taken *modulo* 256 and then converted; that is, 256 is subtracted from the INTEGER until the remainder is less than 256, and that within-range remainder is then converted into a character.
INFILE	Encloses an ID, TEXT, DATE, or RELATION value within single quotes, so that it can be read with an INFILE command. This means that *expression* must have TEXT, ID, DATE, or RELATION values. In the case of TEXT values with no alphanumeric equivalent, INFILE converts them to the correct escape sequences.
PACKED	Converts a number to a decimal value and then to packed format -- a text value 8 bytes long containing 15 digits and a plus or minus sign. Fractions cannot be represented in packed numbers; therefore the conversion process rounds decimal numbers to the nearest integer. See "PACKED and BINARY Conversion" on page 9-61.

Table 9–2 (Cont.) Keywords for the type Argument of the CONVERT Function

Keyword	**Description**
BINARY	Does not indicate conversion to a standard Oracle data type but allows additional conversion capabilities. BINARY does no conversion. The internal representation of every value, regardless of data type, is returned as a text value. For TEXT data types, the result will be the value itself and will therefore be of variable length. For ID and DECIMAL data types, the result will be 8 bytes long; ID values will be blank filled, when necessary. For BOOLEAN or INTEGER, the default result will be 2 or 4 bytes long respectively (see the arguments explanation for an additional argument that lets you vary the width slightly). For all other data types, the result will be 4 bytes long. See "PACKED and BINARY Conversion" on page 9-61.

argument

When you specify TEXT, NTEXT, ID, DATE, or INFILE for the *type*, you can specify additional arguments to determine how the conversion should be done:

- Numeric values to TEXT values

  TEXT [*decimal-int*|DECIMALS [*comma-bool*|COMMAS [*paren-bool*|PARENS]]]

- Numeric values to NTEXT values

  NTEXT [*decimal-int*|DECIMALS [*comma-bool*|COMMAS [*paren-bool*|PARENS]]]

- Numeric values to ID values

  ID [*decimal-int*|DECIMALS]

- DATE values to TEXT, NTEXT, or ID values

  ID|TEXT|NTEXT ['*date-format*']

- A DATE value or the values of a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR with VNF

  ID [*dwmqy-dimension*]|TEXT [*dwmqy-dimension*|'*vnf*']

- TEXT, NTEXT, or ID values to DATE values

  DATE [*date-order*|*dwmqy-dimname*]

- INFILE conversion

  INFILE [*width-exp*|LSIZE [*escape-int*|0]]

- When specifying BINARY with BOOLEAN or INTEGER data

  BINARY [*width-exp*]

- Onverting between TEXT and NTEXT values

  NOXLATE

decimal-int
An INTEGER expression that controls the number of decimal places to be used when converting numeric data to TEXT or ID values. When this argument is omitted, CONVERT uses the current value of the DECIMALS option (the default is 2).

comma-bool
A Boolean expression that determines whether commas are used to mark thousands and millions in the text representation of the numeric data. When the value of the expression is `YES`, commas are used. When this argument is omitted, CONVERT uses the current value of the COMMAS option (the default is `YES`).

paren-bool
A Boolean expression that determines whether negative values are enclosed in parentheses in the text representation of the numeric data. When the value of the expression is `YES`, parentheses are used; when the value is `NO`, a minus sign precedes negative values. When this argument is omitted, CONVERT uses the current value of the PARENS option (the default is `NO`).

date-format
A text expression that specifies the template to use when converting a DATE expression to text. The template can include format specifications for any of the four components of a date (day, month, year, and day of the week). Each component in the template must be preceded by a left angle bracket (<)and followed by a right angle bracket (>). You can include additional text before, after, or between the components.

The valid formats for each date component are the same as the formats allowed in the DATEFORMAT option.

In the following statement, CONVERT returns today's date as a text value that is formatted by a *date-format* argument.

```
SHOW CONVERT(TODAY TEXT '<MM>-<DD>-<YY>')
```

In this example, today's date is March 31, 1998, and the SHOW command presents it in the following format.

```
03-31-98
```

When you do not include the *date-format* argument, the format of the result is determined by the current setting of the DATEFORMAT option.

dwmqy-dimension
The name of a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR. Oracle OLAP uses the VNF of *dwmqy-dimension* when converting a DATE value to a TEXT or an ID value. When you have not specified the VNF of *dwmqy-dimension,* Oracle OLAP uses its default VNF.

In the following statement, CONVERT returns today's date as a text value that is formatted by the VNF of the YEAR dimension.

```
show convert(today text year)
```

In this example, today's date is March 31, 1998, and the SHOW command presents it in the following format.

```
YR98
```

vnf
A text template that specifies the value name format to use when converting values of a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR to text. The template can include format specifications for any of the components of a time period. Time period components include all the components of a date (day, month, year, and day of the week), plus the fiscal year and period components. The template can also include the name of the DAY, WEEK, MONTH, QUARTER, or YEAR dimension as a component. Each component in the template must be preceded by a left angle bracket and followed by a right angle bracket. You can include additional text before, after, or between the components.

The *vnf* argument to the CONVERT function is similar to the template in a VNF command. However, a VNF command template must be designed for precise and unambiguous interpretation of input, while the *vnf* argument is not so constrained. Therefore, the format styles allowed in the *vnf* argument are more extensive than those allowed in a VNF command template.

Valid format styles for a *vnf* argument include all the format styles allowed in the template of a VNF command, plus all the format styles allowed in a DATEFORMAT template. DATEFORMAT provides the following format styles that are not allowed in VNF command templates but that are valid in the *vnf* argument to the CONVERT function:

- Ordinal styles for the day of the month (DT and DTL)
- First-letter style for the month (MT)
- Styles for the day of the week (W, WT, WTXT, WTXTL, WTEXT, and WTEXTL)

Append a B code to any of these formats to indicate that you want to display the beginning day or month of the period, rather than the final day or month.

You can use any combination of VNF and DATEFORMAT format styles with for any dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR. This contrasts with the template in a VNF command, in which only certain format combinations are valid for each type of dimension.

In the following statement, CONVERT returns the current value of the MONTH dimension as a text value that is formatted by a *vnf* argument.

```
SHOW CONVERT(month TEXT '<MTEXTL>, <YYYY>')
```

In this example, the first MONTH value in status is DEC97, and the SHOW command presents it in the following format.

```
December, 1997
```

When you do not include the *vnf* argument, the format of the result is determined by the VNF of the dimension whose values you are converting. When the dimension has no VNF, the result is formatted according to the default VNF for the type of dimension being converted.

date-order
A text expression that specifies how to interpret the specified text value as a DATE value when the order of the text value's components (month, day, and year) is ambiguous. The expression can be one of the following: `'MDY'`, `'DMY'`, `'YMD'`, `'YDM'`, `'MYD'`, or `'DYM'`. Each letter represents a component of the date: `M` stands for month, `D` stands for day, and `Y` stands for year.

dwmqy-dimname
The name of a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR whose VNF or default date-order determines how to interpret the specified text value as a DATE value when the order of the text value's components is ambiguous.

The following examples show how you can control the conversion of a text expression by using a *date-order* or *dwmqy-dimname* argument.

The following statements use the *date-order* argument.

```
SHOW CONVERT('1/3/98' DATE 'MDY')
SHOW CONVERT('1/3/98' DATE 'DMY')
```

These statements produce the following output.

```
03JAN98
01MAR98
```

The following statement uses the *dwmqy-dimname* argument. It specifies the `qrtr` dimension, which was defined as a dimension of type QUARTER.

```
show convert('96-2' date qrtr)
```

The statement produces the following output.

```
31MAR96
```

The following statement also uses the *dwmqy-dimname* argument. It specifies the `fyear` dimension, which has the following definition.

```
DEFINE fyear DIMENSION YEAR ENDING JUNE
VNF 'TY<ff>'
```

This statement

```
SHOW CONVERT('jan97' DATE fyear)
```

produces the following output.

```
30JUN97
```

When you do not include the *date-order* or *dwmqy-dimname* argument, any ambiguity in the interpretation of a text expression is resolved by the current setting of the DATEORDER option. Refer to the DATEORDER option for a complete description of DATE values and how they are interpreted.

width-exp
An INTEGER expression that indicates the width of the output from CONVERT. The minimum width is 7. The default width is the current value of the LSIZE option. This argument is required when you specify the *escape-int* argument.

escape-int

Indicates whether escape sequences are to be used in the output. For this argument you can specify one of the values listed in Table 9–3, " Values for escape-int Argument of the CONVERT".

Table 9–3 Values for escape-int Argument of the CONVERT

Value	Description
-1	Do not use escapes. Precede -1 with a comma (`,-1`) so that Oracle OLAP does not subtract 1 from a preceding WIDTH argument.
0	Use escapes for unprintable characters. (Default)
1	Use escapes for all characters.

For more information on escape sequences in the OLAP DML, see "Escape Sequences" on page 2-4.

width-exp

An INTEGER expression that controls the width of the converted result. It can evaluate to `1`, `2`, or `4` bytes. The default width is `2` for BOOLEAN, or `4` for INTEGER. When an integer is too large to fit in the specified width, the result is `NA`. When the width is invalid or specified for some other data type, an error occurs.

NOXLATE

A keyword indicating that no character set conversion should be performed. Instead, Oracle OLAP only tags the converted value with the target data type, leaving the data as it was before the CONVERT function was called. Use this keyword only when it is necessary to store binary data in a TEXT or NTEXT variable.

Notes

INFILE Conversion

The maximum number of characters in a line is 4000. An error occurs when you try an INFILE conversion that produces a line with more than 4000 characters. This can occur when the source line exceeds 99 characters and enough of them need escape sequences.

Converting DATE Values

When you convert a DATE value to an INTEGER value, the result is the sequence number that represents the date (the sequence number `1` represents January `1, 1900`). When you convert a DATE value to another numeric type, the date's integer sequence number is converted to the specified numeric data type.

Converting INTEGER Values

When you convert an INTEGER value to a DATE value, the result is the date whose sequence number matches the specified integer (January `1, 1900` is represented by the sequence number 1). When you convert from another numeric type to a DATE value, the number is converted to an INTEGER, then the INTEGER is converted to a DATE value.

Converting DATE, DAY, WEEK, MONTH, QUARTER, or YEAR to ID

When you convert a value of a dimension of type DATE, DAY, WEEK, MONTH, QUARTER, or YEAR to an ID value, and the result is more than eight characters long, the result is truncated.

Converting Relation Values to INTEGER

When you convert a given value of a relation into an INTEGER value, the result represents the position of the value in the relation's dimension. This behavior reflects the fact that the values of a relation are dimension values, not TEXT values.

Converting Numbers to a DATE Value

When you convert a number to a DATE value and the result is outside the range of valid dates, CONVERT returns `NA`. Valid dates range from January 1, 1900 (sequence number 1) to December 31, 9999 (sequence number 2,958,464).

Converting to INTEGER or SHORTINTEGER

When you try to convert a number larger than 2,147,483,647 or smaller than -2,147,483,647 (the maximum and minimum integer values), to an INTEGER, you get a result of `NA`.

Likewise, when you try to convert a number larger than 32,767 or smaller than -32,768 to a SHORTINTEGER, you get a result of `NA`. For a value of type DATE, the integer 32,767 represents the date September 17, 1992. Therefore, CONVERT returns `NA` when you attempt to convert any date later than this to a SHORTINTEGER value.

Converting a Null String

When you convert a null string to a BYTE, you get a result of `32`. CONVERT returns the same value for a null string (`''`) as it does for a blank string (`'  '`).

PACKED and BINARY Conversion

The PACKED and BINARY types are useful for creating binary files that contain PACKED and BINARY data. To create such a file, use FILEOPEN statement with the BINARY keyword to open the file and FILEPUT to write values to it. You can use the ROW function as an argument to the FILEPUT command to help format the file.

Examples

Example 9–21 Converting Decimal Values to Text

This example shows how to use the JOINCHARS and CONVERT functions together to combine some text with the value of the variable `price` for a product and month, and show the price without decimal places.

```
LIMIT month TO 'Jul96'
LIMIT product to 'Canoes'
SHOW JOINCHARS('Price of Canoes = $' CONVERT(price TEXT 0))
Price of Canoes = $200
```

Example 9–22 Converting Text Values to Escape Sequences

This example shows how to use the CONVERT function with the ESCAPEBASE option to convert a TEXT value from its default decimal escape sequences to hexadecimal escape sequences.

```
DEFINE textvar VARIABLE TEXT
textvar = 'testvalue'
SHOW CONVERT(textvar INFILE 9 1)
'\d116\d101\d115\d116\d118\d097\d108\d117\d101'
ESCAPEBASE = 'x'
SHOW CONVERT(textvar INFILE 9 1)
'\x74\x65\x73\x74\x76\x61\x6C\x75\x65'
```

COPYDFN

The COPYDFN command defines a new object in the analytical workspace and uses the same definition as a specified object in the current workspace or in an attached workspace.

Syntax

COPYDFN *newobject oldobject*

Arguments

newobject
The name of the new object to define.

oldobject
The name of the object whose definition you want to copy.

Notes

Copying Objects

COPYDFN copies the DEFINE, LD, and PROPERTY lines for any type of object, and it copies the formula (EQ) of a formula object, and the value name format (VNF) of a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR. COPYDFN also copies the text of a program or model.

COPYDFN does not copy the PERMIT lines for any object, and it does not copy the compiled code of a formula, program, or model.

Examples

Example 9–23 Copying Programs

The following statements use COPYDFN to create a new program, called `newprog`, which is a copy of an existing one called `oldprog`. You could then edit `newprog` to create a slightly different program. The `oldprog` program has the following definition.

```
DEFINE oldprog PROGRAM
LD Shows total sales for the top five months from high to low
PROGRAM
LIMIT district TO 'BOSTON'
LIMIT month TO TOP 5 BASEDON TOTAL(sales, month)
REPORT TOTAL(sales, month)
END
```

The statements

```
COPYDFN newprog oldprog
DESCRIBE newprog
```

produce the following definition for `newprog`.

```
DEFINE newprog PROGRAM
LD Shows total sales for the top five months from high to low
PROGRAM
LIMIT district TO 'BOSTON'
LIMIT month TO TOP 5 BASEDON TOTAL(sales, month)
REPORT TOTAL(sales, month)
END
```

CORRELATION

The CORRELATION function returns the correlation coefficients for the pairs of data values in two expressions. A correlation coefficient indicates the strength of relationship between the data values. The closer the correlation coefficient is to positive or negative `1`, the stronger the relationship is between the data values in the expressions. A correlation coefficient of `0` (zero) means no correlation and a `+1` (plus one) or `-1` (minus one) means a perfect correlation. A positive correlation coefficient indicates that as the data values in one expression increase (or decrease), the data values in the other expression also increase (or decrease). A negative correlation coefficient indicates that as the data values in one expression increase, the data values in other expression decrease.

Return Value

DECIMAL

Syntax

CORRELATION(*expression1 expression2* [PEARSON|SPEARMAN|KENDALL] -

[BASEDON *dimension-list*])

Arguments

expression1
A dimensioned numeric expression with at least one dimension in common with *expression2*.

expression2
A dimensioned numeric expression with at least one dimension in common with *expression1*.

PEARSON
Calculates the Pearson product-moment correlation coefficient. Use this method when the data is interval-level or ratios, such as units sold and price for each unit, and the data values in the expressions have a linear relationship and are distributed normally.

SPEARMAN

Calculates Spearman's rho correlation coefficient. Use this nonparametric method when the expressions do not have a linear relationship or a normal distribution. In computing the correlation coefficient, this method ranks the data values in *expression1* and in *expression2* and then compares the rank of each element in *expression1* to the corresponding element in *expression2*. This method assumes that most of the values in the expressions are unique.

KENDALL

Calculates Kendall's tau correlation coefficient. This nonparametric method is similar to the SPEARMAN method in that it also first ranks the data values in *expression1* and in *expression2*. The KENDALL method, however, compares the ranks of each pair to the successive pairs. Use this method when few of the data values in *expression1* and in *expression2* are unique.

BASEDON *dimension-list*

An optional list of dimensions along which CORRELATION computes the correlation coefficient. Both *expression1* and *expression2* must be dimensioned by all of the *dimension-list* dimensions. CORRELATION correlates the data values of *expression1* to those of *expression2* along all of the *dimension-list* dimensions. CORRELATION returns an array that contains one correlation coefficient for each cell that is dimensioned by all of the dimensions of *expression1* and *expression2* except those in *dimension-list*.

When you do not specify a *dimension-list* argument, then CORRELATION computes the correlation coefficient over all of the common dimensions of *expression1* and *expression2.* When all of the dimensions of the two expressions are the same, then CORRELATION returns a single correlation coefficient. When either expression contains dimensions that are not shared by the other expression, then CORRELATION returns an array that contains one correlation coefficient for each cell that is dimensioned by the dimensions of the expressions that are not shared.

Notes

The Effect of NASKIP

CORRELATION is affected by the NASKIP option. When NASKIP is set to `YES` (the default), then CORRELATION ignores `NA` values. When NASKIP is set to `NO`, then an `NA` value in the expressions results in a correlation coefficient of `NA`.

Examples

Example 9–24 Correlating with the PEARSON Method

These examples use the `units` and `price` variables. The two dimensions of the `price` variable, `month` and `product`, are shared by the `units` variable, which has a third dimension, `district`.

The following CORRELATION statement does not specify a *dimension-list* argument. The output of the CORRELATION function in the command is one correlation coefficient for each of the dimension values in the dimension that the variables do not have in common.

```
REPORT CORRELATION(units price pearson)
```

The preceding statement produces the following output.

```
               CORRELATION
                 (UNITS
                  PRICE
DISTRICT        PEARSON)
-------------- -----------
Boston               -0.75
Atlanta              -0.85
Chicago              -0.83
Dallas               -0.66
Denver               -0.83
Seattle              -0.69
```

The following statements limit the `month` and `product` dimensions.

```
LIMIT month to 'Jan96' TO 'Mar96'
LIMIT product TO 'Tents' TO 'Racquets'
```

The following statement reports the correlation coefficient based on the `product` dimension for the limited dimension values that are in status.

```
REPORT CORRELATION(units price pearson basedon product)

               CORRELATION(UNITS PRICE PEARSON-
               --------BASEDON PRODUCT)--------
               -------------MONTH--------------
DISTRICT         Jan96      Feb96      Mar96
-------------- ---------- ---------- ----------
Boston              -0.96      -0.90      -0.89
Atlanta             -0.97      -0.97      -0.97
Chicago             -0.96      -0.95      -0.95
Dallas              -0.98      -0.98      -0.99
Denver              -0.97      -0.97      -0.97
Seattle             -0.89      -0.83      -0.83
```

The following statement reports the correlation coefficient based on the `month` dimension for the limited dimension values.

```
REPORT CORRELATION(units price pearson basedon month)

               CORRELATION(UNITS PRICE PEARSON-
               ---------BASEDON MONTH)---------
               ------------PRODUCT-------------
DISTRICT         Tents      Canoes    Racquets
-------------- ---------- ---------- ----------
Boston              -0.59      -0.92      -0.55
Atlanta             -0.73      -0.83       0.03
Chicago             -0.91      -0.84      -0.68
Dallas              -0.86      -0.92       0.31
Denver              -0.98      -0.94      -0.67
Seattle             -0.98      -0.89      -0.70
```

COS

The COS function calculates the cosine of an angle expression. The result returned by COS is a decimal value with the same dimensions as the specified expression.

Return Value

DECIMAL

Syntax

COS(*angle-expression*)

Arguments

angle-expression
A numeric expression that contains an angle value, which is specified in radians.

Examples

Example 9–25 Calculating the Cosine of an Angle in Radians

This example calculates the cosine of an angle of `1` radian. The statements

```
DECIMALS = 5
SHOW COS(1)
```

produce the following result.

```
0.54030
```

Example 9–26 Calculating the Cosine of an Angle in Degrees

This example calculates the cosine of an angle of 60 degrees. Since 1 degree = `2 * (pi) / 360` radians, 60 degrees is about `60 * 2 * 3.14159 / 360` radians. The statement

```
SHOW COS(60 * 2 * 3.14159 / 360)
```

produces the following result.

```
0.50000
```

COSH

The COSH function calculates the hyperbolic cosine of an angle expression.

Return Value

DECIMAL

Syntax

COSH(*expression*)

Arguments

expression
A numeric expression that contains an angle value, which is specified in radians.

Examples

Example 9–27 Calculating the Hyperbolic Cosine of an Angle

This example calculates the hyperbolic cosine of an angle of 1 radian. The statements

```
DECIMALS = 5
SHOW COSH(1)
```

produce the following result.

```
1.54030
```

COUNT

The COUNT function counts the number of `TRUE` values of a Boolean expression. It returns `0` (zero) when no values of the expression are `TRUE`.

See also: ANY, EVERY, and NONE.

Return Value

INTEGER

Syntax

COUNT(*boolean-expression* [[STATUS] *dimensions*])

Arguments

boolean-expression
The Boolean expression whose `TRUE` values are to be counted.

STATUS
Can be specified when one or more of the dimensions of the result of the function are not dimensions of the Boolean expression. (See the description of the *dimensions* argument.) When you specify the STATUS keyword when this is not the case, Oracle OLAP produces an error.

In cases where one or more of the dimensions of the result of the function are not dimensions of the Boolean expression, the STATUS keyword might be *required* in order for Oracle OLAP to process the function successfully, or the STATUS keyword might provide a performance enhancement. See "The STATUS Keyword" on page 9-71.

dimensions
The dimensions of the result. By default, COUNT returns a single count of all `TRUE` values. When you indicate one or more dimensions for the results, COUNT counts `TRUE` values along the dimensions that are specified and returns an array of values. Each dimension must be either a dimension of *boolean-expression* or related to one of its dimensions. When it is a related dimension, you can specify the name of the relation instead of the dimension name. This enables you to choose which relation is used when there is more than one.

Notes

The Effect of NASKIP

COUNT is affected by the NASKIP option. When NASKIP is set to YES (the default), COUNT returns the number of TRUE values of the Boolean expression, regardless of how many other values are NA. When NASKIP is set to NO, COUNT returns NA when any value of the expression is NA. When all the values of the expression are NA, COUNT returns NA for either setting of NASKIP.

Data with a Time Dimension

When *boolean-expression* is dimensioned by a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR, you can specify any other DAY, WEEK, MONTH, QUARTER, or YEAR dimension as a related *dimension.* Oracle OLAP uses the implicit relation between the dimensions. To control the mapping of one DAY, WEEK, MONTH, QUARTER, or YEAR dimension to another (for example, from weeks to months), you can define an explicit relation between the two dimensions and specify the name of the relation as the *dimension* argument to the COUNT function.

For each time period in the related dimension, Oracle OLAP counts the TRUE data values for all the source time periods that end in the target time period. This method is used regardless of which dimension has the more aggregate time periods.

The STATUS Keyword

When one or more of the dimensions of the result of the function are not dimensions of the Boolean expression, Oracle OLAP creates a temporary variable to use while processing the function. When you specify the STATUS keyword, Oracle OLAP uses the current status instead of the default status of the related dimensions for calculating the size of this temporary variable.

When the size of the temporary variable for the results of the function would exceed 2 gigabytes, you *must* specify the STATUS keyword in order for Oracle OLAP to execute the function successfully. When the dimensions of the Boolean expression are limited to a few values and are physically fragmented, you can specify the STATUS keyword to improve the performance of the function.

When you use the COUNT function with the STATUS keyword in an expression that requires going outside of the status for results (for example, with the LEAD or LAG functions or with a qualified data reference), the results outside of the status will be returned as NA.

Examples

Example 9–28 Counting True Values by District

You can use COUNT to find the number of months in which each district sold more than 2,000 units of sportswear. To obtain a count for each district, specify `district` as the dimension for the result.

```
LIMIT product TO 'SPORTSWEAR'
REPORT HEADING 'Count' COUNT(units GT 2000, district)
```

The preceding statement statements produce the following output.

```
DISTRICT         Count
-------------- ----------
Boston                  0
Atlanta                23
Chicago                11
Dallas                 24
Denver                  7
Seattle                 0
```

CUMSUM

The CUMSUM function computes cumulative totals over time or over another dimension. When the data being totaled is one-dimensional, CUMSUM produces a single series of totals, one for all values of the dimension. When the data has dimensions other than the one being totaled over, CUMSUM produces a separate series of totals for each combination of values in the status of the other dimensions.

By default, CUMSUM ignores the current status of the dimension over which it is calculating totals. You can override this behavior by specifying the INSTAT keyword.

Return Value

DECIMAL

Syntax

CUMSUM(*cum-expression* [STATUS] *total-dim* [*reset-dim*] [INSTAT])

Arguments

cum-expression
A numeric variable or calculation whose values you want to total, for example UNITS.

STATUS
May be specified to improve the performance of CUMSUM when *cum-expression* has more than one dimension. When you specify the STATUS keyword when the data being totaled is one-dimensional, an error results. For more information, see "Using the STATUS Keyword" on page 9-75.

total-dim
The dimension of *cum-expression* over which you want to total.

reset-dim

Specifies that the cumulative totals in a series should start over with each new reset dimension value, for example at the start of each new year. The reset dimension can be any of the following:

- Any dimension related to *total-dim* through an explicitly defined relation.
- Any dimension with a type of DAY, WEEK, MONTH, QUARTER, or YEAR, when *total-dim* also has a type of DAY, WEEK, MONTH, QUARTER, or YEAR. CUMSUM uses the implicit relation between the two dimensions, so they do not need to be related through an explicit relation. See "Overriding an Implicit Relation" on page 9-74.
- A relation dimensioned by *total-dim.* CUMSUM uses the related dimension as the reset dimension. This enables you to choose which relation is used when there is more than one.

INSTAT

May be specified to cause CUMSUM to use only the values of *total-dim* that are currently in status. When you do not specify INSTAT, CUMSUM produces a total for *all* the values of *total-dim*, independent of its current status. See "Current Status Ignored" on page 9-74.

Notes

Overriding an Implicit Relation

When you specify dimensions with a type of DAY, WEEK, MONTH, QUARTER, or YEAR for both the *total-dim* argument and the *reset-dim* argument, CUMSUM uses the implicit relation between the two dimensions even when an explicit relation exists. However, you can override the default and use the explicit relation by specifying the name of the relation for the *reset-dim* argument.

Current Status Ignored

Unless you specify the INSTAT keyword, CUMSUM ignores the current status in calculating totals. Suppose MONTH is the dimension being totaled over (and INSTAT has not been specified). The CUMSUM total for a given month uses the values for all preceding months, even when some are not in the status. When a reset dimension is specified, the total for a given month uses the values for all preceding months that correspond to the same value of the reset dimension (for example, all preceding months in the same year). To calculate year-to-date totals, specify YEAR as the reset dimension.

NASKIP Option

CUMSUM is affected by the NASKIP option. When NASKIP is set to `YES` (the default), CUMSUM ignores `NA` values and returns a cumulative total using the available values. When NASKIP is set to `NO`, CUMSUM returns `NA` when any data value has a value of `NA`. When all the values are `NA`, CUMSUM returns `NA` for either setting of NASKIP.

Using the STATUS Keyword

When *cum-expression* is multidimensional, CUMSUM creates a temporary variable to use while processing the function. When you specify the STATUS keyword, CUMSUM uses the current status instead of the default status of the dimensions for calculating the size of this temporary variable. When the dimensions of the expression are limited to a few values and are physically fragmented, you can improve the performance of CUMSUM by specifying STATUS.

When you use CUMSUM with the STATUS keyword in an expression that requires going outside of status for results (for example, with the LEAD or LAG functions or with a qualified data reference), the results outside of status will be returned as `NA`.

Examples

Example 9–29 Multiple CUMSUM Calculations

This example shows cumulative `units` totals for tents and canoes in the Atlanta district for the first six months of 1996. The report shows the units figures themselves, year-to-date totals calculated using `year` as the reset dimension, and totals calculated with no reset dimension using all preceding months. Assume that you issue the following statements.

```
LIMIT district TO 'Atlanta'
LIMIT product TO 'Tents' 'Canoes'
LIMIT month TO 'Jan96' TO 'Jun96'
REPORT DOWN month units CUMSUM(units, month year) -
   CUMSUM(units, month)
```

The following report is displayed.

```
DISTRICT: ATLANTA
         ------------------------PRODUCT------------------------
         ---------TENTS------------- ---------CANOES------------
               CUMSUM(UNI                   CUMSUM(UNI
                TS, MONTH CUMSUM(UNI         TS, MONTH CUMSUM(UNI
MONTH    UNITS    YEAR)   TS, MONTH)  UNITS    YEAR)   TS, MONTH)
-----   -------- --------- ---------- ------- --------- ----------
Jan96      279       279      5,999      281       281      5,162
Feb96      305       584      6,304      309       590      5,471
Mar96      356       940      6,660      386       976      5,857
Apr96      537     1,477      7,197      546     1,522      6,403
May96      646     2,123      7,843      525     2,047      6,928
Jun96      760     2,883      8,603      608     2,655      7,536
```

The totals for CUMSUM(UNITS, MONTH) include values for all months beginning with the first month, JAN95. The totals for CUMSUM(UNITS, MONTH YEAR) include only the values starting with JAN96.

Example 9–30 Resetting for a Quarter

This example shows cumulative totals for the same products and district, for the entire year 1996. Because `quarter` is specified as the reset dimension, totals start accumulating at the beginning of each quarter. The cumulative totals for `Jan96`, `Apr96`, `Jul96`, and `Oct96` are the same as the units figures for those months. Assume that you issue the following statements.

```
LIMIT district TO 'Atlanta'
LIMIT product TO 'Tents' 'Canoes'
limit month TO year 'Yr96'
REPORT DOWN month units CUMSUM(units, month quarter)
```

A report displays.

```
DISTRICT: ATLANTA
                 ------------------PRODUCT------------------
                 --------TENTS-------- -------CANOES--------
                            CUMSUM(UNI           CUMSUM(UNI
                             TS, MONTH            TS, MONTH
MONTH                 UNITS     QUARTER)    UNITS     QUARTER)
------------ ---------- ---------- ---------- ----------
Jan96               279        279        281        281
Feb96               305        584        309        590
Mar96               356        940        386        976
Apr96               537        537        546        546
May96               646      1,183        525      1,071
Jun96               760      1,943        608      1,679
Jul96               852        852        626        626
Aug96               730      1,582        528      1,154
Sep96               620      2,202        520      1,674
Oct96               554        554        339        339
Nov96               380        934        309        648
Dec96               284      1,218        288        936
```

DATEFORMAT

The DATEFORMAT option holds the template used for displaying DATE values and converting DATE values to TEXT values. The template can include format specifications for any of the four components of a date (day, month, year, and day of the week). It can also include additional text.

See also: MONTHNAMES option, DAYNAMES option, DATEORDER option.

Data type

TEXT

Syntax

DATEFORMAT = *template*

Arguments

template
A TEXT expression that specifies the template for displaying dates. Each component in the template must be preceded by a left angle bracket and followed by a right angle bracket. You can include additional text before, after, or between the components. The default template is `'<DD><MTXT><YY>'`.

Table 9–4, " DATEFORMAT Templates for Day", Table 9–5, " DATEFORMAT Templates for Week", Table 9–6, " DATEFORMAT Templates for Month", and Table 9–7, " DATEFORMAT Templates for Year" present the valid formats for each component. The tables provide two display examples, one for March 1, 1990 and another for November 12, 2051.

Table 9–4 DATEFORMAT Templates for Day

Format	Meaning	March 1, 1990	November 12, 2051
`<D>`	One digit or two digits	`1`	`12`
`<DD>`	Two digits	`01`	`12`
`<DS>`	Space-padded, two digits	`1`	`12`
`<DT>`	Ordinal, uppercase	`1ST`	`12TH`
`<DTL>`	Ordinal, lowercase	`1st`	`12th`

Table 9–5, " DATEFORMAT Templates for Week" presents the valid formats for weeks. The table provides two display examples, one for March 1, 1990 and another for November 12, 2051.

Table 9–5 DATEFORMAT Templates for Week

Format	Meaning	March 1, 1990	November 12, 2051
`<W>`	Numeric	`4`	`1`
`<WT>`	First letter, uppercase	`W`	`S`
`<WTXT>`	First three letters, uppercase.	`WED`	`SUN`
`<WTXTL>`	First three letters, lowercase	`Wed`	`Sun`
`<WTEXT>`	Full name, uppercase	`WEDNESDAY`	`SUNDAY`
`<WTEXTL>`	Full name, lowercase	`Wednesday`	`Sunday`

Note that when you specify a format of `<WTXT>`, `<WTXTL>`, `<WTEXT>`, or `<WTEXTL>`, the case in which the value is specified in DAYNAMES effects the displayed value:

- When the name in DAYNAMES is entered as all lowercase, the entire name is converted to uppercase. Otherwise, the first letter is converted to uppercase and the second and subsequent letters remain in their original case.
- When the name in DAYNAMES is entered as all uppercase, the second and subsequent letters are converted to lowercase. Otherwise, the entire name remains in the case specified in DAYNAMES.

Table 9–6, " DATEFORMAT Templates for Month" presents the valid formats for months. The table provides two display examples, one for March 1, 1990 and another for November 12, 2051.

Table 9–6 DATEFORMAT Templates for Month

Format	Meaning	March 1, 1990	November 12, 2051
`<M>`	One digit or two digits	1	`11`
`<MM>`	Two digits	`03`	`11`
`<MS>`	Space-padded, two digits	`3`	`11`
`<MT>`	First letter, uppercase	`M`	`N`
`<MTXT>`	First three letters, uppercase	`MAR`	`NOV`
`<MTXTL>`	First three letters, lowercase	`Mar`	`Nov`

Note that when you specify a format of `<MTXT>` or `<MTXTL>`, the case in which the value is specified in MONTHNAMES effects the displayed value:

- When the name in MONTHNAMES is entered as all lowercase, the entire name is converted to uppercase. Otherwise, the first letter is converted to uppercase and the second and subsequent letters remain in their original case.
- When the name in MONTHNAMES is entered as all uppercase, the second and subsequent letters are converted to lowercase. Otherwise, the entire name remains in the case specified in MONTHNAMES.

Table 9–7, " DATEFORMAT Templates for Year" presents the valid formats for years. The table provides two display examples, one for March 1, 1990 and another for November 12, 2051.

Table 9–7 DATEFORMAT Templates for Year

Format	Meaning	March 1, 1990	November 12, 2051
`<YY>`	Two digits or four digits	`90`	`2051`
`<YYYY>`	Four digits	`1990`	`2051`

Notes

Angle Brackets

To include an angle bracket as additional text in a template, specify two angle brackets for each angle bracket to be included as text (for example, to display the entire date in angle brackets, specify `'<<<D><M><YY>>>'`).

Month and Day Names

The names used in the month component for the MT, MTXT, MTXTL, MTEXT, and MTEXTL formats are drawn from the current setting of the MONTHNAMES option. The names used in the day-of-the-week component for the WT, WTXT, WTXTL, WTEXT, and WTEXTL formats are drawn from the current setting of the DAYNAMES option.

Abbreviations

You can set the DAYABBRLEN and MONTHABBRLEN options to use abbreviations of different lengths for day and month names.

Out-of-Range Years

When you specify the YY format, and a year outside the range of 1950 to 2049 is to be displayed, the year is displayed in four digits.

DATE-to-TEXT Conversion

When you use a DATE value where a text value (TEXT or ID) is expected, or when you store a DATE value in a TEXT variable, DATEFORMAT automatically converts the DATE value to a TEXT value. The current template in the DATEFORMAT option is used to format the text.

When you want to override the current DATEFORMAT template, you can convert the DATE value to TEXT by using the CONVERT function with a date-format argument. See the CONVERT function for an example.

Once a DATE value is stored in a TEXT variable, the DATEFORMAT template is no longer used to format the display of the value, and subsequent changes to DATEFORMAT have no impact.

Time Dimension Values

The DATEFORMAT option does not affect the way values of dimensions of type DAY, WEEK, MONTH, QUARTER, or YEAR are displayed. The display of DAY, WEEK, MONTH, QUARTER, and YEAR dimension values is controlled by a VNF (value name format) attached to the dimension definition, or by default conventions for DAY, WEEK, MONTH, QUARTER, and YEAR dimensions.

Examples

Example 9–31 Changing the Format of Dates

Example: The following statements define a DATE variable and set its value to March 24, 1997, then set the date format to two digits each in the order of day, month, and year, and send the result to the current outfile.

```
DEFINE datevar VARIABLE DATE
datevar = '24Mar97'
DATEFORMAT = '<DD>/<MM>/<YY>'
SHOW datevar
```

These statements produce the following output.

```
24/03/97
```

Example: The following statements change the date format to month (text), day (two digits), and year (four digits), and send the result to the current outfile.

```
DATEFORMAT = '<MTEXTL> <D>, <YYYY>'
SHOW DATEVAR
```

These statements produce the following output.

```
March 24, 1997
```

Example: The following commands change the date format to day of the week (text), month (text), day (one or two digits), and year (four digits), and send the result to the current outfile.

```
DATEFORMAT = '<WTEXTL> <MTEXTL> <D>, <YYYY>'
SHOW DATEVAR
```

These commands produce the following output.

```
Monday March 24, 1997
```

Example 9–32 Including Text in the Format of a Date

The following statements save and then change the DATEFORMAT option to include extra text for a workspace startup greeting.

```
PUSH DATEFORMAT
DATEFORMAT = 'Hello.  Today is <wtextl>, the <dtl> -
OF <MTEXTL>.'
SHOW TODAY
POP DATEFORMAT
```

When today's date is May 30, 1997, the following output is sent to the current outfile when the program is run.

```
Hello.  Today is Friday, the 30th of May.
```

DATEORDER

The DATEORDER option holds three characters that indicate the intended order of the month, day, and year components of the DATE values in a workspace for those cases in which their interpretation is ambiguous. Oracle OLAP automatically refers to DATEORDER whenever you enter an ambiguous DATE value or convert one from a text value. For information about date values, see notes.

Data type

ID

Syntax

DATEORDER = *order*

Arguments

order
One of the following text expressions: `'MDY'`, `'DMY'`, `'YMD'`, `'YDM'`, `'MYD'`, `'DYM'`. Each letter represents a component of the date. `M` stands for the month, `D` for the day, and Y for the year. The default date order is `'MDY'`.

Notes

Date Values
A valid DATE value must fall between January 1, 1900, and December 31, 9999. It must conform to one of the following three styles, which you can mix throughout a session:

Numeric style -- Specify the day, month, and year as three integers with one or more separators between them, using these rules:

- The day and month components can have one digit or two digits.
- For any year, the year component can have four digits (for example, 1997). For years in the range 1950 to 2049, the year component can, alternatively, have two digits (50 represents 1950, and so on).

- To separate the components, you can use a space (), dash (-), slash (/), colon (:), or comma (,).
- Examples: `'24/4/97'` or `'24-04-1997'`

Packed numeric style -- Specify the day, month, and year as three integers with no separators between them, using these rules:

- The day and month components must have two digits. When the day or month is less than 10, it must be preceded by a zero.
- For any year, the year component can have four digits (for example, 1997). For years in the range 1950 to 2049, the year component can, alternatively, have two digits (50 represents 1950, and so on).
- You cannot use any separators between the date components.
- Examples: `'240497'` or `'04241997'`

Month name style -- Specify the day and year as integers and the month as text, using these rules:

- The month component must match one of the names listed in the MONTHNAMES option. You can abbreviate the month name to one letter or more, when you supply enough letters to uniquely match the beginning of a name in MONTHNAMES. The case of the letters in the month component (uppercase or lowercase) does not need to match the case in MONTHNAMES.
- The day component can have one digit or two digits.
- For any year, the year component can have four digits (for example, 1997). For years in the range 1950 to 2049, the year component can, alternatively, have two digits (50 represents 1950, and so on).
- When the day and year components are adjacent, they must have at least one separator between them. As separators, you can use a space (), dash (-), slash (/), colon (:), or comma (,). When you want, you can place one or more separators between the day and month or between the year and month.
- Examples: `'24APR97'` or `'24 ap 97'` or `'April 24, 1997'`

Valid Dates

To determine whether a text expression (such as an expression with a data type of TEXT or ID) represents a valid DATE value, use the ISDATE program.

Ambiguous Dates

When you enter an unambiguous DATE value or convert a text value that has only one interpretation as a date, it is handled without consulting the DATEORDER option. For example, in `03-24-97` the `97` can only refer to the year. Considering what is left, the `24` cannot refer to the month, so it must be the day. Only `03` is left, so it must be the month. When, however, the interpretation is ambiguous, as in the value `3-5-97`, the current value of DATEORDER is used to interpret the meaning of each component.

TEXT-to-DATE Conversion

When you use a text value where a DATE value is expected, or when you store a text value in a DATE variable, the text value must conform to one of the styles listed earlier in this entry. Oracle OLAP automatically converts the text value to a DATE value. When the meaning of the text value is ambiguous, the current setting of DATEORDER is used to interpret the value.

To override the current DATEORDER setting in converting a text value to a DATE value, use the CONVERT function with the date-order argument.

Essential Date Components

Suppose you want to assign a date value to a DAY, WEEK, MONTH, QUARTER, or YEAR dimension using a MAINTAIN command or to a valueset using the LIMIT command. When you specify the value in the form of a DATE expression or a text literal, Oracle OLAP uses the DATEORDER option to interpret the value. When supplying a text literal, you can use any valid input style for dates. However, you need to supply only the date components that are necessary for identifying a time period in the particular type of dimension or valueset you are using. For example, for a MONTH dimension or its valueset, you can specify a complete date, such as `30jun97`, or you can provide only the essential components, such as `jun97` or `0697`.

Time Dimension Phases

The DATEORDER option is used to interpret the *phase* argument to the DEFINE DIMENSION command for DAY, WEEK, MONTH, QUARTER, and YEAR dimensions.

Examples

Example 9–33 Changing the Date Order

The following commands define and assign a value to a DATE variable, specify the date format and the date order, and send the output to the current outfile.

```
DEFINE datevar VARIABLE DATE
dATEFORMAT = '<MTXT> <D>, <YYYY>'
DATEORDER = 'MDY'
DATEVAR = '3 5 1997'
SHOW DATEVAR
```

These commands produce the following output.

```
MAR 5, 1997
```

The following commands change the date order, and, therefore, the way the same value of the DATE variable is interpreted.

```
DATEORDER = 'DMY'
SHOW DATEVAR
```

These commands produce the following output.

```
MAY 3, 1997
```

DAYABBRLEN

The DAYABBRLEN option specifies the number of characters to use for abbreviations of day names that are stored in the DAYNAMES option. You can specify how many characters to use for abbreviating particular day names when you specify the `<WT>`, `<WTXT>`, and `<WTXTL>` formats with the DATEFORMAT text option.

Data type

TEXT

Syntax

DAYABBRLEN = *specification* [;|, *specification*]...

where:

specification
Is a text expression that has the following form:

startpos [- *endpos*] : *length*

Arguments

***startpos* [- *endpos*]**
Numbers that represent the first and last days whose abbreviation length is defined by *length.* These numerical positions apply to the corresponding lines of text in the DAYNAMES option. You can specify these ranges of values in reverse order, *endpos* [-*startpos*], when you prefer.

The DAYNAMES option can have more than seven lines, so you can specify *startpos* and *endpos* greater than seven in the setting of DAYABBRLEN. When you specify a range where neither *startpos* nor *endpos* has a corresponding text value in the DAYNAMES option, then Oracle OLAP has no text values to abbreviate for that range. When you later change your day names list so that *startpos* is valid, the specified abbreviation is applied.

length
A number that specifies the length in characters (not bytes) of abbreviated day names.

Notes

Abbreviation Lengths

You can define many different groups of days, each with different abbreviation lengths. When you do so, separate the groups with a comma or a semicolon as shown in the syntax.

Default Abbreviations

When you do not specify an abbreviation length for a given position in the DAYNAMES option, or when you explicitly set a given position to zero, Oracle OLAP uses the default abbreviation. The default abbreviations are one character for `<WT>` and three characters for `<WTXT>` and `<WTXTL>`. Oracle OLAP never uses abbreviations when you have designated the full name specifications `<WTEXT>` and `<WTEXTL>`.

Ambiguous Day Names

You can use DAYABBRLEN to interpret ambiguous names, for example, whether `'T'` stands for Tuesday or Thursday. When the DAYABBRLEN for Tuesday was `1` and for Thursday was 2, then `'T'` would always match Tuesday, and it would require at least `'Th'` to match Thursday. This does not depend on the order of Tuesday and Thursday in the week; it would work the same way when the two days were reversed. If, on the other hand, the DAYABBRLEN for each of these was `2`, then `'T'` would not match either one, and you would have to enter at least `'Tu'` or `'Th'` to get a match.

Examples

Example 9–34 Specifying Day Abbreviations

The following DAYABBRLEN setting specifies that the first five days of the week are abbreviated with one character and the last two days are abbreviated with two characters.

```
DAYABBRLEN = '1-5:1, 6-7:2'
DATEFORMAT = '<WTXT> <MTXT> <D>, <YYYY>'
SHOW CONVERT ('2 august 2005' DATE)
```

These commands product the following result, with Tuesday abbreviated to one character:

```
T AUGUST 2, 2005
```

DAYNAMES

The DAYNAMES option holds the list of valid names for the days of the week. The names are used to display values of type DATE or to convert DATE values to text.

Data type

TEXT

Syntax

DAYNAMES = *name-list*

Arguments

name-list

A multiline text expression that lists the names of the seven days of the week. Each name occupies a separate line. Regardless of which day you are treating as the first day of the week, the list must begin with the name for Sunday. The default value is the list of English names for the days of the week, in uppercase.

Notes

Extra Sets of Names

You can include more than one set of seven names in your list. The eighth name is a synonym for the first name, the ninth name is a synonym for the second name, and so on.

How DAYNAMES Is Used

The DAYNAMES list is consulted when you display or convert a date using the `<WT>`, `<WTXT>`, `<WTXTL>`, `<WTEXT>`, or `<WTEXTL>` formats. These formats are specified in the DATEFORMAT option. When you have more than one set of day names, Oracle OLAP chooses the synonym whose number of characters and capitalization pattern best match the DATEFORMAT specification.

Abbreviations

You can set the DAYABBRLEN option to control the number of characters used for abbreviations of day names.

Examples

Example 9–35 Specifying Day Names

The following commands set DAYNAMES to the French names for the days of the week and send the output to the current outfile.

```
DAYNAMES = 'dimanche\nlundi\n-
mardi\nmercredi\njeudi\nvendredi\nsamedi'
SHOW DAYNAMES
```

These commands produce the following output.

```
dimanche
lundi
mardi
mercredi
jeudi
vendredi
samedi
```

DAYOF

The DAYOF function returns an integer in the range of 1 through 7, giving the day of the week on which a specified date falls. A result of `1` refers to Sunday. The result has the same dimensions as the specified DATE expression.

Return Value

INTEGER

Syntax

DAYOF(*date-expression*)

Arguments

date-expression
An expression that has the DATE data type, or a text expression that specifies a date. See "TEXT-to-DATE Conversion" on page 9-93.

Notes

TEXT-to-DATE Conversion

In place of a DATE expression, you can specify a text expression that has values that conform to a valid input style for dates. DAYOF automatically converts the values of the text expression to DATE values, using the current setting of the DATEORDER option to resolve any ambiguity.

Examples

Example 9–36 Finding Today's Weekday

The following command sends the day of the week on which today's date falls to the current outfile.

```
SHOW DAYOF(TODAY)
```

When today's date is January 15, 1997, which is a Wednesday, this command produces the following output.

```
4
```

Example 9–37 Finding the Weekday of a Date

The following command sends the day of the week on which July 4 fell in 1996 to the current outfile.

```
SHOW DAYOF('04jul96')
```

This command produces the following output.

```
5
```

DBGOUTFILE

The DBGOUTFILE command sends debugging information to a file. When you set PRGTRACE and MODTRACE to `YES`, the file produced by DBGOUTFILE interweaves each line of your program, model, or infile with its corresponding output. When you set ECHOPROMPT to `YES`, the debugging file also includes error messages. The abbreviation for DBGOUTFILE is DOTF.

Syntax

DBGOUTFILE {EOF|[APPEND] *file-id* [NOCACHE]}

Arguments

EOF
Closes the current debugging file, and debugging output is no longer sent to a file.

file-id
Specifies the file identifier of the file to receive the debugging output.

APPEND
Specifies that the output should be added to the end of an existing file. When you omit this argument and a file exists with the specified name, the new output replaces the current contents of the file.

NOCACHE
Specifies that Oracle OLAP should write to the debugging file each time a line is executed. Without this keyword, Oracle OLAP reduces file I/O activity by saving text and writing it periodically to the file.

The NOCACHE keyword slows performance significantly, but it ensures that the debugging file records every line as soon as it is executed. When you are debugging a program that aborts after a certain line, NOCACHE ensures that you see every line that was executed.

Notes

PRGTRACE and MODTRACE Options

By setting PRGTRACE to `YES`, you specify that you want every line of a program written to the debugging file. Setting MODTRACE to `YES` has the same effect for models. For more information about these options, see their entries.

ECHOPROMPT Option

Setting ECHOPROMPT to `YES` specifies that you want not only input lines, but also error messages, to appear in the debugging file. For more information about these options, see their entries.

Examples

Example 9–38 Debugging with a Debugging File

The following commands create a useful debugging file called `debug.txt` in the current directory object.

```
PRGTRACE = yes
ECHOPROMPT = yes
DBGOUTFILE 'debug.txt'
```

After executing these commands, you can run your program as usual. To close the debugging file, execute this command.

```
DBGOUTFILE EOF
```

In the following sample program, the first LIMIT command has a syntax error.

```
DEFINE ERROR_TRAP PROGRAM
PROGRAM
TRAP ON traplabel
LIMIT month TO FIRST badarg
LIMIT product TO FIRST 3
LIMIT district TO FIRST 3
REPORT sales
traplabel:
SIGNAL ERRORNAME ERRORTEXT
END
```

With PRGTRACE and ECHOPROMPT both set to YES and with DBGOUTFILE set to send debugging output to a file called debug.txt, the following text is sent to the debug.txt file when you execute the error_trap program.

```
(PRG= ERROR_TRAP)
(PRG= ERROR_TRAP) TRAP ON traplabel
(PRG= ERROR_TRAP)
(PRG: ERROR_TRAP) LIMIT month TO FIRST badarg
ERROR: BADARG does not exist in any attached database.
(PRG= ERROR_TRAP) traplabel:
(PRG= ERROR_TRAP) SIGNAL ERRORNAME ERRORTEXT
ERROR: BADARG does not exist in any attached database.
```

Example 9–39 Sending Debugging Information to a File

The following is the text of a program whose first LIMIT command has a syntax error.

```
DEFINE error_trap PROGRAM
PROGRAM
TRAP ON traplabel
LIMIT month TO FIRST BADARG
LIMIT product TO FIRST 3
LIMIT district TO FIRST 3
REPORT sales
traplabel:
SIGNAL ERRORNAME ERRORTEXT
END
```

The following command sends debugging information to a file named debug.txt.

```
DBGOUTFILE 'debug.txt'
```

With PRGTRACE and ECHOPROMPT both set to YES, Oracle OLAP sends the following text to the debug.txt file when you execute the ERROR_TRAP program. The last line in the file is the command to stop recording the debugging information.

```
error_trap
(PRG= ERROR_TRAP)
(PRG= ERROR_TRAP) trap on traplabel
(PRG= ERROR_TRAP)
(PRG: ERROR_TRAP) limit month to first badarg
ERROR: BADARG does not exist in any attached workspace.
(PRG= ERROR_TRAP) traplabel:
(PRG= ERROR_TRAP) signal errorname errortext
ERROR: BADARG does not exist in any attached workspace.
dbgoutfile eof
```

DDOF

The DDOF function returns an integer in the range of 1 through 31, giving the day of the month on which a specified date falls. The result returned by DDOF has the same dimensions as the specified DATE expression.

Return Value

INTEGER

Syntax

DDOF(*date-expression*)

Arguments

date-expression
An expression that has the DATE data type, or a text expression that specifies a date. See "TEXT-to-DATE Conversion" on page 9-99.

Notes

TEXT-to-DATE Conversion

In place of a DATE expression, you can specify a text expression that has values that conform to a valid input style for dates. The values of the text expressions are automatically converted to DATE values using the current setting of the DATEORDER option to resolve any ambiguity.

Examples

Example 9–40 Finding Today's Day of the Month

The following command returns the day of the month on which today's date falls.

```
SHOW DDOF(TODAY)
```

When today's date is September 8, 2000, this command produces the following output.

```
8
```

10 DECIMALCHAR to DELETE

This chapter contains the following OLAP DML statements:

DECIMALCHAR

(Read-only) The DECIMALCHAR option is the value specified for the NLS_NUMERIC_CHARACTERS option.

Data type

ID

Syntax

DECIMALCHAR

Notes

Format for Decimal Input

DECIMALCHAR only affects the way Oracle OLAP formats numbers in *output*. When you format numbers for input, use a period (.) for the decimal marker. To use a different decimal marker, enclose the value in single quotes and use the TO_NUMBER function to convert the value from text to a valid number.

The Thousands Marker

The THOUSANDSCHAR option lets you check the value of the thousands marker.

Examples

Example 10–1 Identifying the Decimal and Thousands Markers

The statements in this example show the DECIMALCHAR and THOUSANDSCHAR values.

- The following statement might produce a comma as output.

  ```
  SHOW THOUSANDSCHAR
  ```

- The following statement might produce a period as output.

  ```
  SHOW DECIMALCHAR
  ```

- With these values, the following statement might produce the output that follows it it.

  ```
  SHOW TOTAL(sales)
  63,181,743.50
  ```

DECIMALOVERFLOW

The DECIMALOVERFLOW option controls the result of arithmetic operations that produce out-of-range numbers. Decimal numbers are stored as a mantissa and an exponent. Decimal overflow occurs when the result of a calculation is very large and can no longer be represented by the exponent portion of the decimal representation.

Data type

BOOLEAN

Syntax

DECIMALOVERFLOW = YES|NO

Arguments

YES
Allows overflow. A calculation that generates overflow will execute without error, and the results of the calculation will be `NA`.

NO
Disallows overflow. A calculation involving overflow will stop executing, and an error message will be produced. (Default)

Examples

Example 10–2 The Effect of DECIMALOVERFLOW

This example shows the effect of changing the value of the DECIMALOVERFLOW option.

When you execute a SHOW command such as the following without changing DECIMALOVERFLOW from its default value of `NO`, an error occurs.

```
SHOW 1000000.0 ** 133
```

When you change DECIMALOVERFLOW to `YES`, the same statement executes without an error and produces `NA` as the result of the operation. The statements

```
DECIMALOVERFLOW = YES
SHOW 1000000.0 ** 133
```

produce the following result.

```
NA
```

DECIMALS

The DECIMALS option controls the number of decimal places that are shown in numeric output. Values are rounded to fit the specified number of decimal places.

Data type

INTEGER

Syntax

DECIMALS = *n*

Arguments

n

An integer expression that specifies the number of decimal places to include in all output of DECIMAL and SHORTDECIMAL values; *n* can be any number in the range 0 to 40 or the number 255. A value of `255` produces "best presentation" formats. (See "Best Presentation Formats" on page 10-7.) The default is 2.

Notes

Showing INTEGER Data

The setting of DECIMALS does not affect the format of INTEGER values in output. INTEGER values are shown with no decimal places, unless you explicitly apply a DECIMAL attribute to them in a HEADING, REPORT, or ROW command.

Best Presentation Formats

When you set DECIMALS to `255`, you are specifying the following formats for numbers:

- SHORTDECIMAL values are shown with as many decimal places as can fit into an overall total of 7 significant digits.
- DECIMAL values are shown with as many decimal places as can fit into 15 significant digits.

Both DECIMAL and SHORTDECIMAL values are rounded to the last decimal place that is shown, and trailing decimal zeros are dropped. See "Comparing 2 Decimal Places with Best Presentation Format" on page 10-7.

Examples

Example 10–3   Showing Data with No Decimal Places

To show no decimal places in numeric output, set the DECIMALS option to `0` (zero) before you produce your report.

```
DECIMALS = 0
LIMIT line TO 'COGS'
LIMIT month TO 'Jan96' 'Feb96'
REPORT DOWN division ACROSS month: budget
```

These statements produce the following output.

```
LINE: COGS
               -------BUDGET--------
               --------MONTH--------
DIVISION         Jan96      Feb96
-------------- ---------- ----------
Camping           355,933    385,308
Sporting          279,773    323,982
Clothing          528,370    546,468
```

Example 10–4   Comparing 2 Decimal Places with Best Presentation Format

This example contrasts the effects of setting DECIMALS to `2` and setting it to `255` ("best presentation" format).

The OLAP DML statements

```
DECIMALS = 2
SHOW JOINCHARS(1.1 'A')
```

produce the following output.

```
1.10A
```

The OLAP DML statements

```
DECIMALS = 255
SHOW JOINCHARS(1.1 'A')
```

produce the following output.

```
1.1A
```

DECODE

The DECODE function compares one expression to one or more other expressions and, when the base expression is equal to a search expression, returns the corresponding result expression; or, when no match is found, returns the default expression when it is specified, or `NA` when it is not.

Return Value

The data type of the first *search* argument.

Syntax

DECODE (*expr* , *search*, *result* [, *search* , *result*]... [, *default*])

Arguments

expr
The expression to be searched.

search
An expression to search for.

result
The expression to return when *expression* is equal to *search*.

default
An expression to return when *expression* is not equal to *search*.

Notes

Order of Value Evaluation

The *search*, *result*, and *default* values can be derived from expressions. The function evaluates each *search* value only before comparing it to *expr*, rather than evaluating all *search* values before comparing any of them with *expr*. Consequently, the function never evaluates a *search* when a previous *search* is equal to *expr*.

Automatic Data Type Conversion Before Searching for Values

The function automatically converts *expr* and each *search* value to the data type of the first *search* value before comparing. The function automatically converts the return value to the same data type as the first *result*.

Decoding NA Values

The DECODE function considers two `NA`s to be equivalent. When *expr* is `NA`, then the function returns the *result* of the first *search* that is also `NA`.

DEFAULTAWSEGSIZE

The DEFAULTAWSEGSIZE option holds the default maximum segment size for an analytic workspace created in your database session. The setting is in effect for the duration of your session. For each new session, DEFAULTAWSEGSIZE reverts to the default value.

Syntax

DEFAULTAWSEGSIZE = *n*

Arguments

n
A number of bytes.

Notes

AW SEGMENTSIZE Command and AW(SEGMENTSIZE) Function

To change the maximum size for new segments in an existing workspace, use the AW command with the SEGMENTSIZE keyword. To discover the current maximum size for new segments, use the AW function with the SEGMENTSIZE keyword.

Examples

Example 10–5 Displaying the Maximum Segment Size for a Session

The following statement lists the current maximum segment size for workspaces.

```
SHOW DEFAULTAWSIZE
```

Example 10–6 Setting the Maximum Segment Size for a Session

The following statement sets the maximum segment size to approximately 1/2 gigabyte.

```
DEFAULTAWSIZE = 536870910
```

DEFINE

The DEFINE command adds a new object to the analytic workspace. This entry describes the DEFINE command in general. The following entries discuss the use of the DEFINE command for creating specific types of object:

- DEFINE AGGMAP
- DEFINE COMPOSITE
- DEFINE DIMENSION
    - DEFINE DIMENSION (simple)
    - DEFINE DIMENSION (conjoint)
    - DEFINE DIMENSION CONCAT
    - DEFINE DIMENSION ALIASOF
- DEFINE FORMULA
- DEFINE MODEL
- DEFINE PARTITION TEMPLATE
- DEFINE PROGRAM
- DEFINE RELATION
- DEFINE SURROGATE
- DEFINE VALUESET
- DEFINE VARIABLE
- DEFINE WORKSHEET

Syntax

DEFINE *name object-type attributes* [AW *workspace*] [SESSION]

Arguments

name

A `TEXT` expression that is the name for the new object. Follow these guidelines when specifying a value for name:

- The name must consist of 1 to 64 characters. When you are using a multibyte character set, you can still specify 64 characters even when this requires more than 64 bytes. Each character may be a letter (`A-Z`), a number (`0-9`), an underline (_), or a dot (`.`). However, the following restrictions apply to the use of these characters:
  - The name cannot consist of a single dot (`.`) character or a single underscore (_) character.
  - The name cannot duplicate a reserved word. For more information on identifying reserved words, see RESERVED.
  - The first character in the name cannot be a number.
  - The first character cannot be a dot (`.`) when the second character is a number.
- By default Oracle OLAP creates the definition in the current workspace. To create the definition in a different attached workspace, you can specify a qualified object name for *name* or you can use the AW argument to specify the workspace. Do not use both.

Caution: Oracle OLAP does not warn you when you create an object that has the same name as an existing object in another attached workspace.

object-type

The type of object being defined. The default is VARIABLE. The object types are discussed in the subsections for the DEFINE command.

attributes

Attributes are different for each type of object. The attributes are listed in the entry for each object type.

AW *workspace*
The name of an attached workspace in which you wish to define the object. You can also specify a noncurrent attached workspace using a qualified object name for *name*. Do not use this phrase when qualified object name for *name*.

SESSION
Specifies that the object exists only in the current session. The object is created in the EXPRESS analytic workspace to which you have read-only access. When you close the current session, the object no longer exists.

Notes

Modifying Object Definitions

To modify an existing object definition, use CHGDFN.

Extending Object Definitions

A DEFINE statement defines a basic object definition.You can extend that definition to include a calculation specification, a long definition, properties, permissions, format information, and triggers.

Use DESCRIBE to view the basic definition of an object, its calculation specification, a long definition, properties, permissions, format information. To view the complete definition of an object (including its properties and triggers), use FULLDSC.

Adding a Calculation Specification A DEFINE statement to create a definition for an aggmap object, formula, model or program merely creates a definition for the object. It does not define the calculation specification for the object. After you define the object, you must explicitly add the calculation specification. You can add the specification using an Edit window in the OLAP Worksheet or using one of the following OLAP DML statements:

- AGGMAP to add an aggregation specification to the definition of an aggmap object.
- ALLOCMAP to add an allocation specification to the definition of an aggmap object.
- EQ to add a calculation specification to the definition of a formula.
- MODEL to add a calculation specification to the definition of a model.
- PROGRAM to add a calculation specification to the definition of a program.

Adding a Long Description Use LD to add a long description to the definition of an object.

Adding Properties to Objects Use PROPERTY to add one or more properties to the definition of an object.

Adding Permissions Use PERMIT to specify permissions for an object.

Adding Triggers Use the TRIGGER command to specify triggers for an object.

Triggering Program Execution When DEFINE Executes

Using a TRIGGER_DEFINE program, you can make the DEFINE command an event that will automatically execute an OLAP DML program. See "Trigger Programs" on page 1-14 for more information.

The NAME Dimension

When you execute a DEFINE command with the NAME dimension limited to less than all its values, the status of NAME is automatically limited to `ALL`.

Viewing Session Objects

Objects created with the SESSION keyword are stored in the analytic workspace named `EXPRESS` instead of the current analytic workspace. Therefore, statements that operate against the current analytic workspace (such as LISTNAMES) do not list session objects unless you do one of the following:

- Specify the `EXPRESS` analytic workspace in the statement (such as `LISTNAMES AW EXPRESS`)
- Make the `EXPRESS` analytic workspace the current analytic workspace by issuing an `AW ATTACH EXPRESS` statement.

DEFINE AGGMAP

The DEFINE command with the AGGMAP keyword adds a new aggmap object to an analytic workspace. An aggmap object is a specification for how Oracle OLAP allocates or aggregates variable data.

Note: Defining an aggmap merely creates an aggmap object in the analytic workspace; it does not define the calculation specification. The aggmap specification can either specify how to aggregate or how to allocate data:

- For information on coding an aggregation specification, see AGGMAP.
- For information on coding an allocation specification, see ALLOCMAP.

Syntax

DEFINE *aggname* AGGMAP [<*dims*...>][AW *workspace*][SESSION]

Arguments

aggname
The name of the object that you are defining. For general information about this argument, see the main entry for the DEFINE command.

AGGMAP
The object type when you are defining an aggmap.

dims
(Optional; retained for compatibility with earlier software versions.) When defining an aggmap object for aggregation (that is, an AGGMAP type aggmap), the names of the dimensions. You cannot specify a conjoint dimension as a base dimension in the definition or specification for the aggmap.

AW *workspace*
The name of an attached workspace in which you wish to define the object. For more about this argument, see the main entry for the DEFINE command.

SESSION
Specifies that the object exists only in the current session. For more information about this argument, see the main entry for the DEFINE command.

Notes

Creating Temporary or Custom Aggregates

Most aggmap objects are defined to calculate variable values that are dimensioned by permanent dimension members (that is, dimension members that persist from one session to another). However, at runtime, users might wish to aggregate or allocate data for their own use for forecasting or what-if analysis, or just because they want to view the data in an unforeseen way. Adding temporary members to dimensions and aggregating or allocating data for those members is sometimes called creating temporary or custom aggregates or allocations. For an example of creating temporary aggregates, see Example 16–38, "Creating Calculated Dimension Members with Aggregated Values" on page 16-84.

Examples

Example 10–7 Creating an Aggmap for Aggregation

Suppose you define a `sales` variable with the following statement.

```
DEFINE sales VARIABLE <time, product, geography>
```

Assume also that you have defined an aggmap named `sales.agg` with the following definition and specification.

```
DEFINE sales.agg AGGMAP <time, product, geography>
AGGMAP
RELATION time.r PRECOMPUTE (time NE 'Year99')
RELATION product.r PRECOMPUTE (product NE 'All')
RELATION geography.r
CACHE STORE
END
```

The `sales.agg` aggregation specification contains the preceding three RELATION statements and a CACHE statements. In this example, you are specifying that all of the data for the `time.r` hierarchy of the `time` dimension should be aggregated, except for any data that has a `time` dimension value of `Year99`. All of the data for the `product.r` hierarchy of the `product` dimension should be aggregated, except for any data that has the `product` dimension value of `ALL`. (In this example, the product dimension has a dimension value named ALL that represents all products in the hierarchy.) All geography dimension values are aggregated. The CACHE STORE command specifies that any data that is rolled up on the fly should be calculated just once and stored in the cache for other access requests during the same session.

Note that users should not have write access to the analytic workspace when CACHE STORE is set, because the data calculated during the session may be saved inadvertently.

In this example, any data value that dimensioned by a Year99 time value or an ALL product dimension value is calculated on the fly.

You can now use the sales.agg aggmap with an AGGREGATE command, such as the following.

```
AGGREGATE sales USING sales.agg
```

Example 10–8 Creating an Aggmap for Allocation

Suppose you have a sales variable that you defined with the following statement.

```
DEFINE sales VARIABLE <time, product, geography>
```

To allocate data from a source to cells in the sales variable that are specified by the time and product dimension hierarchies, you have created an ASCII disk file called salesalloc.txt, which contains the following aggmap definition and specification.

```
DEFINE sales.alloc AGGMAP
ALLOCMAP
RELATION time.r OPERATOR EVEN
RELATION product.r operator EVEN NAOPERATOR HEVEN
SOURCEVAL ZERO
CHILDLOCK DETECT
END
```

To include the sales.alloc aggmap in your workspace, execute the following statement.

```
INFILE 'salesalloc.txt'
```

The sales.alloc aggmap is now defined, and it contains the preceding two RELATION commands, the SOURCEVAL command and the CHILDLOCK command. You end the entry of statements into the aggmap with the END command. In this example, you are specifying that the first allocation of source values occurs down the `time` dimension hierarchy and that the source value is divided evenly between the target cells at each level of the allocation. The second allocation occurs down the `product` dimension hierarchy, with the source value again divided evenly between the target cells at each level of the allocation, and when the allocation encounters a deadlock, the source values is divided evenly between the target cells of the hierarchy including cells that have a basis value of `NA`. With the SOURCEVAL command you specify that after the allocation, ALLOCATE sets the value of each source cell to zero. With the CHILDLOCK command you specify that ALLOCATE detects the existence of locks on both a parent and a child element of a dimension hierarchy.

You can now use the `sales.alloc` aggmap with an ALLOCATE command, such as the following.

```
ALLOCATE sales USING sales.alloc
```

The preceding statement does not specify a basis or a target object so ALLOCATE uses the `sales` variable as the source, the basis, and the target of the allocation.

DEFINE COMPOSITE

The DEFINE command with the COMPOSITE keyword adds a new named composite to an analytic workspace.

Conceptually, you can think of a composite consisting of two structures:

- The composite object itself. The composite contains the dimension-value combinations (that is, a composite tuples) that Oracle OLAP uses to determine the structure of any variables dimensioned by the composite.
- An index between the composite values and its base dimension values.

You define a variable using one or more composites when you want to reduce the amount of `NA` values stored in the variable. Reducing the sparsity of a variable in this way results in more efficient data storage as discussed in "B-Tree and Hash Composites" on page 10-22.

> **Note:** Oracle OLAP also supports the use of unnamed composites as described in "Unnamed Composites" on page 10-24.

Syntax

DEFINE *name* COMPOSITE <*dims...*> [AW *workspace*] [*index-algorithm*] [SESSION]

where:

index-algorithm specifies the algorithm that Oracle OLAP uses to create an index that relates the composite values to its base dimension values. When you omit this optional argument, Oracle OLAP uses the value specified by the SPARSEINDEX option. Valid values for *index-algorithm* are:

```
BTREE
COMPRESSED
HASH
```

Arguments

name
The name of the object you are defining. For general information about this argument, see the main entry for the DEFINE command.

COMPOSITE
The object type when you are defining a named composite.

dims
The names of two or more dimensions or b-tree or hash composites that you want to be the base dimensions of the composite. When you specify `COMPRESSED` as the value of *index-algorith* , at least one of the dimensions must be a hierarchal dimension.

> **Note:** A compressed composite cannot be a base dimension of another composite. You cannot specify a compressed composite as one of the dimensions in *dims*.

The order of the dimensions in *dims* varies by the value you specify for *index-algorithm*:

- For b-tree or hash composites, specify the dimensions in fastest to slowest-varying order as discussed in "Effect of Dimension Order on Variable Storage" on page 10-74.
- For compressed composites, specify the dimensions in order by hierarchy depth and the degree of aggregation—shallowest to deepest, least to most aggregated. When these requirements conflict, experiment to determine the most effective order. Use values returned by OBJ function with the PHYSVALS keyword to evaluate the results of your experimentation.

You must define all the dimensions and named composites used in the list before defining the composite. DEFINE will automatically create any unnamed composites in the list for you.

AW *workspace*
The name of an attached workspace in which you wish to define the object. For more information about this argument, see the main entry for the DEFINE command.

BTREE
Specifies the creation of a b-tree index to relate composite values to base dimension values. BTREE is the standard indexing method for composites.

COMPRESSED
Specifies the creation of a compressed index to relate composite values to base dimension values. You specify COMPRESSED only when you want to create a

composite for a variable that will be aggregated using the AGGREGATE command and when at least one hierarchical dimension is specified in *dims*. (See "Compressed Composites" on page 10-23 for more information.)

HASH
Specifies the creation of a hash index to relate composite values to base dimension values. HASH is rarely used and, then, typically, only when the composite has two or three dimensions.

SESSION
Specifies that the object exists only in the current session. When you close the current session, the object no longer exists.

Notes

B-Tree and Hash Composites

For a variable that is dimensioned by a b-tree or hash composite, Oracle OLAP creates variable cells only for those dimension values that are stored in the tuples of the composite; it does not create a cell for every value in the base dimensions. Data for the variable is stored in order, cell by cell, for each tuple in the composite. From the perspective of data storage, each combination of base dimension values in a composite is treated like the value of a regular dimension. This means that when you define a variable with one regular dimension and one composite, the data for the variable is stored as though it was a two-dimensional variable.

Before Oracle OLAP populates a cell in a variable dimensioned by a b-tree or hash composite, it first determines if the dimension-value combination for that variable cell is already in the composite (that is, if the composite tuple exists). When the composite tuple exists, Oracle OLAP stores the data in the variable using the existing composite structure. When the composite tuple does *not* exist, then Oracle OLAP populates the composite and the composite index before it stores the data in the variable. For an example of populating a variable with a composite, see Example 10–29, "Defining a Variable That Uses a Named B-Tree Composite" on page 1-14.

Unshared and Shared Composites

The actual sparsity of a variable dimensioned by a composite varies depending on whether or not the composite is an unshared composite or a shared composite:

- An **unshared composite** is a b-tree, hash, or compressed composite that is used to dimension only one variable. An unshared composite is populated only when the variable that uses it is populated. Consequently, an unshared composite perfectly reflects the sparsity of the variable that it is used to dimension. It only has the dimension value combinations for each non-`NA` value in that variable.

- A **shared composite** is a b-tree or hash composite that is used to dimension more than one variable. (A compressed composite can dimension only one variable or one partition of a variable. A compressed composite cannot be a shared composite.) A shared composite is populated when *any* of the variables that use it are populated. A shared composite has *all* of the dimension value combinations for non-`NA` values for *all* of the variables that it dimensions. A shared composite reflects the sparsity of *all* of the variable that it is used to dimension. Typically, therefore, variables dimensioned by shared composites are not perfectly sparse variables.

When the size of variables is important, when you have variables that will be sparse along the same dimensions, but will have significantly different patterns of sparsity, define different composites for the different variables.

Compressed Composites

In some cases, when you aggregate data in a variable dimensioned by a composite defined with one or more hierarchical dimension, one parent node may have only one descendant node — and so on all the way up to the top level. When a variable has a good deal of this type of sparsity, use a compressed composite as the dimension of the variable. Dimensioning this type of variable with a compressed composite creates the smallest possible variable, composite, and composite index—much smaller than if you dimension a variable with a b-tree or hash composite.

This reduction in size does not occur at the detail level. Oracle OLAP creates composite values for detail level the same way for all composites. A composite contains one composite tuple for each set of base dimension values that identifies non-NA detail data in the variables that use it.

The reduction in size occurs for those sets of base dimension values that identify non-NA data at higher levels of hierarchical dimensions. Oracle OLAP populates these higher-level values differently depending on whether a variable is dimensioned by a b-tree, hash, or compressed composite:

- For variables dimensioned by b-tree and hash composites, Oracle OLAP creates composite tuples for non-NA data at higher levels the same way that it does for non-NA data at the detail level. There is one composite tuple (with its own physical position) for each set of base dimension values that identifies non-NA data. The composite index contains all of the index entries needed to relate the composite tuple to the base dimension values.

- For variables dimensioned by compressed composites, Oracle OLAP reduces redundancy in the variable, composite, and composite index by using the"intelligence" of the AGGREGATE command that populates the variable. For sets of base dimension values that represent parent nodes, Oracle OLAP creates a physical position in the composite *only* for those tuples that represent a parent with more than one descendant. Oracle OLAP then creates an index between this composite structure and the base dimensions and uses this composite structure as the dimension of the variable. Since the actual structure of a compressed composite is smaller than that of a b-tree or hash composite, a variable dimensioned by a compressed composite is also smaller than a variable dimensioned by a b-tree or hash composite. Also, since the index for a compressed composite only has nodes for parents with more than one descendant, the index of a compressed composite has fewer levels and is smaller than the index of a b-tree composite.

Although performance varies depending on the depth of the hierarchies and the order of the dimensions in the composite, aggregating variables defined with compressed composites is typically much faster than aggregating variables defined with b-tree or hash composites.

Unnamed Composites

Oracle OLAP automatically defines an unnamed composite when a DEFINE VARIABLE statement with a SPARSE <*dimlist*> phrase executes. An unnamed composite can have either a b-tree or hash index. The type of index is determined by the value of the SPARSEINDEX option at the time that Oracle OLAP defines an unnamed composite.

Once Oracle OLAP has created a definition for an unnamed composite for a certain dimension list, it uses that composite any time you define a variable with the same SPARSE <*dimlist*> phrase. Thus all variables that are defined with the same SPARSE <*dimlist*> phrase share the same unnamed composite. For more information on sharing composites, see "Unshared and Shared Composites" on page 10-22.

Examples

Example 10–9 Creating a Named B-Tree Composite

Assume that the value of SPARSEINDEX is `BTREE`. The following statements define two objects: a named composite that has a b-tree index and base dimensions of `market` and a variable called `expenses` that is dimensioned by the `month` dimension and the `market.product` composite.

```
DEFINE market.product COMPOSITE <market product>
DEFINE expenses DECIMAL <month market.product <market product>>
```

DEFINE DIMENSION

The DEFINE command with the DIMENSION keyword adds a new dimension object to an analytic workspace. A dimension is a list of values that provides an index to the data.

Because the syntax of the DEFINE DIMENSION command is different depending on the type of the dimension that you are defining, four separate entries are provided:

- DEFINE DIMENSION (simple) for defining a dimension with unique values of the same data type.
- DEFINE DIMENSION (DWMQY) for defining a non-hierarchical dimension whose values represent a time period (day, week, month, quarter, or year).
- DEFINE DIMENSION (conjoint) for defining a dimension over two or more other base dimensions when the base dimesnions do not contain duplicate values or have different data types and when you want to explicitly specify the dimension value combinations.
- DEFINE DIMENSION CONCAT for defining a dimension over two or more other base dimension when the base dimensions contain duplicate values or different data types or when you want Oracle OLAP to automatically populate the dimension value combinations.
- DEFINE DIMENSION ALIASOF for defining an alias for a simple dimension.

DEFINE DIMENSION (simple)

The DEFINE DIMENSION (simple) command defines a simple dimension. A simple dimension is a list of unique data values with the same data type. A simple dimension can be a flat dimension or a hierarchical dimension that contains values from different levels of a hierarchy.

Tip: To create a hierarchical dimension using duplicate values or values of different data types, use a concat dimension as described in DEFINE DIMENSION CONCAT.

Syntax

DEFINE *name* DIMENSION *type* [TEMP] [AW *workspace*] [SESSION] where:

type is the data type of the dimension. The syntax of *type* varies depending on the data type:

TEXT [WIDTH *n*]

NTEXT [WIDTH *n*]

ID

INTEGER

NUMBER(*precision* [, *scale*])

Arguments

name
The name of the object you are defining. For general information about this argument, see the main entry for the DEFINE command.

DIMENSION
The object type when you are defining a dimension.

TEXT
Specifies that the values of the dimension have the TEXT data type which is equivalent to the CHAR and VARCHAR2 data types in the Oracle database. This data type stores up to 4000 bytes for each line in the database character set.

NTEXT
Specifies that the values of the dimension have the NTEXT data type which is equivalent to the NCHAR and NVARCHAR2 data types in the Oracle Database. This data type stores up to 4000 bytes for each line in UTF-8 character encoding.

ID
Specifies a special text data type that stores up to 8 single-byte characters for each line in the database character set.

WIDTH *n*
For TEXT or NTEXT dimensions, the width, in bytes, of the storage area of each value of an object. Valid width values are `1` through `4000`. Specify a fixed width only when you are certain that the values of a particular dimension are of similar size. When a value exceeds the specified width, it will be truncated.

INTEGER
Specifies that the values of the dimension have the INTEGER data type. The data type for a dimension with values that are identified by their numeric position (1, 2, and so on). A data type of INTEGER means that the dimension has no character values. For ease of use, you should use a text or time period data type, when possible.

NUMBER
Specifies that the values of the dimension have the NUMBER data type. A NUMBER dimension differs from other dimensions in that its values cannot be specified by position, only by value. To specify the values of a NUMBER dimension by position, you can define an INTEGER type dimension surrogate for the NUMBER dimension.

precision
The total number of digits a value of type NUMBER can have.

scale
The number of digits a value of type NUMBER can have to the right of a decimal point. For example, when you specify a precision of 7 and a scale of 2, then the highest value that the dimension can have is 99999.99. When you do not specify a scale value, then the scale is 0.

TEMP
Indicates that the dimension's values are only temporary and only for the current session. The dimension has a definition in the current workspace and can contain values during the current session. However, when you update and commit, only the definition of the dimension is saved. When you leave end your session or switch to another workspace, the data values are discarded. Each time you start the workspace, the values of a temporary dimension are `NA`.

AW *workspace*
The name of an attached analytic workspace in which you wish to define the dimension. Any objects dimensioned by the dimension must be defined in the same workspace. For general information about this argument, see the main entry for the DEFINE command.

SESSION
Specifies that the object exists only in the current session. When you close the current session, the object no longer exists.

Examples

Example: Defining a Simple Dimension

This example adds the dimension `city` to a workspace. You can attach a description to the object immediately after defining it. (You can also add the description later when you use the CONSIDER and LD commands.) After defining the dimension `city`, you can give it values with the MAINTAIN command.

The statements

```
DEFINE city DIMENSION ID
LD List of cities
MAINTAIN city ADD 'Boston' 'Chicago' 'Dallas' 'Seattle'
DESCRIBE city
```

produce the following definition.

```
DEFINE city DIMENSION ID
LD List of cities
```

DEFINE DIMENSION (DWMQY)

The DEFINE DIMENSION (DWMQY) command defines a special type of dimension whose values represent time periods.

Note: After defining a DWMQY dimension, you can use the VNF command to add a value name format to the dimension's definition. The VNF command controls the format for entering dimension values as well as the format for showing them in output.

Note: When you want to aggregate over time do not define the time dimension as a DWMQY dimension since you cannot aggregate over dimensions of this type. Instead, define the time dimension as a hierarchical dimension of type `TEXT` or `NTEXT`.

Syntax

DEFINE *name* DIMENSION *dwmqy* [TEMP] [AW *workspace*] [SESSION] where:

dwmqy is the time period of the dimension. The valid types for *dwmqy* are DAY, WEEK, MONTH, QUARTER, and YEAR. Each type indicates the span of the time period represented by the individual dimension values of the dimension. The syntax of *dwmqy* varies depending on the type:

DAY

[*multiple*] WEEK [BEGINNING [*phase*]| ENDING [*phase*]]

[*multiple*] MONTH [BEGINNING *phase* | ENDING *phase*]

QUARTER [BEGINNING *phase* | ENDING *phase*]

YEAR [BEGINNING *phase* | ENDING *phase*]

Arguments

name
The name of the object you are defining. For general information about this argument, see the main entry for the DEFINE command.

DIMENSION
The object type when you are defining a dimension.

multiple
For the WEEK and MONTH types, specifies time periods that span a multiple number of weeks or months. With the WEEK keyword, *multiple* can be an integer from 2 to 52. With the MONTH keyword, *multiple* can be 2, 3, 4, or 6.

BEGINNING *phase*
ENDING *phase*
Specifies the beginning or ending phase of a WEEK, MONTH, QUARTER, or YEAR dimension:

- For single weeks, *phase* can be a day of the week (corresponding to a name in the DAYNAMES option) or a date.
- For multiple weeks, *phase* must be a date.
- For months, quarters, or years, *phase* must be a month, expressed as a month name (corresponding to a name in the MONTHNAMES option) or as a date.

When you specify *phase* as a date, you give the month, day, and year, enclosed in single quotes, using any of the input styles that are valid for variable values with a data type of DATE. When you specify a date with an ambiguous meaning (such as

`'03 05 97'`), the date is interpreted according to the current setting of the DATEORDER option.

> **Note:** When you define a multiple-period dimension of type WEEK but you do not specify a BEGINNING or an ENDING argument, DEFINE automatically supplies a phase that begins with the date `'31DEC1899'`.

TEMP
Indicates that the dimension's values are only temporary and only for the current session. The dimension has a definition in the current workspace and can contain values during the current session. However, when you update and commit, only the definition of the dimension is saved. When you leave end your session or switch to another workspace, the data values are discarded. Each time you start the workspace, the values of a temporary dimension are `NA`.

AW *workspace*
The name of an attached analytic workspace in which you wish to define the dimension. Any objects dimensioned by the dimension must be defined in the same workspace. For general information about this argument, see the main entry for the DEFINE command.

SESSION
Specifies that the object exists only in the current session. When you close the current session, the object no longer exists.

Notes

Implicit Relations Between DAY, WEEK, MONTH, QUARTER, and YEAR Dimensions

When you define two or more dimensions of type DAY, WEEK, MONTH, QUARTER, or YEAR, Oracle OLAP automatically defines implicit relations between the values of the dimensions. For example, when you define a dimension of type MONTH and a dimension of type YEAR, Oracle OLAP automatically defines a relation that associates all the MONTH values that fall within a particular year with the corresponding value of the YEAR dimension.

Using BEGINNING or ENDING Phase to Organize Data by Fiscal Calendar

For dimensions of type MONTH, QUARTER, and YEAR, the BEGINNING *phase* or ENDING *phase* argument is especially useful for data organized on a fiscal-year calendar.

By specifying a phase for a dimension of type MONTH or QUARTER, you identify the time period that is the first or last period within a year. For example, when you define a dimension of type MONTH with an ending phase of June, then June is identified as the twelfth month of the year. When a dimension of type QUARTER has an ending phase of June, the quarter ending in June is identified as the fourth quarter of the year. When you give a dimension a VNF that includes a period code, you can enter or report dimension values according to their period within the year.

By default, the single or multiple weeks in a dimension of type WEEK end on Saturday. The BEGINNING *phase* or ENDING *phase* argument lets you specify the day of the week on which each period begins or ends. For multiple-week periods, the phase argument also controls the starting or ending date for grouping the weeks into periods. By default, the starting point for grouping multiple weeks is December 31, 1899 (a Sunday).

However, the phase argument does not determine the period that is counted as the first period within a year. For dimensions of type WEEK, Period 1 in a given calendar year is always the first period that ends in that year. For example, suppose you specify a dimension of type WEEK with a four-week period ending on June 7, 1997. DEFINE works backward and forward from this date, forming weeks into four-week periods. For 1997, Period 1 will be the period beginning on December 22, 1996 and ending on January 18, 1997.

Examples

Example 10–10 Defining a YEAR Dimension

The following statement defines a dimension of type YEAR that will hold values for fiscal years that end on June 30.

```
DEFINE fyear DIMENSION YEAR ENDING june
```

After defining the dimension, you can give it a description and a VNF (value name format). You can use the MAINTAIN command to give values to the dimension.

```
LD Fiscal years ending June 30
VNF 'FY<ff>'
MAINTAIN fyear ADD 'FY97' 'FY00'
```

Example 10–11 Using the Default Phrase for Date in an ENDING Phrase

This example illustrates how DEFINE automatically supplies a phase that begins with the date `'31DEC1899'` when you define a multiple-period dimension of type WEEK but you do not specify a BEGINNING *phase* or an ENDING *phase* argument. Assume that you issue the following statements

```
DEFINE twoweek DIMENSION 2 WEEK
DESCRIBE TWOWEEK
```

When you issue a DESCRIBE statement for `twoweek`, the following output is produced.

```
DEFINE twoweek DIMENSION 2 WEEK ENDING '13Jan1900'
```

DEFINE DIMENSION (conjoint)

The DEFINE DIMENSION (conjoint) command defines a conjoint dimension.

Conceptually, you can think of a conjoint dimension consisting of two structures:

- The dimension object itself. The values of the dimension are combinations of values of two or more other dimensions (that is, a conjoint tuples) that Oracle OLAP uses to determine the structure of any variables dimensioned by the conjoint dimension.
- An index between the conjoint dimension values and its base dimension values.

Composites are another object that you can use to dimension a variable using a list of dimension value combinations. See "Differences Between Conjoint Dimensions and Composites" on page 36 for a discussion of the major differences between composites and conjoint dimensions.

Syntax

DEFINE *name* DIMENSION <*dims. . .*> *index-algorithm* [AW *workspace*] [SESSION]

where:

index-algorithm specifies the algorithm that Oracle OLAP uses to create the index into the conjoint dimension. Valid values for *index-algorithm* are:

BTREE
NOHASH
HASH

Arguments

name

The name of the conjoint dimension you are defining. For general information about this argument, see the main entry for the DEFINE command.

DIMENSION

The object type when you are defining a conjoint dimension.

dims

One or more previously defined dimensions that are the base dimensions of the conjoint dimension. Specify the dimensions in fastest to slowest-varying order as discussed in "Effect of Dimension Order on Variable Storage" on page 10-74. You must enclose the dimension list in angle brackets.

Typically, a base dimension of a conjoint dimension is a simple dimension, but it can also be another conjoint dimension. You cannot have as base dimensions a simple dimension and a conjoint or concat dimension that has same simple dimension as one of its bases. For example, the following definitions are permissible.

```
DEFINE conjointdim.a DIMENSION <simpledim.b, simpledim.c>
DEFINE conjointdim.b DIMENSION <simpledim.a, simpledim.b>
DEFINE conjointdim.c DIMENSION <simpledim.a, conjointdim.a>
```

However, the following definition is *not* permitted because the same simple dimension, `simpledim.a`, is a base dimension of `conjointdim.d` and a component of `concatdim.a`.

```
DEFINE conjointdim.d DIMENSION <simpledim.a, concatdim.a>
```

The following definition is *not* permitted because the same simple dimension is a base dimension of `conjointdim.e` and a base dimension of `conjointdim.a`.

```
DEFINE conjointdim.e DIMENSION <simpledim.a, conjointdim.b>
```

BTREE
Specifies the creation of a b-tree index to relate conjoint values to base dimension values. Typically, you specify BTREE as the index algorithm for a conjoint dimension.

Note: When you are unsure whether to specify BTREE or NOHASH, use NOHASH, since you can always use the CHGDFN command to change a NOHASH conjoint into a BTREE conjoint, while you can use the CHGDFN command to change a BTREE conjoint into a NOHASH conjoint only when the conjoint was originally defined as a NOHASH conjoint.

NOHASH
Specifies that Oracle OALP does *not* create an index for the conjoint dimension, but instead uses internal structures to relate conjoint values to base dimension values.

Note: Because no index is created for NOHASH, NOHASH decreases the number of structures associated with the conjoint dimension; and, in many cases, decreases the time it takes to load and access conjoint dimension values. However, NOHASH is used infrequently, as it is a complicated algorithm that, on occasion, can result in unpredictable performance.

HASH
(Not recommended.) Specifies the creation of a has index to relate conjoint values to base dimension values. (Default)

Important: Even though HASH is the default, typically, you specify BTREE as the index algorithm for a conjoint dimension. When your conjoint dimension has more than 3 base dimensions, for best performance, use BTREE instead of HASH.

AW *workspace*
The name of an attached analytic workspace in which you wish to define the dimension. Any objects dimensioned by the dimension must be defined in the same workspace. For general information about this argument, see the main entry for the DEFINE command.

SESSION

Specifies that the object exists only in the current session. When you close the current session, the object no longer exists.

Notes

Differences Between Conjoint Dimensions and Composites

You can use either a composite or a conjoint dimension to dimension a variable with a list of dimension value combinations. Keep the following points in mind when deciding on which type of object to use:

- Object population maintenance—Conjoint dimensions offer the most control, while composites provide the greatest ease of use:
  - Oracle OLAP determines the dimension value combinations stored in a composite. Oracle OLAP populates a composite automatically when a variable dimensioned by composite is populated.
  - You determine the dimension value combinations that are stored in a composite. You must explicitly populate and maintain a conjoint dimension using MAINTAIN statements the same way you populate and maintain other dimensions.
- Dimension operations —You can perform dimension operations on conjoint dimensions, but not composites; however, you can only perform dimension operations on the base dimensions of composites. For example, you can LIMIT conjoint dimensions, but you must limit the base dimensions of a composite to limit your view to a subset of composite values; and you can define relations using conjoint dimensions, but not composites.

For more information on composites, see DEFINE COMPOSITE.

Relationship of Conjoint Dimensions to Base Dimensions

The values of the conjoint dimension are related to the base dimensions. You can specify data in a variable dimensioned by the conjoint dimension using the conjoint value combinations, the individual values of the base dimensions, or other dimensions related to either of the base dimensions of the conjoint dimension.

Defining a Subset of a Dimensions Values

You can have a conjoint dimension with only one base dimension, which enables you to create a subset of that dimension's values. You must still enclose that one base dimension within angle brackets.

Using Conjoint Dimension Values in Expressions

To refer to the value of a conjoint dimension in an expression, specify the value following these guidelines:

- Enclose the entire dimension value specification in angle brackets and then enclose this entire specification in single quotes; do not enclose the individual values in single quotes.
- Use the exact upper- and lowercase spellings for the base dimension values.
- When the specification includes a text value with an embedded blank, you must separate the dimension values with commas.

For example, when `item.org` is a conjoint dimension with base dimensions `item` and `org`, use the following format to refer to values of `item.org`.

```
'<Expenses, Direct Sales>'
```

Examples

Example 10–12 Defining a Conjoint Dimension

Assume that you have defined and populated the simple dimensions `city`, `state`, and `region` and that they have the following values.

```
CITY             STATE          REGION
---------       ----------      ------
Princeton       New Jersey      East
Newark          New Jersey      Central
Patterson       New York
New York        Illinois
Chicago         Indiana
```

To define a conjoint dimension named `cityandstate` and add values to it use the following OLAP DML statements.

```
DEFINE cityandstate DIMENSION <city state>
MAINTAIN cityandstate add <'Princeton' 'New Jersey'>
MAINTAIN cityandstate add <'Newark' 'New Jersey'>
MAINTAIN cityandstate add <'Patterson' 'New Jersey'>
MAINTAIN cityandstate add <'New York' 'New York'>
MAINTAIN cityandstate add <'Chicago' 'Illinois'>
MAINTAIN cityandstate add <'Princeton' 'Indiana'>
```

DEFINE DIMENSION CONCAT

The DEFINE DIMENSION CONCAT commands defines a concat dimension. A concat dimension is a dimension that groups a set of base dimensions with duplicate values or different data types into one dimension.

When there are duplicate data values, you create a non-unique concat dimensions. For example, you would create a nonunique dimension for a geography hierarchy when "New York" is both the value at the city level and at the state level. When all of the data values in all of the base dimensions are unique, you can create a unique concat dimension.

Note: The way that you specify the values of concat dimension varies depending on whether or not the concat dimension is a unique or nonunique concat dimension. See Values of CONCAT Dimensions on page 10-40 for more information.

Syntax

DEFINE *name* DIMENSION CONCAT(*basedimlist*. . .)[UNIQUE] [TEMP] [AW *workspace*] [SESSION]

Arguments

name
The name of the object you are defining. For general information about this argument, see the main entry for the DEFINE command.

DIMENSION CONCAT
The object type when you are defining a concat dimension.

basedimlist
One or more previously-defined dimensions that are the base dimensions of the concat dimension. Specify the dimensions in fastest to slowest-varying order as discussed in "Effect of Dimension Order on Variable Storage" on page 10-74. You must enclose the dimension list in parenthesis.

The types of dimensions that can be base dimensions varies depending on whether you are defining a unique or nonunique concat dimension:

- When defining a non-unique concat dimension, a base dimension can be a simple dimension of any data type, a conjoint dimension, or another concat dimension.
- When defining a unique concat dimension, a base dimension can be a simple dimension of type TEXT or ID, or another unique concat dimension as long as the data values of all of the base dimensions are unique and not duplicated in any of the base dimensions.

A composite cannot be the base dimension of a concat dimension.

Simple dimensions and conjoint dimensions are the bottom-level components of a concat dimension. When you specify a concat dimension as a base dimension when defining a concat, then the base dimensions of that inner concat are *component* dimensions of the outer concat.

The same dimension cannot appear more than once in the component dimensions of a concat dimension. However, in a concat, a conjoint dimension is an indivisible unit and Oracle OLAP does not consider the base dimensions of a conjoint in the definition of the concat. Therefore, a simple dimension can be a base dimension of a conjoint and that conjoint and the same simple dimension can be base dimensions (or components) of a concat dimension.

For example, the following definitions are permissible.

```
DEFINE conjointdim.a DIMENSION <simpledim.b, simpledim.c>
DEFINE conjointdim.b DIMENSION <simpledim.a, simpledim.b>
DEFINE conjointdim.c DIMENSION <simpledim.a, conjointdim.a>
DEFINE concatdim.a DIMENSION CONCAT (simpledim.a, conjointdim.a)
DEFINE concatdim.b DIMENSION CONCAT (simpledim.a, conjointdim.b)
DEFINE concatdim.c DIMENSION CONCAT (simpledim.b, conjointdim.b)
DEFINE concatdim.d DIMENSION CONCAT (simpledim.a, concatdim.c)
```

In the definition of `concatdim.a`, the base dimensions are `simpledim.a` and `conjointdim.a`. In the definition of `concatdim.d`, the base dimensions are `simpledim.a` and `concatdim.c`. The component dimensions of `concatdim.d` are `simpledim.a`, `simpledim.b`, and `conjointdim.b`. `simpledim.a` and `simpledim.b` appear only once as component dimensions even though they are the base dimensions of `conjointdim.b` because the base dimensions of a conjoint are not component dimensions of a concat.

However, the following definition is *not* permitted because the same simple dimension is a base dimension of `concatdim.e` and a component of `concatdim.e` because it is a base dimension of `concatdim.b`.

```
DEFINE concatdim.e DIMENSION CONCAT (simpledim.a, concatdim.b)
```

Note: The simple dimensions in the *basedimlist* argument, and the simple dimensions that are base dimensions of any conjoint dimensions or concat dimensions in *basedimlist*, can*not* have an INTEGER data type.

UNIQUE
Specifies that the text values of the base dimensions are unique. When you specify this keyword, the dimensions listed in *basedimlist* must be either simple text or ID dimensions or unique concat dimensions.

TEMP
Indicates that the dimension's values are only temporary and only for the current session. The dimension has a definition in the current workspace and can contain values during the current session. However, when you update and commit, only the definition of the dimension is saved. When you leave end your session or switch to another workspace, the data values are discarded. Each time you start the workspace, the values of a temporary dimension are `NA`.

AW *workspace*
The name of an attached analytic workspace in which you wish to define the dimension. Any objects dimensioned by the dimension must be defined in the same workspace. For general information about this argument, see the main entry for the DEFINE command.

SESSION
Specifies that the object exists only in the current session. When you close the current session, the object no longer exists.

Notes

Values of CONCAT Dimensions

Once you have defined a unique CONCAT dimension, you can refer to its values simply by specifying the values of the base dimensions.

However, you must specify the values of a nonunique CONCAT dimension as a concatonation of the name of the base dimensions and the base dimension values separated by a colon (:) and a space and enclosed in angle brackets(<>). In an expression, use the following format.

<BASE_DIMENSION_NAME: base_dimension value>

For example, assume that you have defined the base dimensions named `city` and `state` and, a CONCAT dimension for them named `geog`. When you report on the geog dimension, the values of geog include the names of the base dimensions along with the values.

```
DEFINE city DIMENSION TEXT
DEFINE state DIMENSION TEXT
DEFINE geog DIMENSION CONCAT(city state)
MAINTAIN city ADD 'New York'
MAINTAIN state ADD 'New York'
REPORT geog

 GEOG
-----------------------------------
<CITY: New York>
<STATE: New York>
```

Examples

Example 10–13 Defining a CONCAT Dimension

Assume that you have defined and populated the simple dimensions `city`, `state`, and `region` and that they have the following values.

```
CITY             STATE           REGION
---------       ----------       ------
Princeton       New Jersey       East
Newark          New Jersey       Central
Patterson       New York
New York        Illinois
Chicago         Indiana
```

You define a concat dimension based on these dimensions using the following OLAP DML statement.

```
DEFINE geog DIMENSION CONCAT(region cityandstate)
```

The values of geog are the following.

```
<REGION: East>
<REGION: Central>
<CITYANDSTATE: <Princeton New Jersey>>
<CITYANDSTATE: <Newark New Jersey>>
<CITYANDSTATE: <Patterson New Jersey>>
<CITYANDSTATE: <New York New York>>
<CITYANDSTATE: <Chicago Illinois>>
<CITYANDSTATE: <Princeton Indiana>>
```

DEFINE DIMENSION ALIASOF

The DEFINE DIMENSION ALIASOF command defines a dimension alias for a simple dimension. An alias dimension has the same type and values as its base dimension. Typically, you define an alias dimension when you want to dimension a variable by the same dimension twice.

Additionally, You can use the LIMIT command to limit alias dimensions and define variables and relations using an alias dimension. However, you cannot maintain an alias dimension directly; instead you maintain its base dimension using MAINTAIN.

Syntax

DEFINE *name* DIMENSION ALIASOF *dimension* [TEMP] [AW *workspace*] [SESSION]

Arguments

name
The name of the object you are defining. For general information about this argument, see the main entry for the DEFINE command.

DIMENSION ALIASOF
The object type when you are defining a dimension. Indicates that the dimension being defined is an alias for another dimension.

dimension
The name of a simple dimension for which you want to define an alias. This dimension cannot be a concat or conjoint dimension, composite, or surrogate.

TEMP
Indicates that the dimension's values are only temporary and only for the current session. The dimension has a definition in the current workspace and can contain

values during the current session. However, when you update and commit, only the definition of the dimension is saved. When you leave end your session or switch to another workspace, the data values are discarded. Each time you start the workspace, the values of a temporary dimension are NA.

AW workspace
The name of an attached analytic workspace in which you wish to define the dimension. Any objects dimensioned by the dimension must be defined in the same workspace. For general information about this argument, see the main entry for the DEFINE command.

SESSION
Specifies that the object exists only in the current session. When you close the current session, the object no longer exists.

Examples

Example 10–14 Defining an Alias Dimension

Assume that your department has multiple projects that employees participate in and that an employee may be a leader of one project and a participant in another. Assume also that you want to track the hours that each employee participates in a project as either a leader or a participant. In order to keep track of this information, you can design a variable that is dimensioned by the time you want to track by (in this example, `year`), `project`, and two dimensions for employee—one dimension named `employee` for employee as participant and another dimension named `leader` for employee as leader. The following definitions support this structure.

```
DEFINE year DIMENSION TEXT
DEFINE project DIMENSION TEXT
DEFINE employee DIMENSION TEXT
DEFINE leader DIMENSION ALIASOF employee
DEFINE hours VARIABLE INTEGER <year project employee leader>
```

The following statements populate all of the dimensions.

```
MAINTAIN year ADD '2001' '2002' '2003'
MAINTAIN project ADD 'projA' 'projB'
MAINTAIN employee add 'Adams' 'Baker' 'Charles'
```

Note that you do not have to explicitly populate the alias dimension (that is, `leader`). When you populate the `employee` dimension, it's alias dimension `leader`, is also populate as you can see when you issue REPORT statements for all four dimensions.

```
YEAR
--------------
2001
2002
2003

PROJECT
--------------
projA
projB

EMPLOYEE
--------------
Adams
Baker
Charles

LEADER
--------------
Adams
Baker
Charles
```

You can limit a dimension without limiting its alias; or limit an alias without limiting the dimension for which it is an alias. For example, when you issue the following statements to limit employee to Adams for project ProjA in year 2001, a report displays all of the leaders of the projects that Adams participates in.

```
LIMIT year TO '2001'
LIMIT employee TO 'Adams'
LIMIT project TO 'projA'
REPORT DOWN leader ACROSS employee: hours

PROJECT: projA
YEAR: 2001
               --HOURS---
               -EMPLOYEE-
LEADER           Adams
-------------- ----------
Adams                   1
Baker                   2
Charles                 1
```

On the other hand, when you limit `leader` to Adams for `project` ProjA in `year` 2001, a report displays all of the employees of the projects that Adams leads.

```
LIMIT employee TO ALL
LIMIT leader TO 'Adams'
LIMIT project TO 'projA'
REPORT DOWN leader ACROSS employee: hours

PROJECT: projA
YEAR: 2001
               -------------HOURS--------------
               ------------EMPLOYEE------------
LEADER           Adams      Baker     Charles
-------------- ---------- ---------- ----------
Adams                   1          3          3
```

DEFINE FORMULA

The DEFINE command with the FORMULA keyword adds a new formula object to an analytic workspace. You define a formula to save an expression. A formula can take the place of an expression you use repeatedly. The name of the formula takes the place of the text of the expression. Oracle OLAP does not store the data for a formula in a variable; instead it is calculated at runtime each time it is requested.

See also: "Formulas" on page 4-1

Syntax

DEFINE *name* FORMULA {*expression*|*datatype* [<*dimensions...*>]} [AW *workspace*] [SESSION]

Arguments

name
The name of the object you are defining. For general information about this argument, see the main entry for the DEFINE command.

FORMULA
The object type when you are defining a formula.

expression
The calculation to be performed to produce values when you use the formula. It can be any valid expression, including a constant or the name of a variable as described in Chapter 3, "Expressions".

You can specify an expression for a formula when you define it or after you define using an EQ statement. When you define a formula without specify an expression, a formula returns `NA` with the specified data type.

Note: Oracle OLAP does not automatically convert text in a formula to uppercase.

datatype
The intended data type for the formula. You can use any of the data types that apply to variables. The *datatype* argument is optional. When you include an expression in the formula definition, you do not specify a value. DEFINE automatically determines the data type.

However, when you do *not* include an expression in the definition, you must specify the data type. When you add the expression later using an EQ statement, its data type should match the type you specify now. When it does not, DEFINE converts the output to the specified type.

Tip: You can determine the data type of an expression before adding it to a formula by using the PARSE command and INFO (PARSE) function.

dimensions

The dimensions of the formula. Enclose the list in angle brackets. The *dimensions* argument is optional. When the formula is a single-cell value, you do not specify any dimensions. Also, when you include an expression in the definition, you do not specify a value. DEFINE automatically determines the dimensions.

However, when you do *not* include an expression in the definition, you must specify the dimensions. When you add the expression later using an EQ statement, the expression must have the same dimensions as the formula definition. When it does not, DEFINE forces the output to have the specified dimensions.

Restriction: You cannot define a formula that is dimensioned by a composite.

AW *workspace*

The name of an attached workspace in which you wish to define the formula. When the formula is dimensioned, it must be defined in the same workspace as its dimensions. For general information about this argument, see the main entry for the DEFINE command.

SESSION

Specifies that the object exists only in the current session. When you close the current session, the object no longer exists.

Notes

Effect of Changing the Characteristics of Objects Used by a Formula

When you change the name, data type, or dimensions of any of the objects used by a formula, the formula is *not* automatically updated. The formula causes an error when objects it refers to have been deleted or are now the wrong data type.

Storing Complex Expressions and Calculations

To define a very complex calculation, you can define a program that uses a RETURN command to return a value. You can then use the program as a function wherever you would use an expression or formula.

Examples

Example 10–15 Defining a Formula

This example adds a formula named `sales.diff` to a workspace. This formula calculates the percent difference between total sales for the current year and last year.

The statements

```
DEFINE sales.diff FORMULA LAGPCT(TOTAL(actual year) 1 year)
DESCRIBE sales.diff
```

produce the following definition.

```
DEFINE sales.diff FORMULA DECIMAL <year>
EQ lagpct(TOTAL(actual year) 1 year)
```

DEFINE MODEL

The DEFINE command with the MODEL keyword adds a new model object to an analytic workspace. A model is a set of interrelated equations. The calculations in an equation can be based either on variables or on dimension values. You can assign the results of the calculations directly to a variable or you can specify a dimension value for which data is being calculated. For example, in a financial application, all the equations might be based on the values of a line item dimension, and data would be calculated for line items such as total expenses and net income.

Note: Defining a model merely creates a model object in the analytic workspace. You must also code a specification for the model, as described in MODEL.

Syntax

DEFINE *name* MODEL [AW *workspace*] [SESSION]

Arguments

name
The name of the object you are defining. For general information about this argument, see the main entry for the DEFINE command.

MODEL
The object type when you are defining a model.

AW *workspace*
The name of an attached workspace in which you wish to define the object. For more information about this argument, see the main entry for the DEFINE command.

SESSION
Specifies that the object exists only in the current session. When you close the current session, the object no longer exists.

Examples

Example 10–16 Defining a Simple Model

This example shows a simple model named `income.calc` that calculates the line items in an income statement. The model equations are based on the `line` dimension in the `demo` workspace. First, define the model and give it an LD.

```
DEFINE income.calc MODEL
LD Model for calculating Income Statement items
```

Then use the MODEL command to enter the specification for the model. For this example, you can enter model lines such as the ones in the following model description.

```
DEFINE income.calc MODEL
LD Model for calculating Income Statement items
MODEL
dimension line
net.income = opr.income - taxes
opr.income = gross.margin - (marketing+selling+r.d)
gross.margin = revenue - cogs
END
```

To solve the model for the `actual` variable, enter data in `actual` for the input line items (`Revenue`, `Cogs`, `Marketing`, `Selling`, `R.D`, and `Taxes`). Then execute the following statement.

```
income.calc actual
```

DEFINE PARTITION TEMPLATE

The DEFINE command with the PARTITION TEMPLATE keywords adds a new partition template object to an analytic workspace. A partition template is a specification for the partitions of a partitioned variable. A partitioned variable is stored as multiple rows in the relational table of LOBs that is the analytic workspace—each partition is a row in the table.

You define both partitioned and unpartitioned variables using DEFINE VARIABLE statements. You must define a partition template object before you can define a partitioned variable.

Syntax

DEFINE *name* PARTITION TEMPLATE <*dimlist*> PARTITION BY

  {RANGE|LIST|CONCAT} (*dims_partitioned_by*) ([*partition_definition_statement*...]) [AW *workspace*]

where:

partition_definition_statement defines a partition. The syntax varies depending on whether you specify RANGE, LIST, or CONCAT:

- When you specify RANGE, the syntax for *partition_definition_statements* is:

  PARTITION *partition-name* VALUES LESS THAN *const-exp* <*partition-dimlist*>

- When you specify LIST, the syntax for *partition_definition_statements* is:

  PARTITION *partition-name* VALUES ([*valuelist*)] <*partition-dimlist*>

- When you specify CONCAT, the syntax for *partition_definition_statements* is:

  PARTITION *partition-name* <*partition_basedimlist*>

Arguments

name
The name of the object you are defining. For general information about this argument, see the main entry for the DEFINE command.

dimlist
A list of all of the logical dimensions for the variable that you are partitioning. You must enclose the names of the dimensions in a single set of angle brackets (< >). You

must define a dimension before you can include it in the definition of a partition template.

PARTITION BY RANGE
Indicates partitioning by values within a specified range. The syntax for *partition_definition_statements* is:

PARTITION *partition-name* VALUES LESS THAN *const-exp* <*partition-dimlist*>

PARTITION BY LIST
Indicates partitioning by listed values. The syntax for *partition_definition_statements* is:

PARTITION *partition-name* VALUES [(*valuelist*)] <*partition-dimlist*>

PARTITION BY CONCAT
Indicates partitioning by base dimensions of a concat dimension. The syntax for *partition_definition_statements* is:

PARTITION *partition-name* <*partition_basedimlist*>

dims_partitioned_by
The subset of dimensions specified by *dimlist* that actually specify the partitions of the variable. For range and list partitioning (that is, when you specify either the RANGE or LIST keywords), you can specify only one dimension for *dims_partitioned_by*.

Note: You cannot partition a variable along an `INTEGER` dimension.

PARTITION *partition-name*
The name of the partition.

VALUES LESS THAN
Indicates that you are specifying a RANGE partition by comparing values.

constant-exp
A constant expression that has the same data type as the data type of the dimension specified for *dims_partitioned_by*.

partition-dimlist
A list of all of the of dimensions of the partition template object (although the dimensions may be members of a composite). You must enclose the names of the

dimensions in a single set of angle brackets (< >). Use this argument to specify the composite (if any) used to dimension the partitions that correspond to *partition-name*. When you do not specify a value then the partition is dimensioned densely by all of the of dimensions of the partition template object.

VALUES
Indicates that you are specify a LIST partition by specifying values.

valuelist
A list of dimension values, separated by commas. You must surround text values with single quotes (for example, `'mytext'`). Specify values of conjoints by specify the values of the base dimensions, separated by a comma, in a single set of angle brackets (for example, `<'Value1', 'Value2'>`). Specify values of nonunique concat dimensions by specify the values of the base dimensions, separated by a colon, in a single set of angle brackets (for example, `<'Value1': 'Value2'>`).

> **Tip:** I f you want to use a valueset object to specify values, do not specify values for *valuelist*. Instead, omit *valuelist* from the partition template definition and use a MAINTAIN MOVE TO PARTITION statement to specify values for the partition.

basepartition-dimlist
A list of dimensions for the partition enclosed in a single set of angle brackets (< >). For every dimension of the partition template, *basepartition-dimlist* must include either that dimension, a base of that dimension, a concat of the base dimensions of that dimension, or a composite that includes that dimension, its base, or concat dimension of its base dimensions. In other words, *basepartition-dimlist* must have the same number of logical dimensions as the partition template itself (that is, the dimensions in *dimlist*).

Each listed dimension must be one of the following:

- A partition template object with the appropriate dimensionality specified for its *dimlist* parameter.
- A base dimension for one of the dimensions listed in *dims_partitioned_by*.
- A dimension of the partition template that is not in *dims_partitioned_by*.
- A composite consisting of dimensions of partition template object with the appropriate dimensionality specified for its *dimlist* parameter, a base dimension for one of the dimensions listed in *dims_partitioned_by*, or a dimension in *dimlist* that is not in *dims_partitioned_by*.

Examples

See: Examples of defining partition template objects are integrated into the following examples of defining partitioned variables:

- Example 10–26, "Defining a Variable with Internal Partitions" on page 10-77.
- Example 10–27, "Defining a Variable with External Partitions" on page 10-80.

DEFINE PROGRAM

The DEFINE command with the PROGRAM keyword adds a new OLAP DML program object to an analytic workspace. An OLAP DML program is a collection of OLAP DML statements that helps you accomplish some workspace management or analysis task.

Note: Defining a program merely creates a program object in the analytic workspace. You must also code the actual lines of the program, beginning with PROGRAM command.

Syntax

DEFINE *name* PROGRAM [*datatype*|*dimension*] [AW *workspace*] [SESSION]

Arguments

name
The name of the object you are defining. For general information about this argument, see the main entry for the DEFINE command.

PROGRAM
The object type when you are defining a program.

datatype
The data type of the value to be returned by the program when it is called as a function. You can use any of the data types that apply to variables.

dimension
The name of a dimension, whose value the program returns when it is called as a function. The return value is a single value of the dimension, not a position (integer). The dimension must be defined in the same workspace as the program.

AW *workspace*
The name of an attached workspace in which you wish to define the program. When the program returns a dimension, the program must be defined in the same workspace as the dimension. For general information about this argument, see the main entry for the DEFINE command.

SESSION
Specifies that the object exists only in the current session. When you close the current session, the object no longer exists.

Notes

Returning Values

Use a RETURN command in a program when you want it to return a value. The argument to the RETURN command is an expression that specifies the value to return. When the expression does not match the declared data type or dimension, the value is converted (if possible) to the declared data type or dimension value.

When you do not specify a data type or dimension in the definition of a program, its return value is treated as worksheet data. This means Oracle OLAP converts any return value to the data type required by the calling context. This may lead to unexpected results.

For a program to return a value, you must call the program as a function. That is, you must use it as an expression in a command. In the following example, the program `isrecent` is being treated as a function. It is an argument to the REPORT command.

```
REPORT isrecent(actual)
```

When the program returns values of a dimension, the program is in the output of the LISTBY function, and OBJ(ISBY) is `TRUE` for the dimension.

See the entries for the ARGUMENT, CALL, and RETURN commands for more information about programs as user-defined functions.

Returning NA

When you call the program as a function, but it does not use the RETURN command to provide a return value, the program returns `NA`.

Examples

Example 10–17 Basing Program Flow on Test Results

The `saleseval` program tests whether total sales for a month exceeds total planned sales for the month. The program executes different statements based on the results of the test.

```
DEFINE SALESEVAL PROGRAM
PROGRAM
ARGUMENT onemonth MONTH
VARIABLE excess DECIMAL
ALLSTAT
LIMIT month TO onemonth
IF TOTAL(sales, month) GT TOTAL(sales.plan, month)
   THEN DO
     excess = (TOTAL(sales, month) -
       - TOTAL(sales.plan, month)) -
       / TOTAL(sales.plan, month) * 100
     SHOW JOINCHARS('Sales exceeded plan by ' excess '%.')
     DOEND
ELSE SHOW JOINCHARS('We\'re not meeting plan. ' -
   'Let\'s get working!')
REPORT DOWN product W 10 ACROSS district: sales - sales.plan
END
```

When total sales for the month exceeds total planned sales for the month, the THEN command lines are executed. The program calculates the percentage by which actual sales exceeds planned sales and places the result in a numeric variable called `excess`. The program then sends the results to the current outfile. The JOINCHARS function is used to combine the calculated expression `excess` with the text expression "Sales exceeded plan by" in the output.

When total sales does not exceed planned sales, the ELSE command line is executed and a different message is produced.

After the THEN or ELSE command lines are executed, control flows to the next line in the program, and a report of sales in excess of plan is produced.

DEFINE RELATION

The DEFINE command with the RELATION keyword adds a new relation object to an analytic workspace. A relation describes a correspondence between the values of two or more dimensions. It can have dimensions, just like a variable, but the values of the relation must be values from the related dimension.

Syntax

DEFINE *name* RELATION *related-dim* [<*dimensions...*>] [TEMP] [AW *workspace*] [SESSION]

Arguments

name

The name of the object you are defining. For general information about this argument, see the main entry for the DEFINE command.

RELATION

The object type when you are defining a relation.

related-dim

Specifies the dimension to which one or more *dimensions* are related. A relation is normally used to store information about the relationship between two dimensions; for example, the cities that belong in each region.

In the definition, the dimension having fewer values is normally specified as the related dimension (for example, regions). The dimension having more values is normally specified as a dimension of the relation (for example, cities).

<dimensions...>

The names of the dimensions of the relation. You must enclose the names of the dimensions in a single set of angle brackets (< >). You must define a dimension before including it in the definition of a relation. Do not include composites in the dimension list.

Restriction: Oracle OLAP does not support the use of composites as dimensions for relations. Do not attempt to define them.

TEMP

Indicates that the values of the relation are only temporary. The relation is defined in the current workspace and can contain values during the current session. However, when you update and commit the workspace, only the definition of the relation is saved. When you end the session or switch to another workspace, the data values are discarded. Each time you start the workspace, the values of a temporary relation are `NA`.

AW *workspace*

The name of an attached workspace in which you wish to define the relation. The relation must be defined in the same workspace as its dimensions. For general information about this argument, see the main entry for the DEFINE command.

SESSION

Specifies that the object exists only in the current session. When the session ends, the object no longer exists. This differs from the TEMP keyword, which specifies that the values are temporary but the object definition remains in the workspace in which you create it.

Notes

Implicit Relations Between DAY, WEEK, MONTH, QUARTER, and YEAR Dimensions

When you define two or more dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR, Oracle OLAP automatically defines implicit relations between the values of the dimensions. For example, when you define a dimension of type MONTH and a dimension of type YEAR, Oracle OLAP automatically defines a relation that associates all the MONTH values that fall within a particular year with the corresponding value of the dimension of type YEAR.

Examples

Example 10–18 Creating, Populating, and Totaling by a Relation

The following example defines a relation between `division` and `product`, stores the values of the relation, and then totals `units` by `division`, even though `units` is dimensioned by `product`. The following statement defines the `div.prod` relation.

```
DEFINE div.prod RELATION division <product>
```

The following statements store values of `division` in `div.prod`.

```
LIMIT product TO 'Tents' 'Canoes'
div.prod = 'Camping'
LIMIT product TO 'Racquets'
div.prod = 'Sporting'
LIMIT product TO 'Sportswear' 'Footwear'
div.prod = 'Clothing'
```

You can use a REPORT command to see the values stored in `div.prod`.

```
report div.prod
```

This statement produces the following output.

```
PRODUCT        DIV.PROD
------------- ----------
Tents         Camping
Canoes        Camping
Racquets      Sporting
Sportswear    Clothing
Footwear      Clothing
```

The `div.prod` relation lets you look at division totals in a report, even though the data is dimensioned by `product`.

```
REPORT TOTAL(units division)
```

DEFINE SURROGATE

The DEFINE command with the SURROGATE keyword adds a a new surrogate object to an analytic workspace. A surrogate provides an alternative set of values for a dimension. You can use a surrogate rather than a dimension in a model, in a LIMIT command, in a qualified data reference, or in data loading with statements such as FILEREAD, FILEVIEW, SQL FETCH, and SQL IMPORT.

Note: You cannot specify a dimension surrogate as the dimension or related dimension argument when you define a concat dimension, a formula, a program, a relation, a valueset, or a variable. Additionally, in data loading you cannot create new dimension values using a dimension surrogate.

Syntax

DEFINE *name* SURROGATE *targetname type* [AW *workspace*] [SESSION]

where:

type has the following syntax:

[TEXT|NTEXT] [WIDTH *n*]|ID|INTEGER|NUMBER (*precision*[, *scale*])

Arguments

name
The name of the object you are defining. For general information about this argument, see the main entry for the DEFINE command.

SURROGATE
The object type when you are defining a dimension surrogate.

targetname
The name of the dimension for which you are creating a surrogate.

Note: Keep the following restrictions in mind when determining a target for your surrogate:

- You cannot create a surrogate for a dimension that has a type of DAY, WEEK, MONTH, QUARTER, or YEAR or for a composite.
- When you create a surrogate for a conjoint, you cannot convert the conjoint to a composite.

TEXT
NTEXT
ID
The data type for a dimension surrogate with text values. When all the values of a dimension surrogate are eight single-byte characters or less, give it a data type of ID. When one or more dimension values has more than eight single-byte characters, you must give it a data type of TEXT or NTEXT. For greater efficiency and ease of use, you should give dimensions a data type of ID whenever possible.

WIDTH *n*
For TEXT or NTEXT dimension surrogate, the width, in bytes, of the storage area of each value of an object. Valid width values are 1 through 4000. Specify a fixed width only when you are certain that the values of a particular dimension surrogate are of similar size. When a value exceeds the specified width, Oracle OLAP truncates it.

INTEGER
The data type for a dimension surrogate with values that are the ordinal positions (1, 2, and so on) of the values in its dimension. You might create an INTEGER type dimension surrogate for a NUMBER type dimension so that you can specify dimension values by position instead of by the value of the dimension. When you define an INTEGER type dimension surrogate, Oracle OLAP automatically assigns an integer value to the surrogate for each of the positions in the dimension.

NUMBER
Specifies that the dimension surrogate has a data type of NUMBER.

precision
Specifies the total number of characters in the value of a dimension surrogate of type NUMBER.

scale
Specifies the number of characters that can be to the right of a decimal point of a dimension surrogate of type NUMBER.

AW *workspace*
The name of an attached workspace in which you wish to define the dimension surrogate. The dimension for which you define the surrogate must be defined in the same workspace. For general information about this argument, see the main entry for the DEFINE command.

SESSION
Specifies that the object exists only in the current session. When you close the current session, the object no longer exists. Use this keyword when the definition of the *targetname* dimension includes SESSION.

Examples

Example 10–19 Creating an INTEGER Dimension Surrogate

The following statement creates an INTEGER type dimension surrogate for the `store_id` dimension.

```
DEFINE storepos SURROGATE store_id INTEGER
```

Example 10–20 Creating a NUMBER Dimension Surrogate

The following statement creates an NUMBER type dimension surrogate for the `product` dimension, which is a TEXT dimension that has product names as values. The *precision* argument to the NUMBER keyword specifies that a value in `prodnum` can have no more than seven characters and the scale argument specifies that no more than three characters can be to the right of the decimal point.

```
DEFINE prodnum SURROGATE product NUMBER(7, 3)
```

The following statement sets the first value of `prodnum` to `1083.375`.

```
prodnum(product 1) = 1083.375
```

DEFINE VALUESET

The DEFINE command with the VALUESET keyword adds a new valueset object to an analytic workspace. A valueset contains a list of dimension values for a dimension. The values in a valueset can be saved across sessions. When you begin a new session or start up a workspace, each dimension has all values in the status. You can then limit a dimension to the values stored in the valueset for that dimension.

Note: When you first define a valueset, its value is null. You must eplicitly assign values to the valueset as described in "Assigning Values to a Valueset" on page 10-65.

Syntax

DEFINE *name* VALUESET *dimension* [<*dims*...>] [TEMP] [AW *workspace*] [SESSION]

Arguments

name
The name of the object you are defining. For general information about this argument, see the main entry for the DEFINE command.

VALUESET
The object type when you are defining a valueset.

dimension
The name of the dimension whose values you want to store in the valueset.

dims
The names of the dimensions, if any, of the valueset. You must define a dimension before you include it in the definition of a valueset.

TEMP
Indicates that the valueset's values are only temporary. The valueset has a definition in the current workspace and can contain values during the current session. However, when you update and commit, only the definition of the valueset is saved. When you end the session or switch to another workspace, the values are discarded. Each time you start the workspace, the value of a temporary valueset is null.

AW *workspace*
The name of an attached workspace in which you wish to define the valueset. The valueset must be defined in the same workspace as its dimensions. For general information about this argument, see the main entry for the DEFINE command.

SESSION
Specifies that the object exists only in the current session. When the session ends, the object no longer exists. This differs from the TEMP keyword, which specifies that the values are temporary but the object definition remains in the workspace in which you create it.

Notes

Assigning Values to a Valueset

When you first define a valueset, its value is null. Use the LIMIT command to assign values to the valueset or to change its values. Use the STATUS command and functions such as STATFIRST, INSTAT, and VALUES to work with a valueset.

Examples

Example 10–21 **Creating and Assigning Values to a Valueset**

This example adds the valueset named `lineset` to the demonstration workspace. The `lineset` valueset is dimensioned by `line`, and therefore it can be limited by the current values of the `line` dimension. The LD command attaches a description to the object.

The statements

```
LIMIT line TO FIRST 2
STATUS line
```

produce the following output.

```
The current status of LINE is:
REVENUE, COGS
```

The statements

```
DEFINE lineset VALUESET line
LD Valueset for LINE dimension values
LIMIT lineset TO line
SHOW VALUES(lineset)
```

produce the following output.

```
Revenue
Cogs
```

Example 10–22   Creating and Assigning Values to a Multidmensional Valueset

Assume that your analytic workspace has the variables and dimensions with the following definitions.

```
DEFINE geography DIMENSION TEXT
DEFINE product DIMENSION TEXT
DEFINE sales VARIABLE DECIMAL <geography product>
DEFINE salestax VARIABLE DECIMAL <geography>
```

Assume also that the analytic workspace contains the following dimensions whose values are the names of variables and dimensions within the workspace.

```
DEFINE all_variables DIMENSION TEXT
MAINTAIN all_variables ADD 'sales' 'salestax'
DEFINE all_dims DIMENSION TEXT
MAINTAIN all_dims ADD 'geography' 'product'
```

The following statement creates a valueset for the values of `all_variables` and `all_dims`.

```
DEFINE variables_dims VALUESET all_dims <all_variables>
REPORT values(variables_dims)

ALL_VARIABLES        VALUES(VARIABLES_DIMS)
---------------- ------------------------------
sales            geography
                 product
salestax         geography
                 product
```

To create a multidimensional valueset that has the correct dimensions related to the variables that use them, you issue the following statement that uses a QDR to limit the `all_dims` values for the `salestax` value of `all_variables`.

```
LIMIT variables_dims(all_variables  'salestax') TO 'geography'
REPORT values(variables_dims)

ALL_VARIABLES        VALUES(VARIABLES_DIMS)
---------------- ------------------------------
sales            geography
                 product
salestax         geography
```

DEFINE VARIABLE

The DEFINE command with the VARIABLE keyword adds a new variable object to an analytic workspace. Variables store one type of data, which can be numeric, text, Boolean, or dates. Beside the data type of a variable, the definition that you create for a variable also determines the following characteristics of the variable:

- Dimensionality
- Permanent or temporary
- Unpartitioned or partitioned

You can also define local variables in programs using a VARIABLE statement. These variables exist only as long as the program is running.

Syntax

DEFINE *name* [<u>VARIABLE</u>] *datatype* [<*dims...*>] [<u>PERMANENT</u> | TEMP] -

[(*partition-instance...*)] [WIDTH *n*] [AW *workspace*] [SESSION]

where:

partition-instance has the following syntax.

PARTITION *partition-name* [{<u>INTERNAL</u> [TEMP | PERMANENT] } | {EXTERNAL *target*}]

Arguments

name
The name of the object you are defining. For general information about this argument, see the main entry for the DEFINE command.

VARIABLE
The object type when you are defining a variable. You do not have to include the word VARIABLE, because it is the default.

datatype
The kind of data to be stored in the variable. The data types, their abbreviations, and the range of acceptable values are shown in Table 10–1, " Valid Data Types for Variables".

Table 10–1 Valid Data Types for Variables

Data Type	Abbreviation	Values
INTEGER	INT	Whole numbers in the range of `(-2**31)` to `(2**31)-1`
LONGINTEGER	LONGINT	Whole numbers in the range of `(-2**63)` to `(2**63)-1`
SHORTINTEGER	SHORTINT	Whole numbers in the range of `(-2**15)` to `(2**15)-1`
NUMBER [(*p*,[*s*])]	None	Decimal numbers with up to 38 significant digits, as defined by the precision and scale where precision (*p*) is a whole number between 1 and 38 and scale (*s*) is a whole number between -84 and 127
DECIMAL	DEC	Decimal numbers with up to 15 significant digits
SHORTDECIMAL	SHORT	Decimal numbers with up to seven significant digits
TEXT	None	TEXT data type values (with no WIDTH setting) of one or more lines with no more than 4000 bytes for each line. Text values, with WIDTH set, of 1 line with no more than 4000 bytes (See "Values on Each Page" on page 10-75.) The TEXT data type corresponds to CHAR and VARCHAR2 data types in the Oracle relational database. TEXT characters are encoded in the database character set
NTEXT	None	NTEXT data type values with the same restrictions as TEXT data type values. The NTEXT data type corresponds to NCHAR and NVARCHAR2 data types in the Oracle relational database. However, an NTEXT character is always encoded in UTF8 Unicode. This encoding might be different from the NCHAR character set of the database, which can be UTF16. (See "Text Data Types" on page 2-3.)
ID	None	ID data type values of one line with no more than eight characters
BOOLEAN	BOOL	Logical `YES`/`NO` values (valid synonyms are ON/OFF and TRUE/FALSE)
DATE	None	Dates between Jan 1, 1000 A.D. and Dec 31, 9999 A.D.
DATETIME	None	Dates between Jan 1, 4712 B.C. and Dec 31, 9999 A.D., and times in hours, minutes, and seconds

dim

The dimensions of the variable. A dimension may be one of the following:

- A simple, concat, conjoint, or alias dimension that you have previously defined using a DEFINE DIMENSION statement. In this case, you specify the name of the dimension.

> **Note:** The order in which you list the *dims* of a variable is the default order of the dimensions and behavior of a variety of statements (such as REPORT, and UNRAVEL) and effects how the data for the variable is stored (as discussed in "Effect of Dimension Order on Variable Storage" on page 10-74. Also, When you define more than one object with the same dimensions, most operations will work much more efficiently when you list the dimensions in the same order in each definition.

- A partition template object that you have previously defined using a DEFINE PARTITION TEMPLATE statement. In this case, you specify the value using the following syntax.

  ```
  <partition-template-name<dims>>
  ```

  The dimensions that you specify for *dims* must be the same dimensions as those of *partition_template*.

- A named or unnamed composite. In this case, you specify the value using the following syntax.

  ```
  {SPARSE|composite-name} <sparsedims...>
  ```

  where:

  - **SPARSE** indicates that you want Oracle OLAP to create an unnamed composite and use it when dimensioning the variable. For a discussion of unnamed composites, see "Unnamed Composites" on page 10-24.
  - ***composite-name*** is the name of a named composite previously defined using a DEFINE COMPOSITE statement.
  - ***sparsedims*** are the names, separated by commas, of the dimensions for which the named or unnamed composite is created. You must enclose the names of the dimensions in a single set of angle brackets (< >).

PERMANENT
TEMP
Specifies that a variable or a partition of a variable is either permanent or temporary. After you update and commit, the definition of both permanent and temporary variables and partitions is always saved between sessions. Specifying permanent or temporary determines whether or not the values of a variable or partition of a variable are saved or discarded, after you update and commit, when you leave end your session or switch to another workspace:

- Permanent variables and partitions—Oracle OLAP saves the data values or a permanent variable or permanent partitions. When you start the workspace, the data values or a permanent variable or permanent partitions are the same as they were at the last commit.
- Temporary variables and partitions—Oracle OLAP discards the data values of a temporary variable or temporary partition. Each time you start the workspace, the values of a temporary variable or temporary partition are `NA`.

Keep the following points in mind when specifying the PERMANENT and TEMP keywords:

- By default, a variable is permanent.
- Temporary variables can be dimensioned by partition template objects or by temporary dimensions.
- External partitions of a variable have the permanence of the variable that they represent.
- By default, a top-level internal partition of a variable has the same permanence as the variable that contains it. Specifically, an internal partition of a temporary variable is a temporary partition unless you use the PEMANENT keyword to make it a permanent partition, and an internal partition of a permanent variable is a permanent partition unless you use the TEMPORARY keyword to make it a temporary partition. To indicate different behavior, use either the PERMANENT or TEMP keyword.
- By default, an internal subpartition has the same permanence as its parent partition. To indicate different behavior, use either the PERMANENT or TEMP keyword.

WIDTH *n*
(You can abbreviate WIDTH as W.) The width, in bytes, of the storage area for each value of a variable. When you are using a multibyte character set, be sure to specify the number of bytes, not characters.

You specify fixed widths to create faster and more compact data storage formats. You can specify fixed widths for dimensioned TEXT, NTEXT, and INTEGER variables only, as described in the following list:

- For dimensioned TEXT and NTEXT variables, you can specify a width from 1 byte through 4000 bytes. Specify a fixed width for such variables only when you are certain that the values of a particular variable are of similar size. You cannot assign a width to a scalar variable.
- For dimensioned INTEGER variables, you can specify a width of 1 byte only. Define a fixed width INTEGER variable only when you are certain that all the values for that variable are between -128 and 127.

The default widths for variables are as follows: 2 bytes for SHORTINTEGER, 4 bytes for DATE, INTEGER, and SHORTDECIMAL, and 8 bytes for DECIMAL and ID. TEXT and NTEXT variables that do not have fixed widths are stored on two sets of pages. The first set contains 4-byte cells, each of which points to the actual text value that is stored in the other set of pages. The default width of 4 bytes for TEXT and NTEXT variables is for these 4-byte cells.

partition-name
The name of the partition.

INTERNAL
(Default) Indicates that this partition is not a previously defined variable.

EXTERNAL
Indicates that this partition is a previously defined variable that is a base dimension of concat dimension by which the variable is partitioned.

> **Note:** You can only use this keyword when variable is dimensioned by a partition template object that was defined using a DEFINE PARTITION TEMPLATE statement that included the PARTITION BY CONCAT clause.

target
The name of the variable that is the external partition.

AW *workspace*
The name of an attached workspace in which you wish to define the variable. When the variable is dimensioned, it must be defined in the same workspace as its

dimensions. For general information about this argument, see the main entry for the DEFINE command.

SESSION
Specifies that the object exists only in the current session. When the session ends, the object no longer exists. This differs from the TEMP keyword, which specifies that the values are temporary but the object definition remains in the workspace in which you create it.

Notes

Variable Size

Theoretically, a variable can contain up to `2**63` cells and a TEXT or NTEXT variable can contain up to 2 billion bytes. However, certain considerations apply when defining large variables, as described in "Values on Each Page" on page 10-75.

Variable Properties

As with other types of objects, you can set properties on variables with the PROPERTY command. For dimensioned variables the $NATRIGGER and $STORETRIGGERVAL properties have special meaning.

How Variable Data is Stored

How variable data is stored in an analytic workspace is determined by the following:

- Whether or not the variable is dimensioned by a composite or a conjoint dimension.
- The order in which the dimensions for the variable are defined.
- Whether or not the variable is defined as a partitioned variable.
- Whether or not the variable is stored in segments of default size or of an explicit size.

Effect of Composites and Conjoint Dimensions on Variable Storage When a variable is dimensioned by regular dimensions, Oracle OLAP creates a cell in the variable for each set of its dimension values. When a cell is empty, then the cell is said to contain an `NA` value. In some cases, this can result in a sparse variable—that is, a variable in which a relatively high percentage of cells are empty. There are two types of sparsity:

- **Controlled sparsity** occurs when a range of one or more dimensions has no data; for example, a new variable dimensioned by `month` for which you do not have data for past months.
- **Random sparsity** occurs when some combinations of dimension values never have any data. For example, a district might only sell certain products and never have data for other products. Other districts might sell some of those products and other ones, too.

You can reduce the number of empty cells by dimensioning a variable with one or more composites or conjoint dimensions. Composites and conjoint dimensions are lists of dimension value combinations. For variables dimensioned by these objects Oracle OLAP does not create a variable cell for every value in the base dimensions. Instead, it creates cells only for those dimension values that are stored in the list of dimension value combinations of the composite or conjoint dimension. See DEFINE COMPOSITE and DEFINE DIMENSION (conjoint) for more information.

> **Note:** Special considerations apply when you dimension a variable by a compressed composite, see "Defining Variables with Compressed Composites" on page 10-75 for more information.

Effect of Dimension Order on Variable Storage The order in which you list the dimensions in an unpartitioned variable definition determines how the data of that variable is stored and accessed. The first dimension in the variable definition is the **fastest-varying dimension**, and the last dimension is the **slowest-varying dimension**.

For example, assume your analytic workspace has an `opcosts` variable that contains the operating costs, by month, of each city in which you have offices. In the following definition for the `opcosts` variable, `month` is the fastest-varying dimension and `city` is the slowest-varying dimension.

```
DEFINE opcosts VARIABLE DECIMAL <month city>
```

The data for a multidimensional variable is stored as a linear stream of values, in which the values of the fastest-varying dimension are clustered together. For example, for the `opcosts` variable, the values for Boston for all the months are stored in a sequence, and then it stores the values for Chicago for all the months in a sequence, and so on. Thus the month values vary fastest in the `opcosts` variable, as shown in the following table.

When you define variables and other dimensioned objects, and when you write programs that loop over multidimensional expressions in nested loops, you should always try to maximize performance by matching the fastest-varying dimension with the inner loop.

Effect of Partitioning on Storage Each unpartitioned data object is a single row in the table that is an analytic workspace. Variables defined as partitioned variables are stored as multiple rows in the table. Each partition is a single row. Within a partition, the way that the variable's data is stored is determined by the order in which the dimensions for the variable are defined and the type of segments used by the variable.

Effect of Segment Type on Storage Within a partition, variable data is stored in segments. A segment is continuous disk space. By default, the segment sizes of a variable are automatically determined by Oracle OLAP. Each segment is the exactly the amount of continuous disk space needed to store all of the values assigned by a single OLAP DML statement.You can explicitly specify a segment size for a variable using the SEGWIDTH keyword of the CHGDFN command. In this case, when you assign values to a variable, Oracle OLAP stores the data assigned by multiple OLAP DML statements into a segment until the segment is full.

Values on Each Page Pages are the units of storage in a analytic workspace. To calculate the maximum number of values for a variable of a given width that will fit on one page, use the VALSPERPAGE program.

Defining Variables with Compressed Composites Keep the following points in mind when defining a variable that is dimensioned by a compressed composite:

- A compressed composite can dimension only one variable or one partition of a variable. A compressed composite cannot be a shared composite.
- The compressed composite must be the last dimension in the variable's dimension list of the DEFINE VARIABLE statement that defines the variable.
- The partitions of a variable dimensioned by a compressed composite must respect the parent-child relationships of the hierarchical dimensions. When an AGGREGATE command executes, data cannot be aggregated across partitions. To check to see if a variable is partitioned correctly, use the PARTITIONCHECK function.

Examples

Example 10–23 Defining a Variable

This example adds the variable `population` to a workspace. It is dimensioned by `city`, which has already been defined in the workspace. The LD Command attaches a description to the object. The statements

```
DEFINE population INTEGER <city>
LD Population in each city
DESCRIBE population
```

produce the following description.

```
DEFINE POPULATION VARIABLE INTEGER <CITY>
LD Population in each city
```

Example 10–24 Defining a Single-Cell Variable

The following is a definition for a variable named `newdata` which is a single Boolean value. It has no dimensions. An application might set it to `YES` when new data is added to the workspace and to `NO` after a user views the data.

```
DEFINE newdata BOOLEAN
newdata = YES
```

Example 10–25 Defining NUMBER Variables

The following statement defines a NUMBER variable named `sales` and dimensioned by `product` and `geography` with a precision of 16 digits and a scale of 4 digits.

```
DEFINE sales VARIABLE NUMBER (16,4) <product, geography>
```

The following statements define a NUMBER variable named `numvar` with 5 significant digits and 2 decimal places. The number 1234567 is out of its range.

```
DEFINE numvar VARIABLE NUMBER (5, 2)
numvar = 1234567
SHOW numvar
NA
```

A negative scale defines a NUMBER variable named `numnegvar` with 5 significant digits and 2 rounded digits to the left of the decimal point. The number 1,234,567 is rounded up.

```
DEFINE numnegvar VARIABLE NUMBER (5, -2)
numnegvar = 1234567
SHOW numnegvar
1,234,600.00
```

Example 10–26   Defining a Variable with Internal Partitions

Assume that you want to define a `sales` variable that is dimensioned by product and time and that is partitioned so that each year's detail (day) data is in a separate partition and the summary (month and year) data is in yet another partition.

Assume that the analytic workspace contains a `products` dimension, a `time` dimension that is a simple hierarchical dimension with three levels of data (day, month, and year), and a `time_parentrel` relation that represents the child-parent relationships between the values of time.

```
DEFINE TIME DIMENSION TEXT
DEFINE PRODUCT DIMENSION TEXT
DEFINE TIME_PARENTREL RELATION TIME <TIME>
```

For simplicity's sake, in this example the `time` and `product` dimensions are only partially populated and have only the following values.

```
TIME
--------------
2003
2002
Dec2003
Jan2003
Dec2002
Jan2002
31Dec2003
01Dec2003
31Jan2003
01Jan2003
31Dec2002
01Dec2002
31Jan2002
01Jan2002

PRODUCT
-------
00001
00002
```

To create the partitioned variable, take the following steps:

1. Define a partition template that defines one partition for each year's data.

   ```
   DEFINE PARTITION_SALES_BY_YEAR PARTITION TEMPLATE <TIME PRODUCT> -
       PARTITION BY LIST (TIME) -
       (PARTITION TIME_2003 VALUES -
   ('2003','Dec2003','Jan2003', 31Dec2003',01Dec2003','31Jan2003','01Jan2003')-
         PARTITION TIME_2002 VALUES -
   ('2002','Dec2002','Jan2002', 31Dec2002',01Dec2002','31Jan2002','01Jan2002'))
   ```

   (note that for simplicity's sake, only some of each year's dimension values are specified for each partition in this example. Typically, when you want to specify a large number of values for a partition, you do not do so within the DEFINE PARTITION STATEMENT statement. Instead, you define the partition without specifying any values, and then later specify the values using MAINTAIN ADD TO PARTITION or MAINTAIN MOVE TO PARTITION statements as illustrated in Example 16–50, "Specifying the Values of a Partition Using Valuesets" on page 16-108.)

2. Define a partitioned `sales` variable with the partitions defined by the partition template named `partition_sales_by_year`.

```
DEFINE sales DECIMAL <partition_sales_by_year<time product>>
```

3. After you populate sales with day values, you can issue the following REPORT statement to see which `sales` values are in which partition.

```
REPORT DOWN PARTITION(partition_sales_by_year) time product sales

PARTITION(PARTITION_SALES_BY_YEAR)     TIME     PRODUCT     SALES
---------------------------------- ---------- ---------- ----------
TIME_2003                          2003       00001              NA
TIME_2003                          Dec2003    00001              NA
TIME_2003                          Jan2003    00001              NA
TIME_2003                          31Dec2003  00001           14.78
TIME_2003                          01Dec2003  00001           15.52
TIME_2003                          31Jan2003  00001           13.61
TIME_2003                          01Jan2003  00001           10.39
TIME_2003                          2003       00002              NA
TIME_2003                          Dec2003    00002              NA
TIME_2003                          Jan2003    00002              NA
TIME_2003                          31Dec2003  00002           16.05
TIME_2003                          01Dec2003  00002           12.27
TIME_2003                          31Jan2003  00002           10.83
TIME_2003                          01Jan2003  00002           11.07
TIME_2002                          2002       00001              NA
TIME_2002                          Dec2002    00001              NA
TIME_2002                          Jan2002    00001              NA
TIME_2002                          31Dec2002  00001           18.80
TIME_2002                          01Dec2002  00001           13.64
TIME_2002                          31Jan2002  00001           12.41
TIME_2002                          01Jan2002  00001           16.97
TIME_2002                          2002       00002              NA
TIME_2002                          Dec2002    00002              NA
TIME_2002                          Jan2002    00002              NA
TIME_2002                          31Dec2002  00002           17.47
TIME_2002                          01Dec2002  00002           16.58
TIME_2002                          31Jan2002  00002           18.94
TIME_2002                          01Jan2002  00002           18.36
```

Example 10–27 Defining a Variable with External Partitions

Assume you have an analytic workspace that contains individual sales variables for each years data.

```
DEFINE year_2003 DIMENSION TEXT
DEFINE year_2002 DIMENSION TEXT
DEFINE year_2003_PARENTREL RELATION year_2003 <year_2003>
DEFINE year_2002_PARENTREL RELATION year_2002 <year_2002>
DEFINE sales_2003 VARIABLE DECIMAL <year_2003 product>
DEFINE sales_2002 VARIABLE DECIMAL <year_2002 product>
```

Assume also that you want to logically combine the sales data into a single variable that has sales data for all years. To do this you add the following definitions to the analytic workspace:

- A definition for a concat dimension that has the time-related dimensions of sales_2002 and sales_2003 as base dimensions.

  ```
  DEFINE time DIMENSION CONCAT (year_2003 Year_2002) UNIQUE
  ```

- A definition for the relation that specifies the child-parent relationship of the values of the time hierarchy.

  ```
  DEFINE time_parentrel RELATION time <time>
  ```

- A partition template object that defines the partitions for each year's sales data (that is, sales_2002 and sales_2003).

  ```
  DEFINE part_temp_sales_by_year PARTITION TEMPLATE <time product> -
     PARTITION BY CONCAT (time)-
    (PARTITION partition_2002 <year_2002 product>, -
    PARTITION partition_2003 <year_2003 product>)
  ```

- A sales variable with external partitions for sales_2002 and sales_2003.

  ```
  DEFINE sales DECIMAL <part_temp_sales_by_year<time product>> -
   (PARTITION partition_2002 EXTERNAL sales_2002,-
    PARTITION partition_2003 EXTERNAL sales_2003)
  ```

When you issue the following REPORT statement you can see the values in the partitions of sales.

```
REPORT DOWN PARTITION(part_temp_sales_by_year) time product sales

PARTITION(PART_TEMP_SALES_BY_YEAR)     TIME     PRODUCT     SALES
----------------------------------- ---------- ---------- ----------
PARTITION_2002                      01Jan2002  00001           14.44
PARTITION_2002                      31Jan2002  00001           15.55
PARTITION_2002                      01Dec2002  00001           11.39
PARTITION_2002                      31Dec2002  00001           10.53
PARTITION_2002                      Jan2002    00001           29.99
PARTITION_2002                      Dec2002    00001           21.92
PARTITION_2002                      2002       00001           51.91
PARTITION_2002                      01Jan2002  00002           11.03
PARTITION_2002                      31Jan2002  00002           12.20
PARTITION_2002                      01Dec2002  00002           12.80
PARTITION_2002                      31Dec2002  00002           13.77
PARTITION_2002                      Jan2002    00002           23.23
PARTITION_2002                      Dec2002    00002           26.57
PARTITION_2002                      2002       00002           49.80
PARTITION_2003                      01Jan2003  00001           10.00
PARTITION_2003                      31Jan2003  00001           10.88
PARTITION_2003                      01Dec2003  00001              NA
PARTITION_2003                      31Dec2003  00001              NA
PARTITION_2003                      Jan2003    00001           20.88
PARTITION_2003                      Dec2003    00001              NA
PARTITION_2003                      2003       00001              NA
PARTITION_2003                      01Jan2003  00002           15.21
PARTITION_2003                      31Jan2003  00002           13.37
PARTITION_2003                      01Dec2003  00002              NA
PARTITION_2003                      31Dec2003  00002              NA
PARTITION_2003                      Jan2003    00002           28.58
PARTITION_2003                      Dec2003    00002              NA
PARTITION_2003                      2003       00002              NA
```

Example 10–28 Defining a Fixed-Width TEXT Variable

The following statement defines a TEXT variable named `lastname` dimensioned by `employee`. Values in `lastname` are limited to 20 characters, so that longer values are truncated.

```
DEFINE lastname TEXT <employee> WIDTH 20
```

Example 10–29 Defining a Variable That Uses a Named B-Tree Composite

Assume that you have the following dimensions in your analytic workspace.

```
DEFINE month DIMENSION TEXT
DEFINE product DIMENSION TEXT
DEFINE region DIMENSION TEXT
```

When your company does promotional marketing for certain products in some but not all regions, then your variable data will be sparse along the `product` and `region` dimensions. Therefore, suppose you define a composite named `proddist`, whose base dimensions are `product` and `region`. There are dimension-value combinations in the composite only for those values that have data. For example, when you run a promotion for tents but not skis, then the composite includes the tents and region combinations, but not the skis and region combinations.

The following statement creates a b-tree composite named `proddist` whose base dimensions are `product` and `district`, and a variable called `promo` that is dimensioned by `month` and `proddist`.

```
DEFINE proddist COMPOSITE <product region>
DEFINE promo VARIABLE INTEGER <month proddist <product district>>
```

For simplicity's sake assume that you have only stored the following dimension data in your analytic workspace.

```
PRODUCT
--------------
Tents
Skis

REGION
--------------
Northeast
Southwest

MONTH
--------------
Jan2003
Feb2003
Mar2003
Apr2003
May2003
Jun2003
Jul2003
Aug2003
Sep2003
Oct2003
Nov2003
Dec2003
```

You decide to run a promotional sales for skis in the Northeast region in the month of September, 2003 at a cost of $5,000. Once you populate `promo` with this, `promo` contains only 12 cells—each cell is dimensioned by a value of `month` and the composite tuple value of `<'Skis' 'Northeast'>` for `proddist`. The cell for September 2003 contains the value $5,000, and all of the other cells contain `NA`. No other `NA` values are stored in `promo`; there are no cells are created for any other values of `product` or `region`.

DEFINE WORKSHEET

The DEFINE command with the WORKSHEET keyword adds a new worksheet object to an analytic workspace. A worksheet, like a spreadsheet, is a two-dimensional object that is dimensioned by a worksheet row and a worksheet column. It can temporarily store data that you want to transfer between spreadsheet packages and workspace dimensions and variables.

Syntax

DEFINE *name* WORKSHEET [<*column-dim row-dim*>] [TEMP] [AW *workspace*] [SESSION]

Arguments

name
The name of the object you are defining. For general information about this argument, see the main entry for the DEFINE command.

WORKSHEET
The object type when you are defining a worksheet.

<column-dim row-dim>
The names of the dimensions of the worksheet. When you supply this argument, you must give the names of two integer dimensions for *column-dim* and *row-dim*. When you omit this argument, the worksheet will be dimensioned automatically by `WKSCOL` and `WKSROW`. See "Worksheet Dimensions" on page 10-85 for more information

TEMP
Indicates that the worksheet is only temporary. The worksheet is defined in the specified workspace and can contain values during the current session. However, when you update and commit, only the definition of the worksheet is saved. When you end your session or switch to another workspace, the data values are discarded.

AW *workspace*
The name of an attached workspace in which you wish to define the worksheet. The worksheet must be defined in the same workspace as its dimensions. For general information about this argument, see the main entry for the DEFINE command.

SESSION

Specifies that the object exists only in the current session. When the session ends, the object no longer exists. This differs from the TEMP keyword, which specifies that the values are temporary but the object definition remains in the workspace in which you create it.

Notes

Worksheet Dimensions

A worksheet is always dimensioned by two dimensions that represent a worksheet row and a worksheet column. The worksheet row and a worksheet column dimensions can either be automatically created by Oracle OLAP or explicitly created by you.

- When you have not created worksheet row and a worksheet column dimensions and specified their names in the *column-dim* and *row-dim*t arguments of DEFINE WORKSHEET, Oracle OLAP automatically creates the following dimensions:
  - For the worksheet row, an INTEGER dimension named `WKSROW` with values from 1 to 63.
  - For the worksheet column, an INTEGER dimension named `WKSROW` with values from 1 to 63.

  > **Note:** When `WKSCOL` and `WKSROW` already exist in any attached workspace, Oracle OLAP cannot create them in the current worksheet. In this case, the DEFINE WORKSHEET command will fail to create a worksheet with these default dimensions.

  `WKSCOL` and `WKSROW` do not appear in the worksheet description. (For information on how to create a worksheet description, see DESCRIBE.)

- You create worksheet row and a worksheet column dimensions the same way you create any other simple dimension by issuing the following statements:
  1. Create two simple INTEGER dimensions using a DEFINE DIMENSION (simple) statement.
  2. Populate the dimensions with the number of rows and columns that you want in your worksheet using a MAINTAIN statement.

Adding Worksheet Cells Automatically

When you import a file that requires more cells than are available, the worksheet dimensions are maintained automatically. For this reason, you should avoid using the worksheet dimensions for other types of objects.

You can also add or delete values from worksheet row and a worksheet column dimensions with the MAINTAIN command, which changes the number of cells in the worksheet.

Examples

Example 10–30 Defining a Worksheet

These statements define a temporary worksheet named `travelexp`, which is dimensioned by `columns` and `rows`.

```
DEFINE itemsheet WORKSHEET
DEFINE columns INT DIMENSION
MAINTAIN columns ADD 5
DEFINE rows INT DIMENSION
MAINTAIN rows ADD 10
DEFINE travelexp WORKSHEET <columns rows> TEMPORARY
```

Example 10–31 Importing Spreadsheet Data

You can import data from a spreadsheet to a worksheet. When all the cells contain the same type of data, you can use UNRAVEL to transfer the data to a variable with one statement. You can also limit the worksheet dimensions to a smaller group of cells and use UNRAVEL to transfer each group to a separate variable. To transfer imported data from a worksheet named `itemsheet` to a variable named `items`, you might use the following statements.

```
DEFINE itemsheet WORKSHEET
IMPORT itemsheet FROM dif FILE 'file name'
LIMIT WKSCOL TO FIRST 3
LIMIT WKSROW TO FIRST 10
items = UNRAVEL(itemsheet)
```

DELETE

The DELETE command deletes one or more objects from a workspace. The deletion becomes permanent when you execute the UPDATE and COMMIT commands.

Syntax

DELETE *name*... [AW *workspace*]

Arguments

name...

The names of one or more objects, separated by spaces or commas. DELETE removes the definitions of these objects from the appropriate workspace.

You can specify a qualified object name to indicate the attached workspace in which each object can be found. In this way, you can specify different workspaces for different objects that you are deleting. As an alternative, you can use the AW argument to specify the workspace in which all of the objects can be found. Do not use both qualified object names and the AW argument in the same DELETE command.

When you do not use a qualified object name or the AW argument to specify a workspace, objects are deleted in the current workspace.

Note: Oracle OLAP does *not* warn you when you delete an object that has the same name as an existing object in another attached workspace.

AW *workspace*

The name of an attached workspace in which you wish to delete all the specified objects. When you do not use a qualified object name or the AW argument to specify a workspace, objects are deleted in the current workspace.

Notes

Deleting Associated Objects

To delete a dimension or a composite, you must first delete all the objects that are dimensioned by it and any objects that use it. Since a dimension surrogate cannot dimension other objects, you can delete a dimension surrogate at any time. Deleting a surrogate does not affect the dimension for which it is a surrogate.

Status of NAME

When you use the DELETE command when the NAME dimension is limited to less than all its values, DELETE automatically sets the status of NAME to ALL.

Renaming Before Deleting

When you see an error message when you try to delete an object, then the name of that object might be a reserved word. (Use RESERVED to identify reserved words.) When this is the case, use the RENAME command to give the object a new name, and then delete it.

Deleting and PERMIT

You cannot delete an object when a PERMIT command denies you the right to change its permission.

Examples

Example 10–32 Deleting a Dimension

Suppose you have a dimension named `city` and a variable named `population` that you want to delete. The variable `population` is the only object that is dimensioned by or makes use of `city`, so you can delete them both together when you place the variable before the dimension in the DELETE command.

```
DELETE population city
```

Placing `city` before `population` in the preceding statement would produce an error.

11 DEPRDECL to EXISTS

This chapter contains the following OLAP DML statements:

DEPRDECL

The DEPRDECL function calculates the depreciation expenses for a series of assets. DEPRDECL uses the declining balance method to depreciate the assets over the specified lifetime of the assets. The starting value and ending value are specified for the assets acquired in each time period.

Return Value

DECIMAL

Syntax

DEPRDECL(*start-exp end-exp n* [*decline-factor* [{FULL|HALF|*portion-exp*} [*time-dimension*]]])

Arguments

start-exp
A numeric expression that contains the starting values of the assets. The *start-exp* expression must be dimensioned by a time dimension. For each value of the time dimension, *start-exp* contains the initial value of the assets acquired during that time period. In addition to a time dimension, *start-exp* can also have non-time dimensions.

end-exp
A numeric expression that contains the ending values of the assets. The *end-exp* expression must be dimensioned by the same dimensions as *start-exp.* For each value of the time dimension, *end-exp* contains the final (or salvage) value for the assets acquired during that time period. Each value of *start-exp* must have a corresponding *end-exp* value. For example, when the assets acquired in 1996 have a salvage value of `$200`, then the value of *end-exp* for 1996 is `$200`.

n
An integer expression that contains the number of periods for the depreciation life of the assets. The *n* expression can have any of the non-time dimensions of *start-exp,* but it cannot have a time dimension.

decline-factor
A numeric expression that gives the declining balance rate to use for calculating the depreciation expenses. The *decline-factor* expression can have any of the non-time dimensions of *start-exp,* but it cannot have a time dimension.

A factor of 2 indicates a double declining balance. The default is `2`.

FULL
Specifies that the full amount of a time period's depreciation expense is charged to the time period in which assets were acquired. Charges the full amount to all of the assets in the series. (Default)

HALF
Specifies that half of the full amount of a time period's depreciation expense is charged to the time period in which assets were acquired. Charges half the full amount to all of the assets in the series. You might want to use HALF when assets are acquired during the second half of the time period.

portion-exp
When you want to charge the full amount for some assets and half the amount for other assets, you can supply a *portion-exp* expression that is dimensioned by any of the non-time dimensions of *start-exp.* The *portion-exp* expression must be a text expression with values of FULL or HALF.

time-dimension
The name of the time dimension by which *start-exp* and *end-exp* are dimensioned. When the time dimension has a type of DAY, WEEK, MONTH, QUARTER, or YEAR, the *time-dimension* argument is optional.

Notes

How Result Is Dimensioned

The result returned by DEPRDECL is dimensioned by all the dimensions of *start-exp.*

Using Optional Arguments

When you include either of the two final optional arguments (FULL or HALF or *portion* and *time-dimension*), you must also include the preceding optional arguments.

Depreciation for a Given Time Period

DEPRDECL calculates the depreciation expense for a given time period as the sum of that period's depreciation expenses for all assets in the series that are not yet fully depreciated. The first period of depreciation for an asset is the period in which it was acquired.

The HALF Argument

When you specify HALF as the portion of depreciation expenses to charge to the period of acquisition, the HALF factor is applied to each period. Half of each period's full depreciation is rolled to the next period, and the final half period of depreciation takes place in the time period `n + 1`.

Calculation Method Used

For each time period, DEPRDECL calculates the declining balance depreciation expense by multiplying the current value of an asset by the *decline-factor* and dividing the result by the number of periods in the lifetime of an asset. However, when the calculation for a specific time period results in an asset's current value going below the ending value, then the depreciation expense is adjusted. In this instance, the depreciation expense is calculated as the current value minus the ending value.

Low Ending Value

When the ending value specified for an asset is low enough that the depreciation expense for the last period does not need to be adjusted, then the total depreciation expense over all the periods will usually be less than the starting value minus the specified ending value.

High Ending Value

When the ending value specified for an asset is relatively high, then an asset might be totally depreciated in fewer periods than were specified for the lifetime of the depreciation. In this instance, when you want the depreciation expense applied across the specified lifetime of the depreciation, you can lower the decline-factor.

NA Mismatch Errors

When a value of *start-exp* is `NA` and the corresponding value of *end-exp* is not `NA`, an error occurs. Similarly, when a value of *end-exp* is `NA` and the corresponding value of *start-exp* is not `NA`, an error occurs.

NASKIP Option Settings

DEPRDECL is affected by the NASKIP option when a value of *start-exp* and the corresponding value of *end-exp* are both `NA`. When NASKIP is `YES` (the default), DEPRDECL treats the values as zeros when calculating the depreciation expenses. When NASKIP is `NO`, DEPRDECL returns `NA` for all affected time periods.

Dimension Limits Ignored

The DEPRDECL calculation begins with the first time dimension value, regardless of how the status of that dimension may be limited. For example, suppose *start-exp* is dimensioned by `year`, and the values of `year` range from `Yr95` to `Yr99`. The calculation always begins with `Yr95`, even when you limit the status of `year` so that it does not include `Yr95`.

Another Declining-Balance Method

The pure declining-balance method of depreciation used by DEPRDECL is not the most widely used form of the declining-balance method. For a more commonly used form of the declining-balance method, see the DEPRDECLSW function, which uses a combination of the declining-balance and straight-line methods.

Related Functions

The DEPRSL function, which uses the straight-line method to calculate depreciation expenses, and the DEPRSOYD function, which uses the sum-of-years'-digits method to calculate depreciation expenses.

Examples

Example 11–1 Using DEPRDECL to Calculate Depreciation Expenses for Assets Acquired in a Single Period

This example shows how to use DEPRDECL to calculate depreciation expenses for assets acquired in a single time period.

The following statements create two variables called `assets` and `salvage`.

```
DEFINE assets DECIMAL <year>
DEFINE salvage DECIMAL <year>
```

Suppose you assign the following values to the variables `assets` and `salvage`.

```
YEAR             ASSETS    SALVAGE
------------- ---------- ----------
Yr95            1,000.00     100.00
Yr96                0.00       0.00
Yr97                0.00       0.00
Yr98                0.00       0.00
Yr99                0.00       0.00
Yr00                0.00       0.00
```

The `assets` variable contains the starting value of the assets acquired in 1995. The `salvage` variable contains the ending value of the assets acquired in 1995.

The following statement reports asset and salvage values, along with depreciation expenses for the assets. Note that the call to DEPRDECL to calculate the depreciation expenses specifies an asset lifetime of 5 periods (in this case, years) and a decline factor of 2 (double-declining balance).

```
REPORT assets salvage W 12 HEADING 'Depreciation' -
   DEPRDECL(assets salvage 5 2 FULL year)
```

This statement produces the following output.

```
YEAR             ASSETS    SALVAGE   Depreciation
------------- ---------- ---------- ------------
Yr95            1,000.00     100.00       400.00
Yr96                0.00       0.00       240.00
Yr97                0.00       0.00       144.00
Yr98                0.00       0.00        86.40
Yr99                0.00       0.00        29.60
Yr00                0.00       0.00         0.00
```

In this example, the depreciation expense for 1999 is adjusted so that the current asset value does not fall below the salvage value. The current asset value is calculated by subtracting the accumulated depreciation expense from the starting asset value. For example, for 1998 the accumulated depreciation expense is $870.40 ($400.00 + $240.00 + $144.00 + $86.40 = $870.40). This means the current asset value for 1998 is $129.60 ($1,000.00 - $870.40 = $129.60). In this example, the depreciation expense is usually calculated by multiplying the current asset value by 2 and then dividing the result by 5. Now, if $129.60 is multiplied by 2, then divided by 5, the resulting depreciation expense is $51.84. If this depreciation expense is subtracted from the 1998 current asset value of $129.60, the current asset value for 1999 would be $77.76, which is below the salvage value of $100. Instead of letting the current asset value fall below the salvage value, the DEPRDECL function subtracts the salvage value ($100.00) from the current asset value ($129.60) to calculate the depreciation expense ($29.60).

Example 11–2 Using DEPRDECL to Calculate the Depreciation Expenses for Assets Acquired in Multiple Periods

You can also use DEPRDECL to calculate the depreciation expenses for a series of assets.

Suppose you change the values for the year 1997 in the variables `assets` and `salvage` to the values shown in the following report.

```
YEAR              ASSETS    SALVAGE
-------------- ---------- ----------
Yr95             1,000.00     100.00
Yr96                 0.00       0.00
Yr97               500.00      50.00
Yr98                 0.00       0.00
Yr99                 0.00       0.00
Yr00                 0.00       0.00
Yr01                 0.00       0.00
Yr02                 0.00       0.00
```

Now `assets` and `salvage` contain nonzero values for 1995 and for 1997

The following statement reports the values of assets and salvage, and uses DEPRDECL to calculate depreciation expenses for each year, specifying an asset lifetime of 5 years, and a decline factor of 2 (double declining balance).

```
REPORT assets SALVAGE W 12 HEADING 'Depreciation'  -
   DEPRDECL(assets salvage 5 2 FULL year)
```

This statement produces the following output. (Notice that the depreciation expense increases in 1997 due to the assets acquired in that year.)

```
YEAR              ASSETS    SALVAGE   Depreciation
-------------- ---------- ---------- ------------
Yr95             1,000.00     100.00       400.00
Yr96                 0.00       0.00       240.00
Yr97               500.00      50.00       344.00
Yr98                 0.00       0.00       206.00
Yr99                 0.00       0.00       101.00
Yr00                 0.00       0.00        43.20
Yr01                 0.00       0.00        14.80
Yr02                 0.00       0.00         0.00
```

DEPRDECLSW

The DEPRDECLSW function calculates the depreciation expenses for a series of assets. DEPRDECLSW uses a variation on the declining balance method to depreciate assets over the specified lifetime of the assets. DEPRDECLSW begins by using the declining balance method, then switches over to the straight-line method at one of the following points in the time series:

- The first period for which straight-line depreciation over the remaining periods exceeds the declining balance depreciation for those periods (the default)
- The period specified by the *switch-period* argument

Return Value

DECIMAL

Syntax

DEPRDECLSW(*start-exp end-exp n*

[*decline-factor* [{FULL|HALF| *portion-exp*} [*switch-period* [*time-dimension*]]]])

Arguments

start-exp
A numeric expression that contains the starting values of the assets. The *start-exp* expression must be dimensioned by a time dimension. For each value of the time dimension, *start-exp* contains the initial value of the assets acquired during that time period. In addition to a time dimension, *start-exp* can also have non-time dimensions.

end-exp
A numeric expression that contains the ending value of the assets. The *end-exp* expression must be dimensioned by the same dimensions as *start-exp.* For each value of the time dimension, *end-exp* contains the final (or salvage) value for the assets acquired during that time period. Each value of *start-exp* must have a corresponding *end-exp* value. For example, when the assets acquired in 1990 have a salvage value of $200, then the value of *end-exp* for 1990 is $200.

n

An integer expression that contains the number of periods for the depreciation life of the assets. The *n* expression can have any of the non-time dimensions of *start-exp*, but it cannot have a time dimension.

decline-factor

A numeric expression that gives the declining balance rate to use for calculating the depreciation expenses. The *decline-factor* expression can have any of the non-time dimensions of *start-exp*, but it cannot have a time dimension.

A factor of 2 indicates a double declining balance. The default is 2.

FULL

Specifies that the full amount of a time period's depreciation expense is charged to the time period in which assets were acquired. Charges the full amount to all of the assets in the series. (Default)

HALF

Specifies that half of the full amount of a time period's depreciation expense is charged to the time period in which assets were acquired. Charges half the full amount to all of the assets in the series. You might want to use HALF when assets are acquired during the second half of the time period.

portion-exp

When you want to charge the full amount for some assets and half the amount for other assets, you can supply a *portion-exp* expression that is dimensioned by any of the non-time dimensions of *start-exp.* The *portion-exp* expression must be a text expression with values of FULL or HALF.

switch-period

An integer expression that indicates the time period in which the calculation should switch to the straight-line method. The *switch-period* argument is optional.

A common accounting practice is to switch to a straight-line method in the first period for which straight-line depreciation over the remaining periods exceeds the declining-balance depreciation. You can specify this behavior by *not* specifying the *switch-period* argument.

When the *switch-period* argument is not specified or has a value of `NA` or 0, the calculation switches from the declining method to the straight-line method in the first period for which straight-line depreciation over the remaining periods exceeds the declining-balance depreciation.

When you want to specify different switch periods for different assets, you can supply an expression that is dimensioned by any of the non-time dimensions of *start-exp.*

time-dimension
The name of the time dimension by which *start-exp* and *end-exp* are dimensioned. When the time dimension has a type of DAY, WEEK, MONTH, QUARTER, or YEAR, the *time-dimension* argument is optional.

Notes

Calculation Does Not Switch

When *switch-period* is less than 0 or greater than the number of periods in the depreciation schedule, the calculation does not switch. In this case, the DEPRDECLSW function behaves just like the DEPRDECL function.

How Result Is Dimensioned

The result returned by DEPRDECLSW is dimensioned by all the dimensions of *start-exp.*

Using Optional Arguments

When you include any of the final three optional arguments (FULL or HALF or *portion*, *switch-period*, or *time-dimension*), you must also include the preceding optional arguments.

Depreciation for a Given Time Period

DEPRDECLSW calculates the depreciation expense for a given time period as the sum of that period's depreciation expenses for all assets in the series that are not yet fully depreciated. The first period of depreciation for an asset is the period in which it was acquired.

The HALF Argument

When you specify HALF as the portion of depreciation expenses to charge to the period of acquisition, the HALF factor is applied to each period. Half of each period's full depreciation is rolled to the next period, and the final half period of depreciation takes place in the time period `n + 1`.

Calculation Method Used

For each time period in which DEPRDECLSW is calculating depreciation according to the declining balance method, it calculates the depreciation expense by multiplying the current value of an asset by the *decline-factor* and dividing the result by the number of periods in the lifetime of the asset. When DEPRDECLSW switches to the straight-line method, it subtracts the depreciation expense (from previous periods) from the value of an asset and divides the resulting amount by the number of periods left in the lifetime of the asset. However, when the depreciation expense calculated for a specific time period would result in an asset's current value going below its ending value, then the depreciation expense is adjusted. In this instance, the depreciation expense is calculated as the current value minus the ending value.

Straight-Line Method

The straight-line method as used by DEPRDECLSW differs from the traditional straight-line method as used by DEPRSL. Unlike other methods of depreciation, the declining-balance methods of depreciation ignore the salvage value for an asset until the period in which the calculated depreciation would exceed the remaining depreciable value. This holds true for DEPRDECLSW even after it switches from the declining-balance method to the straight-line method. For example, suppose the beginning value for an asset is 16,000 and the salvage value is 1,000 over 5 periods. The total depreciation through the periods using declining balance method (here the first three) is 11,544. The straight-line calculations for the remaining periods would be based on the overall remaining value of 16,000 minus 11,544 (3,456), rather than the overall value minus the salvage value (2,456). Thus the depreciation for the last two periods would be 1,728; but for the very last period the salvage value is subtracted out and thus is 728.

Most Common Declining-balance Method

This variation on the declining-balance method is the most commonly used form of declining-balance depreciation methods.

Unexpected-Balance Method

When the ending value specified for an asset is relatively high, then an asset might be totally depreciated in fewer periods than were specified for the lifetime of the depreciation. In this instance, when you want the depreciation expense applied across the specified lifetime of the depreciation, you can lower the decline-factor.

NA Mismatch Errors

When a value of *start-exp* is NA and the corresponding value of *end-exp* is not NA, an error occurs. Similarly, when a value of *end-exp* is NA and the corresponding value of *start-exp* is not NA, an error occurs.

NASKIP Option Settings

DEPRDECLSW is affected by the NASKIP option when a value of *start-exp* and the corresponding value of *end-exp* are both NA. When NASKIP is YES (the default), DEPRDECLSW treats the values as zeros when calculating the depreciation expenses. When NASKIP is NO, DEPRDECLSW returns NA for all affected time periods.

Dimension Limits Ignored

The DEPRDECLSW calculation begins with the first time dimension value, regardless of how the status of that dimension may be limited. For example, suppose *start-exp* is dimensioned by year, and the values of year range from Yr95 to Yr99. The calculation always begins with Yr95, even when you limit the status of year so that it does not include Yr95.

Examples

Example 11–3 Using DEPRDECLSW to Calculate Depreciation Expenses for Assets Acquired in a Single Period

This example shows how to use DEPRDECLSW to calculate depreciation expenses for assets acquired in a single time period. It also shows the behavior of DEPRDECLSW when you do not specify a switch period.

The following statements create two variables called assets and salvage.

```
DEFINE assets DECIMAL <year>
DEFINE salvage DECIMAL <year>
```

Suppose you assign the following values to the variables `assets` and `salvage`.

```
YEAR        ASSETS     SALVAGE
------- ---------- -----------
Yr95      1,000.00      100.00
Yr96          0.00        0.00
Yr97          0.00        0.00
Yr98          0.00        0.00
Yr99          0.00        0.00
Yr00          0.00        0.00
```

The variable `assets` contains the starting value of the assets acquired in 1995. `salvage` contains the ending value of the assets acquired in 1995.

The following statement reports the values of assets and salvage, and uses DEPRDECLSW to calculate depreciation expenses for each year, specifying an asset lifetime of 5 years, and a decline factor of 2 (double declining balance). The statement does not specify a *switch-period* argument. Because of this, DEPRDECLSW will use the default for *switch-period,* which is to switch from the declining balance method of depreciation in the first period for which straight-line depreciation over the remaining periods exceeds the declining-balance depreciation.

```
REPORT assets salvage W 12 HEADING 'Depreciation' -
    DEPRDECLSW (assets salvage 5 2 FULL)
```

This statement produces the following report.

```
YEAR        ASSETS     SALVAGE   Depreciation
------- ---------- ----------- --------------
Yr95      1,000.00      100.00      400.00
Yr96          0.00        0.00      240.00
Yr97          0.00        0.00      144.00
Yr98          0.00        0.00      108.00
Yr99          0.00        0.00        8.00
Yr00          0.00        0.00        0.00
```

Example 11–4 Specifying the Switch Period

Alternatively, you can specify the period in which the switch occurs.

To switch from the declining balance method to the straight-line method of depreciation in the third year (`Yr97`), specify `3` as the switch period, as shown in the following statement.

```
REPORT assets salvage W 12 HEADING 'DEPRECIATION' -
    DEPRDECLSW (assets salvage 5 2 FULL 3 year)
```

This statement produces the following report.

```
YEAR        ASSETS     SALVAGE   Depreciation
-------- ---------- ---------- --------------
Yr95      1,000.00      100.00      400.00
Yr96          0.00        0.00      240.00
Yr97          0.00        0.00      120.00
Yr98          0.00        0.00      120.00
Yr99          0.00        0.00       20.00
Yr00          0.00        0.00        0.00
```

Example 11–5 Using DEPRDECLSW to Calculate the Depreciation Expenses for Assets Acquired in Multiple Periods

You can use DEPRDECLSW to calculate the depreciation expenses for a series of assets. Suppose you change the values for the year 1997 in the variables `assets` and `salvage` to the values shown in the following report.

```
YEAR             ASSETS    SALVAGE
-------------- ---------- ----------
Yr95             1,000.00     100.00
Yr96                 0.00       0.00
Yr97               500.00      50.00
Yr98                 0.00       0.00
Yr99                 0.00       0.00
Yr00                 0.00       0.00
Yr01                 0.00       0.00
Yr02                 0.00       0.00
```

Now `assets` and `salvage` contain nonzero values for 1995 and for 1997.

The following statement reports asset and salvage values along with depreciation expenses for the assets. Note that the call to DEPRDECLSW to calculate the depreciation expenses specifies an asset lifetime of 5 periods (in this case, years) and a decline factor of 2 (double-declining balance). The statement does not specify a *switch-period* argument. Because of this, DEPRDECLSW will use the default for *switch-period,* which is to switch from the declining balance method of depreciation in the first period for which straight-line depreciation over the remaining periods exceeds the declining-balance depreciation.

```
REPORT assets salvage W 12 HEADING 'Depreciation'  -
          DEPRDECLSW(assets salvage 5 2 FULL)
```

This statement produces the following output.

```
YEAR              ASSETS    SALVAGE   Depreciation
-------------- ---------- ---------- ------------
Yr95            1,000.00     100. 00       400.00
Yr96                0.00       0.00       240.00
Yr97              500.00      50.00       344.00
Yr98                0.00       0.00       228.00
Yr99                0.00       0.00        80.00
Yr00                0.00       0.00        54.00
Yr01                0.00       0.00         4.00
Yr02                0.00       0.00         0.00
```

Notice that the depreciation expense increases in 1997 due to the assets acquired in that year.

DEPRSL

The DEPRSL function calculates the depreciation expenses for a series of assets. DEPRSL uses the straight-line method to depreciate the assets over the specified lifetime of the assets. The starting and ending values are specified for the assets acquired in each time period.

Return Value

DECIMAL

Syntax

DEPRSL(*start-exp end-exp n* [{FULL|HALF| *portion-exp*} [*time-dimension*]])

Arguments

start-exp
A numeric expression that contains the starting values of the assets. The *start-exp* expression must be dimensioned by a time dimension. For each value of the time dimension, *start-exp* contains the initial value of the assets acquired during that time period. In addition to a time dimension, *start-exp* can also have non-time dimensions.

end-exp
A numeric expression that contains the ending values of the assets. The *end-exp* expression must be dimensioned by the same dimensions as *start-exp.* For each value of the time dimension, *end-exp* contains the final (or salvage) value for the assets acquired during that time period. Each value of *start-exp* must have a corresponding *end-exp* value. For example, when the assets acquired in 1995 have a salvage value of $200, then the value of *end-exp* for 1995 is $200.

n
An integer expression that contains the depreciation lifetime of the assets. The *n* expression can have any of the non-time dimensions of *start-exp,* but it cannot have a time dimension.

FULL
Specifies that the full amount of a time period's depreciation expense is charged to the time period in which assets were acquired. Charges the full amount to all of the assets in the series. (Default)

HALF
Specifies that half of the full amount of a time period's depreciation expense is charged to the time period in which assets were acquired. Charges half the full amount to all of the assets in the series. You might want to use HALF when assets are acquired during the second half of the time period.

portion-exp
When you want to charge the full amount for some assets and half the amount for other assets, you can supply a *portion-exp* expression that is dimensioned by any of the non-time dimensions of *start-exp.* The *portion-exp* expression must be a text expression with values of FULL or HALF.

time-dimension
The name of the time dimension by which *start-exp* and *end-exp* are dimensioned. When the time dimension has a type of DAY, WEEK, MONTH, QUARTER, or YEAR, the *time-dimension* argument is optional.

Notes

How Result Is Dimensioned

The result returned by DEPRSL is dimensioned by all the dimensions of *start-exp.*

Using Optional Arguments

When you include the optional *time-dimension* argument, you must also include the preceding optional argument (FULL or HALF or *portion*).

Depreciation for a Given Time Period

DEPRSL calculates the depreciation expense for a given time period as the sum of that period's depreciation expenses for all assets in the series that are not yet fully depreciated. The first period of depreciation for an asset is the period in which it was acquired.

The HALF Argument

When you specify HALF as the portion of depreciation expenses to charge to the period of acquisition, the HALF factor is applied to each period. Half of each period's full depreciation expense is rolled to the next period, and the final half period of depreciation takes place in the time period `n + 1`.

NA Mismatch Errors

When a value of *start-exp* is `NA` and the corresponding value of *end-exp* is not `NA`, an error occurs. Similarly, when a value of *end-exp* is `NA` and the corresponding value of *start-exp* is not `NA`, an error occurs.

NASKIP Option Settings

DEPRSL is affected by the NASKIP option when a value of *start-exp* and the corresponding value of *end-exp* are both `NA`. When NASKIP is `YES` (the default), DEPRSL treats the values as zeros when calculating the depreciation expenses. When NASKIP is `NO`, DEPRSL returns `NA` for all affected time periods.

Dimension Limits Ignored

The DEPRSL calculation begins with the first time dimension value, regardless of how the status of that dimension may be limited. For example, suppose *start-exp* is dimensioned by `year`, and the values of `year`, range from `Yr95` to `Yr99`. The calculation always begins with `Yr95`, even when you limit the status of `year`, so that it does not include `Yr95`.

Related Functions

The DEPRDECL function, which uses the pure declining-balance method to calculate depreciation expenses; the DEPRDECLSW function, which uses a more widely used variation on the declining-balance method to calculate depreciation expenses; and the DEPRSOYD function, which uses the sums-of-years'-digits method to calculate depreciation expenses.

Examples

Example 11–6 Using DEPRSL to Calculate Depreciation Expenses for Assets Acquired in a Single Period

This example shows how to use DEPRSL to calculate depreciation expenses for assets acquired in a single time period.

The following statements create two variables called `assets` and `salvage`.

```
DEFINE assets DECIMAL <year>
DEFINE salvage DECIMAL <year>
```

Suppose you assign the following values to the variables `assets` and `salvage`.

```
YEAR               ASSETS    SALVAGE
-------------- ---------- ----------
Yr95             1,000.00     100.00
Yr96                 0.00       0.00
Yr97                 0.00       0.00
Yr98                 0.00       0.00
Yr99                 0.00       0.00
Yr00                 0.00       0.00
```

The variable `assets` contains the starting value of assets acquired in 1995. The variable `salvage` contains the ending value of the assets acquired in 1995.

The following statement reports the values of assets and salvage, and uses DEPRSL to calculate depreciation expenses for each year, specifying an asset lifetime of 5 years.

```
REPORT assets salvage W 12 HEADING 'Depreciation' -
   DEPRSL(assets salvage 5 FULL year)
```

This statement produces the following output.

```
YEAR               ASSETS    SALVAGE Depreciation
-------------- ---------- ---------- ------------
Yr95             1,000.00     100.00       180.00
Yr96                 0.00       0.00       180.00
Yr97                 0.00       0.00       180.00
Yr98                 0.00       0.00       180.00
Yr99                 0.00       0.00       180.00
Yr00                 0.00       0.00         0.00
```

Example 11–7 Using DEPRSL to Calculate the Depreciation Expenses for Assets Acquired in Multiple Periods

You can also use DEPRSL to calculate the depreciation expenses for a series of assets. Suppose you change the values for the year 1997 in the variables `assets` and `salvage` to the values shown in the following report.

```
YEAR                ASSETS    SALVAGE
-------------- ---------- ----------
Yr95             1,000.00     100.00
Yr96                 0.00       0.00
Yr97               500.00      50.00
Yr98                 0.00       0.00
Yr99                 0.00       0.00
Yr00                 0.00       0.00
Yr01                 0.00       0.00
Yr02                 0.00       0.00
```

Now `assets` and `salvage` contain nonzero values for 1995 and for 1997.

The following statement reports asset and salvage values along with depreciation expenses for the assets. Note that the call to DEPRSL to calculate the depreciation expenses specifies an asset lifetime of 5 periods (in this case, years).

```
REPORT assets salvage W 12 HEADING 'Depreciation' -
    DEPRSL(assets salvage 5 FULL year)
```

This statement produces the following report.

```
YEAR               ASSETS        SALVAGE         Depreciation
-------------- ---------- ------------- --------------------
Yr95             1,000.00       100.00               180.00
Yr96                 0.00         0.00               180.00
Yr97               500.00        50.00               270.00
Yr98                 0.00         0.00               270.00
Yr99                 0.00         0.00               270.00
Yr00                 0.00         0.00                90.00
Yr01                 0.00         0.00                90.00
Yr02                 0.00         0.00                 0.00
```

The assets acquired in 1995 were fully depreciated in 1999. Therefore, for 2000 and 2001, DEPRSL returns a figure that includes the depreciation expense for the assets acquired in 1997 only.

DEPRSOYD

The DEPRSOYD function calculates the depreciation expenses for a series of assets. DEPRSOYD uses the sum-of-years'-digits method to depreciate the assets over the specified lifetime of the assets. The starting and ending values are specified for the assets acquired in each time period.

Return Value

DECIMAL

Syntax

DEPRSOYD(*start-exp end-exp n* [{FULL|HALF| *portion-exp*} [*time-dimension*]])

Arguments

start-exp
A numeric expression that contains the starting values of the assets. The *start-exp* expression must be dimensioned by a time dimension. For each value of the time dimension, *start-exp* contains the initial value of the assets acquired during that time period. In addition to a time dimension, *start-exp* can also have non-time dimensions.

end-exp
A numeric expression that contains the ending values of the assets. The *end-exp* expression must be dimensioned by the same dimensions as *start-exp.* For each value of the time dimension, *end-exp* contains the final (or salvage) value for the assets acquired during that time period. Each value of *start-exp* must have a corresponding *end-exp* value. For example, when the assets acquired in 1995 have a salvage value of $200, then the value of *end-exp* for 1995 is $200.

n
An integer expression that contains the depreciation lifetime of the assets. The *n* expression can have any of the non-time dimensions of *start-exp,* but it cannot have a time dimension.

FULL
Specifies that the full amount of a time period's depreciation expense is charged to the time period in which assets were acquired. Charges the full amount to all of the assets in the series. (Default)

HALF
Specifies that half of the full amount of a time period's depreciation expense is charged to the time period in which assets were acquired. Charges half the full amount to all of the assets in the series. You might want to use HALF when assets are acquired during the second half of the time period.

portion-exp
When you want to charge the full amount for some assets and half the amount for other assets, you can supply a *portion-exp* expression that is dimensioned by any of the non-time dimensions of *start-exp.* The *portion-exp* expression must be a text expression with values of FULL or HALF.

time-dimension
The name of the time dimension by which *start-exp* and *end-exp* are dimensioned.When the time dimension has a type of DAY, WEEK, MONTH, QUARTER, or YEAR, the *time-dimension* argument is optional.

Notes

How Result Is Dimensioned

The result returned by DEPRSOYD is dimensioned by all the dimensions of *start-exp.*

Using Optional Arguments

When you include the optional *time-dimension* argument, you must also include the preceding optional argument (FULL or HALF or *portion*).

Depreciation for a Given Time Period

DEPRSOYD calculates the depreciation expense for a given time period as the sum of that period's depreciation expenses for all assets in the series that are not yet fully depreciated. The first period of depreciation for an asset is the period in which it was acquired.

Calculation Method Used

For each time period in the lifetime of an asset, DEPRSOYD bases the depreciation expense calculation on a specific cut of the total amount to be depreciated. The value of the cut is such that the full depreciation expense can be achieved over the lifetime of an asset by multiplying the cut by the number of time periods not yet depreciated.

For example, when the lifetime of an asset is 5 years, then DEPRSOYD calculates the cut, *x,* as follows.

```
5x + 4x + 3x + 2x + 1x = total depreciation
```

In this case, the cut is 1/15th of the total depreciation. When the initial asset is $1,000 and its salvage value is $100, then the total depreciation is $900.00, and x is $60 ($900/15). For the first time period, the depreciation is $300 ($60 x 5). For the second time period, the depreciation is $240 ($60 x 4) and so on.

The HALF Argument

When you specify HALF as the portion of depreciation expenses to charge to the period of acquisition, the HALF factor is applied to each period. Half of each period's full depreciation expense is rolled to the next period, and the final half period of depreciation expense takes place in the *n* + 1 time period.

NA Mismatch Errors

When a value of *start-exp* is `NA` and the corresponding value of *end-exp* is not `NA`, an error occurs. Similarly, when a value of *end-exp* is `NA` and the corresponding value of *start-exp* is not `NA`, an error occurs.

NASKIP Option Settings

DEPRSOYD is affected by the NASKIP option when a value of *start-exp* and the corresponding value of *end-exp* are both `NA`. When NASKIP is `YES` (the default), DEPRSOYD treats the values as zeros when calculating the depreciation expenses. When NASKIP is `NO`, DEPRSOYD returns `NA` for all affected time periods.

Dimension Limits Ignored

The DEPRSOYD calculation begins with the first time dimension value, regardless of how the status of that dimension may be limited. For example, suppose *start-exp* is dimensioned by `year`, and the values of `year` range from `Yr95` to `Yr99`. The calculation always begins with `Yr95`, even when you limit the status of `year` so that it does not include `Yr95`.

Related Functions

The DEPRDECL function, which uses the pure declining-balance method to calculate depreciation expenses; the DEPRDECLSW function, which uses a more widely used variation on the declining-balance method to calculate depreciation expenses; and the DEPRSL function, which uses the straight-line method to calculate depreciation expenses.

Examples

Example 11–8 Using DEPRSOYD to Calculate Depreciation Expenses for Assets Acquired in a Single Period

This example shows how to use DEPRSOYD to calculate depreciation expenses for assets acquired in a single time period.

The following statements create two variables called `assets` and `salvage`.

```
DEFINE assets DECIMAL <year>
DEFINE salvage DECIMAL <year>
```

Suppose you assign the following values to the variables `assets` and `salvage`.

```
YEAR               ASSETS    SALVAGE
-------------- ---------- ----------
Yr95             1,000.00     100.00
Yr96                 0.00       0.00
Yr97                 0.00       0.00
Yr98                 0.00       0.00
Yr99                 0.00       0.00
Yr00                 0.00       0.00
```

The variable `assets` contains the starting value of assets acquired in 1995. The variable `salvage` contains the ending value of the assets acquired in 1995.

The following statement reports the values of `assets` and `salvage`, and uses DEPRSOYD to calculate depreciation expenses for each year, specifying an asset lifetime of 5 years.

```
REPORT assets salvage W 12 HEADING 'Depreciation' -
     DEPRSOYD(assets salvage 5 FULL year)
```

This statement produces the following report.

```
YEAR               ASSETS    SALVAGE Depreciation
-------------- ---------- ---------- ------------
Yr95             1,000.00     100.00       380.00
Yr96                 0.00       0.00       240.00
Yr97                 0.00       0.00       180.00
Yr98                 0.00       0.00       120.00
Yr99                 0.00       0.00        60.00
Yr00                 0.00       0.00         0.00
```

Example 11–9 Using DEPRSOYD to Calculate the Depreciation Expenses for Assets Acquired in Multiple Periods

You can also use DEPRSOYD to calculate the depreciation expenses for a series of assets. Suppose you change the values for the year 1997 in the variables `assets` and `salvage` to the values shown in the following report.

```
   YEAR              ASSETS    SALVAGE
   -------------- ---------- ----------
   Yr95             1,000.00     100.00
   Yr96                 0.00       0.00
   Yr97               500.00      50.00
   Yr98                 0.00       0.00
   Yr99                 0.00       0.00
   Yr00                 0.00       0.00
   Yr01                 0.00       0.00
   Yr02                 0.00       0.00
```

Now `assets` and `salvage` contain nonzero values for 1995 and for 1997.

The following statement reports asset and salvage values along with depreciation expenses for the assets. Note that the call to DEPRSOYD to calculate the depreciation expenses specifies an asset lifetime of 5 periods (in this case, years).

```
REPORT assets salvage W 12 HEADING 'Depreciation' -
     DEPRSOYD(assets salvage 5 FULL year)
```

This statement produces the following output.

```
YEAR              ASSETS    SALVAGE   Depreciation
-------------- ---------- ---------- ------------
Yr95             1,000.00     100.00       300.00
Yr96                 0.00       0.00       240.00
Yr97               500.00      50.00       330.00
Yr98                 0.00       0.00       240.00
Yr99                 0.00       0.00       160.00
Yr00                 0.00       0.00        60.00
Yr01                 0.00       0.00        30.00
Yr02                 0.00       0.00         0.00
```

Notice that as a result of the second asset, the depreciation expenses increase in 1997. The depreciation is the total depreciation of $180.00 (`$60 x 3`) for the first asset and $150.00 (`$30 x 5`) for the second asset.

DESCRIBE

The DESCRIBE command produces a report that shows the definition of one or more workspace objects.

Syntax

DESCRIBE [*names*]

Arguments

names

The names of one or more workspace objects, separated by spaces or commas. DESCRIBE includes the definition of each specified object in its report. When you omit this argument, DESCRIBE shows the definition of all objects in the current status of NAME.

Notes

Output of DESCRIBE

The object definition that you see in the output from a DESCRIBE command might include a description (LD), a value name format (VNF) for a time dimension, an expression associated with a FORMULA, permission specified with PERMIT commands, trigger programs associated with the object (TRIGGER command), or the contents of a calculation specification (for example, the contents of a program or model).

Limiting the Objects Described

Normally, the status of NAME is ALL, so DESCRIBE with no argument produces a report that includes the definitions of all objects in your current workspace. However, you can use the LIMIT command in combination with DESCRIBE to report the definitions of a particular group of objects in your workspace. First use LIMIT to limit the status of the NAME dimension to the names of the objects whose definitions you want to see. Then execute a DESCRIBE command with no arguments to produce a report of the definitions. See Example 11–11, "Describing All Relations" on page 11-30.

Paginated Output

You can produce paginated output with the DESCRIBE command by setting PAGING to `YES` before using DESCRIBE.

DESCRIBE and PERMIT

You can use DESCRIBE to show the definition of an object even when you do not have permission to access the object or to change its permission. However, when a PERMIT command denies you the right to change the permission of an object, DESCRIBE does not include the permission associated with the definition of the object.

Describing Worksheets

For a worksheet definition, the DESCRIBE report does not include the default dimensions, `WKSCOL` and `WKSROW`. However, it does include user-defined dimensions when they have been used to define a worksheet. See Example 11–12, "Describing a Worksheet" on page 11-31.

Describing Object Properties

The object definitions in the output from the DESCRIBE command do not include the properties associated with objects. To include properties, you must use the FULLDSC command. See the entries for FULLDSC and PROPERTY.

Creating Objects with DESCRIBE Output

You can use the output from the DESCRIBE command to create objects in other workspaces, because each line of the output is a valid statement. For example, you can execute an OUTFILE command to send subsequent output to a file, and then execute a DESCRIBE command. You can then access another workspace and use the INFILE command to read the DESCRIBE output. The same object will be created in that workspace.

Describing Composites and Conjoints

When you define a composite or conjoint that uses an index type other than the default, the DESCRIBE command displays the index type. When you use the default index type (HASH for conjoints, BTREE for composites), that information is not displayed.

Describing Width of Dimensioned BOOLEAN Variables

Dimensioned BOOLEAN variables that are in older 1 or 2 byte formats are listed as WIDTH 1 and WIDTH 2. The width of BOOLEAN variables created in the new single-bit format is not listed.

Examples

Example 11–10 Describing Variables

This example produces a report of the definitions of the two variables, `sales` and `price`. The statement

```
DESCRIBE sales price
```

produces the following output.

```
DEFINE SALES VARIABLE DECIMAL <MONTH PRODUCT DISTRICT>
LD Sales Revenue
DEFINE PRICE VARIABLE DECIMAL <MONTH PRODUCT>
LD Wholesale Unit Selling Price
```

Example 11–11 Describing All Relations

Suppose you want to look at the definitions of all the relations in your workspace. First limit the NAME dimension, using the OBJ function. After limiting NAME, use DESCRIBE with no arguments to produce a report of the definitions. The statements

```
LIMIT NAME TO OBJ(TYPE) EQ 'RELATION'
describe
```

produce the following output.

```
DEFINE REGION.DISTRICT RELATION REGION <DISTRICT>
LD REGION for each DISTRICT
DEFINE DIVISION.PRODUCT RELATION DIVISION <PRODUCT>
LD DIVISION for each PRODUCT
DEFINE MLV.MARKET RELATION MARKETLEVEL <MARKET>
DEFINE MARKET.MARKET RELATION MARKET <MARKET>
LD Self-relation for the Market Dimension
```

Since the values returned by OBJ(TYPE) are always in uppercase, you have to use 'RELATION' rather than 'relation' in your LIMIT command to obtain a match.

Example 11–12 Describing a Worksheet

The dimensions of a worksheet appear in the description only when they are user-defined dimensions. The default dimensions `WKSCOL` and `WKSROW` are not included in the description. The statements

```
DEFINE work1 WORKSHEET
DEFINE columns DIMENSION INTEGER
DEFINE rows DIMENSION INTEGER
DEFINE work2 WORKSHEET <columns rows>
DESCRIBE work1 work2
```

produce the following output.

```
DEFINE WORK1 WORKSHEET
DEFINE WORK2 WORKSHEET <COLUMNS ROWS>
```

DIVIDEBYZERO

The DIVIDEBYZERO option controls the result of division by zero.

Data type

BOOLEAN

Syntax

DIVIDEBYZERO = YES|NO

Arguments

YES
Allows division by zero. A statement involving division by zero will execute without error; however, the result of the division by zero will be `NA`. When you are dividing by a dimensioned variable or expression, setting DIVIDEBYZERO to `YES` enables you to get results for most of the expression's values when a few calculations might involve dividing by zero.

NO
Disallows division by zero. A statement involving division by zero will stop executing and produce an error message. (Default)

Notes

Negative Powers of Zero

Raising zero to a negative power (for example, `0 ** -2`) is division by zero.

Statements Affected by DIVIDEBYZERO

AGGREGATE command and AGGREGATE function

Examples

Example 11–13 The Effect of DIVIDEBYZERO

This example shows the effect of changing the value of the DIVIDEBYZERO option.

When you execute a SHOW command, such as the following, without changing the DIVIDEBYZERO option from its default value of `NO`, Oracle OLAP attempts to divide 100 by 0 and then produces an error message.

```
SHOW 100 / 0
```

When you change DIVIDEBYZERO to `YES`, the same statement executes without error and produces `NA` as the result of the division. The statements

```
DIVIDEBYZERO = YES
SHOW 100 / 0
```

produce the following result.

```
NA
```

DO ... DOEND

The DO and DOEND commands bracket a group of one or more statements in a program. DO and DOEND are normally used to bracket one of the following:

- A group of statements that are to be executed under a condition specified by an IF command
- A group of statements in a repeating loop introduced by FOR or WHILE
- The CASE labels for a SWITCH command.

DO and DOEND are like opening and closing parentheses; you cannot use one without the other. You can use DO and DOEND only within programs.

Syntax

```
DO
   statement1
   ...
   statementN
DOEND
```

Arguments

statement
One or more OLAP DML statements, user-defined programs, or both.

Notes

Nesting

You can put one DO statement inside another to nest groups of statements. You can nest as many groups as you want, if each DO statement has a corresponding DOEND to indicate the end of its statement group.

TEMPSTAT Command and DOEND Command

Within a FOR loop of a program, when a DO/DOEND phrase follows TEMPSTAT, status is restored when the DOEND, BREAK, or GOTO is encountered.

Examples

Example 11–14 DO and DOEND with the FOR Command

Suppose you want to use the ROW command to produce a report that shows the unit sales of tents for each of 2 months. Use DO ... DOEND and DOEND to bracket the ROW and BITAND commands you want to execute repeatedly for each value of the `month` dimension. You might write the following program.

```
LIMIT month TO 'Jan96' to 'Feb96'
ROW district
ROW UNDER '-' VALONLY name.product
FOR month
    DO
      ROW INDENT 5 month WIDTH 6 UNITS
      BLANK
    DOEND
```

The program produces the following output.

```
BOSTON
3-Person Tents
--------------
     Jan96              307
     Feb96              209
```

DSECONDS

(Read-only) The DSECONDS option holds the number of seconds since January 1, 1970. The value of DSECONDS is in decimal (not integer form). In most cases, depending on your operating system, the decimal places after the second one have a value of zero because the measurement cannot be accurate at a more detailed level.

As an aid to enhancing a program's speed, DSECONDS can be used to determine how much time elapses while the program is running.

Data type

DECIMAL

Syntax

```
DSECONDS
```

Notes

Related Statements

For information about holding the number of seconds in integer form, see the SECONDS command. For information about programs, see the PROGRAM command.

Examples

Example 11–15 Timing a Program

The following program puts the value of DSECONDS at the start of the program in a variable called `t1` and then displays the difference between `t1` and the value of DSECONDS at the end of the program.

```
DEFINE prodsummary PROGRAM
PROGRAM
VARIABLE t1 DECIMAL
t1 = dseconds
LIMIT product TO ALL
BLANK
FOR product
DO
  ROW WIDTH 16 name.product ACROSS month Jun96: DECIMAL 0 LSET -
   '$'WIDTH 18 <RSET ' (Actual)' sales rset ' (Plan)' sales.plan>
DOEND
BLANK
ROW WIDTH 35 LSET 'The program took ' rset ' seconds.' -
 (dseconds - t1)
END
```

Running this program produces the following results.

```
3-Person Tents     $95,121 (actual)     $80,138 (plan)
Aluminum Canoes   $157,762 (actual)    $132,931 (plan)
Tennis Racquets    $97,174 (actual)     $84,758 (plan)
Warm-up Suits      $79,630 (actual)     $73,569 (plan)
Running Shoes     $153,688 (actual)    $109,219 (plan)

      The program took .20 seconds.
```

ECHOPROMPT

The ECHOPROMPT option determines whether or not input lines and error messages should be echoed to the current outfile. When ECHOPROMPT is set to `YES` and you have specified a debugging file with DBGOUTFILE, the input lines and error messages are echoed to the debugging file instead of the current outfile.

Data type

BOOLEAN

Syntax

```
ECHOPROMPT = {YES|NO}
```

Arguments

YES

Input lines and error messages are echoed to the current outfile or the debugging file specified by DBGOUTFILE.

NO

Input lines and error messages do not appear in the current outfile or in the debugging file. (Default)

Notes

ECHOPROMPT

The ECHOPROMPT option causes input, as well as error messages, to be echoed to the current outfile, or to the debugging file when there is one.

Current and Default Outfiles

The current outfile is the destination for the output of statements, such as REPORT and DESCRIBE, that produce text. When you have not used the OUTFILE command to send output to a file, Oracle OLAP uses your default outfile.

Examples

Example 11–16 Using ECHOPROMPT

Suppose you want to have all input lines and error messages included in the disk file that will contain your output. Set ECHOPROMPT to `YES` before issuing the OUTFILE command that will send the output to the disk file. In the following statements, the disk file is in the current directory object.

```
ECHOPROMPT = YES
OUTFILE 'newcalc.dat'
```

EDIT

The EDIT command displays an OLAP Worksheet Edit window. The command is available only when you are using OLAP Worksheet to access Oracle OLAP.

For information about using an OLAP Worksheet Edit window, see the OLAP Worksheet Help.

Syntax

EDIT [PROGRAM|MODEL|AGGMAP|FORMULA] *object-name*

Arguments

PROGRAM
MODEL
AGGMAP
FORMULA
Indicates whether the object to be edited is a program, a model, an aggmap, or a formula.

object-name
A text expression that specifies the name of an existing program, model, aggmap, or formula. Before editing one of these objects, use the DEFINE command to create it in an analytic workspace.

Notes

Specifying the Type of an Aggmap

There are two types of aggmaps, one for aggregating data and another for allocating data. You can obtain the type of an aggmap by using the AGGMAPINFO function with the MAPTYPE keyword.

When an aggmap is first created, its type is `NA`. Once you use either the AGGMAP or the ALLOCMAP command to reference the new aggmap, Oracle OLAP specifies its type. When you use the EDIT command on an aggmap whose type has not yet been specified, OLAP Worksheet assumes that it is to be used for aggregating data.

When you plan to use an aggmap for allocating data, use the following statements to identify it as an allocation specification before the first time you open an OLAP Worksheet Edit window for it.

```
CONSIDER aggmap-name
ALLOCMAP 'END'
```

The ALLOCMAP command causes Oracle OLAP to record the fact that this aggmap is for allocating data.

Examples

Example 11–17 Editing a Program

The following statement, executed in the OLAP Worksheet, places the `myprog` program in an OLAP Worksheet EDIT window.

```
EDIT myprog
```

Example 11–18 Editing a Model

The following statement, executed in the OLAP Worksheet, places a model called `myModel` in an OLAP Worksheet Edit window.

```
EDIT MODEL myModel
```

EIFBYTES

(Read-only) The EIFBYTES option holds the number of bytes read by the most recent IMPORT (from EIF) command or written by the most recent EXPORT (to EIF) command.

Data type

INTEGER

Syntax

EIFBYTES

Examples

Example 11–19 Finding Out the Number of Bytes

To find out how many bytes of information were exported to an EIF file when you exported the dimensions of the `demo` workspace, you use the following statements.

```
LIMIT name TO OBJ(TYPE) EQ 'DIMENSION'
EXPORT ALL TO EIF FILE 'myfile.eif'
SHOW EIFBYTES
```

The SHOW command produces the following output.

```
2,038
```

EIFEXTENSIONPATH

The EIFEXTENSIONPATH option contains a list of directory objects that identify the locations where EIF extension files should be created.

Data type

TEXT

Syntax

EIFEXTENSIONPATH = *path-expression*

Arguments

path-expression
A text expression that contains one or more directory object names. When you specify multiple aliases, you must enter each one on a separate line. Specify multiple aliases in the order in which they should be used for storing EIF extension files.

Notes

When Extension Files Are Created

When the size of an EIF file grows beyond the size specified for EIF files by the FILESIZE argument to the EXPORT (to EIF) command, or the current disk or location becomes full, an EIF extension file is created.

Before creating a new extension file, the location specified by EIFEXTENSIONPATH is checked for sufficient disk space. The required amount of disk space is the amount specified for FILESIZE in the EXPORT (to EIF). When no value has been specified for FILESIZE, then a check is made for at least 80K of disk space (the minimum size allowed by FILESIZE). When there is insufficient disk space, checking continues through the list until a location with enough available disk space is found.

Multiple Paths in EIFEXTENSIONPATH

When EIFEXTENSIONPATH contains multiple directory objects, the first extension file is created in the first alias in the list. The second extension file is created in the second alias on the list, and so on. When the end of the list is reached, the process starts over again at the beginning. When EIFEXTENSIONPATH contains a single directory object, all extension files are created in that location.

Identifying Files and Directories

When specifying files and directories in OLAP DML statements, it is good practice to always enclose them in single quotes.

Examples

Example 11–20 Establishing a Location for Extension Files

The following statement establishes the `eifext` directory object as the location in which EIF extension files should be created.

```
EIFEXTENSIONPATH = 'eifext'
```

EIFNAMES

The EIFNAMES option holds a list of the names of all the objects imported by the most recent IMPORT (from EIF) command.

Data type

TEXT

Syntax

EIFNAMES

Notes

Related Statements

IMPORT (from EIF) and EIFTYPES.

Examples

Checking What You Have Imported

Suppose you have exported the `units` variable and the `productset` valueset from the `demo` analytic workspace to a file called `myfile.eif`. After importing the contents of the file into a new workspace, you can use the EIFNAMES option to see the names of the objects you have just imported.

The following statements

```
AW CREATE mytest
IMPORT ALL FROM EIF FILE 'myfile.eif'
SHOW EIFNAMES
```

produce this output.

```
DISTRICT
PRODUCT
MONTH
UNITS
PRODUCTSET
```

EIFSHORTNAMES

The EIFSHORTNAMES option controls the structure of the extension of EIF overflow (extension) file names.

Data type

BOOLEAN

Syntax

EIFSHORTNAMES = YES|NO

Arguments

YES
Sets the extension of EIF overflow (extension) file names to *`xx`*, where each *`x`* is an automatically assigned lowercase letter between `a` and `z`.

NO
Sets the extension of EIF overflow (extension) file names to `e`*`nnn`*, where *nnn* is a three-digit number beginning with 001.(Default)

Notes

EIF Extension File Names

By default, EIF extension file names have the structure *filename*.e*nnn*, where *nnn* is a three-digit number beginning with 001, to distinguish them from workspace extension file names.

For example, when an EIF file is named `export.eif`, the extension files are named `export.e001`, `export.e002`, and so on.

When EIFSHORTNAMES is set to `YES`, the extension files are named `export._aa`, `export._ab`, and so on.

Examples

Example 11–21 Limiting the Extension of a File Name to Three Characters

The following statement specifies that the file extension for EIF extension file names must be in the form *xx*.

```
EIFSHORTNAMES = YES
```

EIFTYPES

The EIFTYPES option holds a list of the types of objects that are contained in the list produced by the EIFNAMES option. The types are listed in the same order as the corresponding object names in the EIFNAMES list.

Data type

TEXT

Syntax

EIFTYPES

Notes

Related Statements

IMPORT (from EIF) and EIFNAMES.

Examples

Example 11–22 Checking What You Have Imported

Suppose you have exported the `units` variable and the `productset` valueset from an analytic workspace named `demo` to a file called `myfile.eif`. After importing the contents of the file into a new workspace, you can use the EIFNAMES and EIFTYPES options to see the names and object types of the objects you have just imported.

Create the workspace and import the objects with these statements.

```
AW CREATE mytest
IMPORT ALL FROM EIF FILE 'myfile.eif'
```

Send the names of the imported objects to the current outfile with this statement

```
SHOW EIFNAMES
```

to produce this output.

```
DISTRICT
PRODUCT
MONTH
UNITS
PRODUCTSET
```

Send the types of the imported objects to the current outfile with this statement

```
SHOW EIFTYPES
```

to produce this output.

```
DIMENSION
DIMENSION
DIMENSION
VARIABLE
VALUESET
```

EIFUPDBYTES

The EIFUPDBYTES option controls the frequency of updates when you are using the IMPORT (from EIF) command with its UPDATE keyword. The value of EIFUPDBYTES has an effect only when the UPDATE keyword is specified in this command.

Data type

INTEGER

Syntax

EIFUPDBYTES = *n*

Arguments

n
An integer expression that specifies the minimum number of bytes to be read between updates, during an import. When EIFUPDBYTES has a value of 0, an update is triggered after each analytic workspace object is imported. When EIFUPDBYTES has a value greater than 0, an update is triggered each time the specified number of bytes is imported. The default is `0` (zero).

Examples

Example 11–23 Reducing Update Frequency

In the following example, the UPDATE keyword in the IMPORT (from EIF) command ensures that updates will occur periodically. The setting of EIFUPDBYTES ensures that the updates will not occur too often.

```
EIFUPDBYTES = 500000
IMPORT ALL FROM EIF FILE 'finance.eif' UPDATE
```

EIFVERSION

The EIFVERSION option is used with the EXPORT (to EIF) and IMPORT (from EIF) commands to copy data between different versions of Express® Server or Oracle OLAP. The version from which the data is exported is referred to as the source. The version to which the data is imported is referred to as the target.

Before you use the EXPORT command to export data to an EIF file, you use the EIFVERSION option to specify the internal version or build number of the target. Then, when you use EXPORT to copy data from the source to an EIF file, the data will be in a format that can be imported by the target.

You can use the EVERSION function to determine the internal version or build number of the target.

Syntax

EIFVERSION = *n*

Arguments

n
The internal version or build number of an Express Server or Oracle OLAP process. This is the target into which you want the data imported.

By default, EIFVERSION is set to the internal version or build number of the current process.

Notes

Versions Later Than a Specified Version

Generally, you can import data from an EIF file into any target that has a later version number than the one you specify for the EIF file with EIFVERSION.

Versions Earlier Than the Current Version

When you set EIFVERSION to a value that is lower than the default version (that is, the version number of the current process), and you try to export data that the earlier version cannot manage, an error is generated. For example, when you try to export an aggmap to a 6.2 version of Express Server, an error is generated because Express Server 6.2 cannot manage aggmaps.

Examples

Example 11–24 Exporting and Importing Between Different Versions

This example shows how to use EIFVERSION when you want to export data from Oracle OLAP to an EIF file and then import it into Express Server version 6.2.0.

This statement (issued from the target 6.2.0 Express Server)

```
SHOW EVERSION
```

returns the following version and build information

```
Module Mgr, Version: 6.2.0.0.0, Build: 60232
OES Kernel, Version: 6.2.0.0.0, Build: 60232
```

The following statements export the data from Oracle OLAP (which has a higher build number than 60232) to an EIF file that can be read in Express 6.2.0

```
EIFVERSION = 60232
EXPORT ALL TO EIF FILE 'myeif.eif'
```

END

The END command marks the end of the specification for an aggmap, a model, or a program.

You do not need to type the END command when you enter the specification for one of these objects in the OLAP Worksheet Edit window, because OLAP Worksheet automatically adds the END command when you save the object. However, when you execute an aggmap, a model, or a program using the INFILE command, you must ensure that the file includes the END command.

Syntax

```
END
```

ENDDATE

The ENDDATE function returns the ending date of the last time period for which an expression has a non-`NA` value.

> **Note:** You can only use this function with dimensions of type DAY, WEEK, MONTH, QUARTER, or YEAR.You can*not* use this function for time dimensions that are implemented as hierarchical dimensions of type TEXT.

Return Value

DATE

Syntax

ENDDATE(*expression*)

Arguments

expression

The expression must have exactly one dimension that has the type of DAY, WEEK, MONTH, QUARTER, or YEAR.

Notes

NA Values

When all the values of the expression are `NA`, ENDDATE returns `NA`.

How ENDDATE Works

ENDDATE returns the final date of the last time period in the dimension status for which the expression has a non-`NA` value. For example, when an expression is dimensioned by MONTH, and when `DEC98` is the last dimension value for which the expression has a non-`NA` value, ENDDATE returns the date `December 31, 1998`.

Format of the Date

When you display the result returned by ENDDATE, the date is formatted according to the date template in the DATEFORMAT option. When the day of the week or the name of the month is used in the date template, the day names specified in the DAYNAMES option and the month names specified in the MONTHNAMES option are used. You can use the result returned by ENDDATE anywhere that a DATE value is expected.

DATE-to-TEXT Conversion

You can also use the result where a text value is expected. The date is automatically converted to a text value, using the current template in the DATEFORMAT option to format the text value. When you want to override the current DATEFORMAT template, you can convert the date result to text by using the CONVERT function with a date-format argument.

Retrieving the First Valid Date

The BEGINDATE function, which returns the first date for which an expression has a non-NA value.

Examples

Example 11–25 Finding the End Date

The following statements limit the values of the dimensions of the units variable, then sends the last date associated with a non-NA value to the current outfile.

```
LIMIT month TO ALL
LIMIT product TO 'Tents'
LIMIT district TO 'Chicago'
SHOW ENDDATE(units)
```

These statements produce the following output.

```
31DEC96
```

ENDOF

The ENDOF function returns the last date of a time period in dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR.

> **Note:** You can only use this function with dimensions of type DAY, WEEK, MONTH, QUARTER, or YEAR.You can*not* use this function for time dimensions that are implemented as hierarchical dimensions of type TEXT.

Return Value

DATE

Syntax

ENDOF(*dwmqy-dimension*)

Arguments

dwmqy-dimension

A dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR. When you have explicitly defined your own relation between dimensions of this type, you can use the name of this time relation here.

Notes

How ENDOF Works

ENDOF returns the last date of the time period that is first in the current status list of the *dwmqy-dimension*.

Phased or Multiple Periods

ENDOF is particularly useful when the dimension has a phase that differs from the default or when the time periods are formed from multiple weeks or years. For example, when the dimension has four-week time periods, the ENDOF function identifies the final date of a particular four-week period.

Format of the Date

When you display the result returned by ENDOF, the date is formatted according to the date template in the DATEFORMAT option. When the day of the week or the name of the month is used in the date template, the day names specified in the DAYNAMES option and the month names specified in the MONTHNAMES option are used. You can use the result returned by ENDOF anywhere that a DATE value is expected.

DATE-to-TEXT Conversion

You can also use the result where a text value is expected. The date is converted automatically to a text value, using the current template in the DATEFORMAT option to format the text value. When you want to override the current DATEFORMAT template, you can convert the date result to text by using the CONVERT function with a date-format argument.

Retrieving the First Date of a Time Period

The STARTOF function, which returns the first date of a time period.

Examples

Example 11–26   Finding the Fiscal Year End Date

The following statements define a year dimension (called `taxyear`, for a tax year that begins in July), add dimension values for tax years `1998` through `2000`, and produce a report showing the last date of each tax year.

```
DEFINE taxyear DIMENSION YEAR BEGINNING july
VNF 'TY<ffb>'
MAINTAIN taxyear ADD '01july98' '01july00'
REPORT W 14 ENDOF(taxyear)
```

These statements produce the following output.

```
TAXYEAR        ENDOF(TAXYEAR)
-------------- --------------
TY98           30JUN99
TY99           30JUN00
TY00           30JUN01
```

EQ

The EQ command specifies a new expression for an already defined formula. In order to assign an EQ to a formula definition, the definition must be the one most recently defined or considered during the current session. When it is not, you must first use a CONSIDER command to make it the current definition.

An alternative to the EQ command is the EDIT FORMULA command, which is available only in OLAP Worksheet. The EDIT FORMULA command opens an Edit window in which you can add, delete, or change the expression to be calculated for a formula.

Be sure to distinguish between the EQ command described here and the EQ operator used to compare values of the same type.

Syntax

EQ [*expression*]

Arguments

expression
The calculation that will be performed to produce values when you use the formula. When you do not specify an expression, the EQ command sets the expression to `NA`. The formula text is not converted to uppercase.

Notes

Using EQ

You can use EQ to add an expression to a formula that had no expression before, or to replace the old expression with a new one.

Data Type and Dimensions

The data type and dimensions of the new expression should match the specified data type and dimensions in the definition of the formula. When they do not, the resulting values are converted to the formula's data type and the results are forced into the formula's dimensionality. The DESCRIBE command shows the formula's data type and dimensions. You can find out the data type and dimensions of the new expression by parsing it. See Example 11–28, "Using PARSE with EQ" on page 11-59.

You cannot use the EQ command to change the data type or dimensions of a formula. To make changes in these, you must delete the formula and redefine it.

Examples

Example 11–27 Adding an EQ

This example specifies a new expression for the `f1` formula with the following definition.

```
DEFINE f1 FORMULA INTEGER <month line division>
EQ actual * 2
```

The statements

```
CONSIDER f1
EQ actual * 3
DESCRIBE f1
```

produce the following definition of the formula with a new EQ.

```
DEFINE F1 FORMULA INTEGER <MONTH LINE DIVISION>
EQ actual * 3
```

Example 11–28 Using PARSE with EQ

The following example supposes that your workspace already has a formula named `line.totals`. The PARSE and SHOW INFO (PARSE) statements check the dimensionality and data type of an expression. The CONSIDER and EQ statements assign the expression to the `line.totals` formula. The `line.totals` formulas has the following definition.

```
DEFINE line.totals FORMULA DECIMAL <year line>
```

The statements

```
PARSE 'total(actual line year)'
SHOW INFO(PARSE DIMENSION)
```

produce the following output.

```
YEAR
LINE
```

The statement

```
SHOW INFO(PARSE DATA)
```

produces the following output.

```
DECIMAL
```

The output from INFO(PARSE) shows that the expression has the same dimensionality and data type as the `line.totals` formula. The statements

```
CONSIDER line.totals
EQ TOTAL(actual line year)
DESCRIBE line.totals
```

show the definition of `line.totals` with its new EQ.

```
DEFINE LINE.TOTALS FORMULA DECIMAL <YEAR LINE>
EQ total(actual line year)
```

ERRNAMES

The ERRNAMES option controls whether the value of the ERRORTEXT option contains the name of the error (that is, the value of the ERRORNAME option) as well as the text of the error message.

Data type

BOOLEAN

Syntax

ERRNAMES = {NO|YES}

Arguments

NO
When you set ERRNAMES to `NO`, ERRORTEXT contains only the text of the error message.

YES
When you set ERRNAMES to `YES` (the default), ERRORTEXT contains the name and the text of the error message.

Notes

Related Statements
ERRORNAME, ERRORTEXT, and TRAP.

Examples

Example 11–29 ERRORTEXT Value Depending on ERRNAMES Setting

Suppose that you run the following program.

```
VARIABLE myint INTEGER
myint = 35/0
SHOW ERRORTEXT
```

When the value of ERRNAMES is set to `YES`, the program returns the following value for ERRORTEXT:

```
ERROR: (MXXEQ01) A division by zero was attempted.  (If you want NA to
  be returned as the result of a division by zero, set the DIVIDEBYZERO
  option to YES.)
```

When the value of ERRNAMES is set to `NO`, the program returns the following value for ERRORTEXT:

```
ERROR: A division by zero was attempted.  (If you want NA to be
  returned as the result of a division by zero, set the DIVIDEBYZERO
  option to YES.)
```

ERRORNAME

The ERRORNAME option holds the name of the first error that occurs when you execute a program or when you execute an OLAP DML statement.

Data type

TEXT

Syntax

ERRORNAME

Notes

ERRORNAME and TRAP

ERRORNAME is often used in programs in conjunction with the TRAP command for handling errors. See Example 11–30, "Using ERRORNAME with TRAP" on page 11-64.

ERRORTEXT Option

The text of the error whose name is contained in ERRORNAME is contained in the option ERRORTEXT.

ERRNAMES Option

The ERRNAMES option controls whether the value of the ERRORTEXT option contains the name of the error (that is, the value of the ERRORNAME option) as well as the text of the error message.

ERRORNAME and SIGNAL

You can create your own error conditions in a program with the SIGNAL command. SIGNAL sets ERRORNAME and ERRORTEXT to the values you specify.

SIGNAL PRGERR

You can use the special name `PRGERR` to communicate to a calling program that an error has occurred. The command `SIGNAL PRGERR` sets ERRORNAME to a blank value and passes an error condition to the calling program without causing another error message to be displayed. For information on using SIGNAL to pass an Oracle OLAP error up a chain of nested programs, see the TRAP command.

Oracle OLAP Error Messages

The analytic workspace has a list of Oracle OLAP error messages with their associated names. The messages are contained in the variable `_MSGTEXT`, which is dimensioned by `_MSGID`. To produce this list, execute the following statement.

```
REPORT WIDTH 60 _MSGTEXT
```

Examples

Example 11–30 Using ERRORNAME with TRAP

In a report program that uses a TRAP command to handle errors, you can use the SIGNAL command to send the appropriate error name to the current outfile.

```
DEFINE myreport PROGRAM
LD Monthly Report
PROGRAM
TRAP ON CLEANUP NOPRINT
PUSH month DECIMALS LSIZE PAGESIZE
LIMIT month TO LAST 1
   ...
POP month DECIMALS LSIZE PAGESIZE
RETURN
CLEANUP:
POP month DECIMALS LSIZE PAGESIZE
SIGNAL ERRORNAME ERRORTEXT
END
```

ERRORTEXT

The ERRORTEXT option holds the text of the first error message that occurs when you execute a program or a statement.

Note: The name of the error whose message is found in ERRORTEXT is contained in the option ERRORNAME. See the entries for ERRORNAME and TRAP for more information about handling Oracle OLAP errors. The ERRNAMES option controls whether the value of the ERRORTEXT option contains the name of the error (that is, the value of the ERRORNAME option) as well as the text of the error message.

Data type

TEXT

Syntax

ERRORTEXT

Examples

Example 11–31 ERRORTEXT with the SIGNAL Command

In a report program that uses a TRAP command to handle errors, you can use the SIGNAL command to send the appropriate error message to the current outfile.

```
DEFINE myreport PROGRAM
LD Monthly Report
PROGRAM
TRAP ON CLEANUP NOPRINT
PUSH month DECIMALS LSIZE PAGESIZE
LIMIT month TO LAST 1
   ...
POP month DECIMALS LSIZE PAGESIZE
RETURN
CLEANUP:
POP month DECIMALS LSIZE PAGESIZE
SIGNAL ERRORNAME ERRORTEXT
END
```

ESCAPEBASE

The ESCAPEBASE option specifies the type of escape that is produced by the INFILE keyword of the CONVERT function.

Syntax

ESCAPEBASE = '*escape-type*'

Arguments

escape-type

Specify `'d'` for decimal escape, `'x'` for hexadecimal escape.

The default escape type is decimal, which produces the integer value for a character in the following form.

```
'\dnnn'
```

A hexadecimal escape is the integer value for a character in the following form.

```
'\xnn'
```

Examples

For an example of using ESCAPEBASE with CONVERT to convert a text value to an escape sequence, see Example 9–22, "Converting Text Values to Escape Sequences" on page 9-61.

EVERSION

The EVERSION function returns a text value that specifies the internal Oracle OLAP build number.

Return Value

TEXT

Syntax

EVERSION

Notes

EVERSION and Major Releases

The build number in the output of the EVERSION function is not the Oracle Database version number. The EVERSION value does *not* change only with major releases of the database.

Examples

Example 11–32 Obtaining the Version Number

The following statement produces text output that indicates the Oracle OLAP build number.

```
SHOW EVERSION
```

This statement produces output like the following.

```
Oracle OLAP Build 80020
```

EVERY

The EVERY function returns `YES` when every value of a Boolean expression is `TRUE`. It returns `NO` when any value of the expression is `FALSE`.

See also: ANY, COUNT, and NONE

Return Value

BOOLEAN

Syntax

EVERY(*boolean-expression* [[STATUS] *dimensions*])

Arguments

boolean-expression
The Boolean expression whose values are to be evaluated.

STATUS
Can be specified when one or more of the dimensions of the result of the function are not dimensions of the Boolean expression. (See the description of the *dimensions* argument.) When you specify the STATUS keyword when this is not the case, Oracle OLAP produces an error.

In cases where one or more of the dimensions of the result of the function are not dimensions of the Boolean expression, the STATUS keyword might be *required* in order for Oracle OLAP to process the function successfully, or the STATUS keyword might provide a performance enhancement. See "The STATUS Keyword" on page 11-70.

dimensions
The dimensions of the result. By default, EVERY returns a single value. When you indicate one or more dimensions for the results, EVERY tests for `TRUE` values along the dimensions that are specified and returns an array of values. Each dimension must be either a dimension of *boolean-expression* or related to one of its dimensions. When it is a related dimension, you can specify the name of the relation instead of the dimension name. This enables you to choose which relation is used when there is more than one.

Notes

NA Values

When the Boolean expression involves an NA value, EVERY returns a YES or NO result when it can, as shown in Table 11–1, " YES and NO Values Returned by EVERY When the Boolean Expression Includes an NA Value"

Table 11–1 YES and NO Values Returned by EVERY When the Boolean Expression Includes an NA Value

Boolean Expression	Result
NA EQ NA	YES
NA NE NA	NO
NA EQ non-NA	NO
NA NE non-NA	YES
NA AND NO	NO
NA OR YES	YES

However, in cases where a YES or NO result would be misleading, EVERY returns NA. For example, when you test whether an NA value is greater than a non-NA value, EVERY returns NA.

The Effect of NASKIP

EVERY is affected by the NASKIP option. When NASKIP is set to YES (the default), EVERY ignores NA values and returns YES when every value of the expression that is not NA is TRUE and returns NO when any values are not TRUE. When NASKIP is set to NO, EVERY returns NA when any value of the expression is NA. When all the values of the expression are NA, EVERY returns NA for either setting of NASKIP.

Data with a Time Dimension

When *boolean-expression* is dimensioned by dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR, you can specify any other DAY, WEEK, MONTH, QUARTER, or YEAR dimension as a related *dimension.* Oracle OLAP uses the implicit relation between the dimensions. To control the mapping of one DAY, WEEK, MONTH, QUARTER, or YEAR dimension to another (for example, from weeks to months), you can define an explicit relation between the two dimensions and specify the name of the relation as the *dimension* argument to the EVERY function.

For each time period in the related dimension, Oracle OLAP tests the data values for all the source time periods that end in the target time period. This method is used regardless of which dimension has the more aggregate time periods.

The STATUS Keyword

When one or more of the dimensions of the result of the function are not dimensions of the Boolean expression, Oracle OLAP creates a temporary variable to use while processing the function. When you specify the STATUS keyword, Oracle OLAP uses the current status instead of the default status of the related dimensions for calculating the size of this temporary variable.

When the size of the temporary variable for the results of the function would exceed 2 gigabytes, you *must* specify the STATUS keyword in order for Oracle OLAP to execute the function successfully. When the dimensions of the Boolean expression are limited to a few values and are physically fragmented, you can specify the STATUS keyword to improve the performance of the function.

When you use EVERY with the STATUS keyword in an expression that requires going outside of the status for results (for example, with the LEAD or LAG functions or with a qualified data reference), the results outside of the status will be returned as `NA`.

Examples

Example 11–33 Testing for All-True Values by District

You can use the EVERY function to test whether each district's sales of sportswear have exceeded $50,000 in every month. To have the results dimensioned by district, specify `district` as the second argument to EVERY.

```
LIMIT product TO 'Sportswear'
REPORT HEADING 'Top Sales' EVERY(sales GT 50000, district)
```

The preceding statements produce the following output.

```
DISTRICT       Top Sales
-------------- ----------
Boston                 No
Atlanta               Yes
Chicago               Yes
Dallas                Yes
Denver                Yes
Seattle                NO
```

Example 11–34 Testing for All-True Values by Region

You might also want to find out the regions for which every district has sportswear sales that exceed $50,000 in every month. Since the `region` dimension is related to the `district` dimension, you can specify `region` instead of `district` as a dimension for the results of EVERY.

```
REPORT HEADING 'Top Sales' EVERY(sales GT 50000, region)
```

The preceding statement produces the following output.

```
REGION         Top Sales
-------------- ----------
East                   No
Central               Yes
West                   NO
```

EXISTS

The EXISTS function determines whether an object is defined in any attached workspace. The EXISTS function is useful in a program to test whether a definition exists before you try to use it.

Return Value

BOOLEAN

Syntax

EXISTS(*name-expression*)

Arguments

name-expression
A text expression that specifies the name you want to test.

Notes

Specifying More Than One Name

When *name-expression* contains the names of more than one object, EXISTS returns `NO` even when all the objects specified by *name-expression* exist in attached workspaces.

Examples

Example 11–35 Using EXISTS

This example tests whether the variable `actual` has been defined in any attached workspace. The statement

```
SHOW EXISTS('actual')
```

produces the following result.

```
YES
```

EXP

The EXP function returns *e* raised to the *n*th power, where *e* equals `2.71828183....`

Return Value

`NUMBER`

Syntax

EXP (*n*)

Arguments

n

The power by which you want to raise *e*.

Examples

The following example returns *e* to the 4th power.

```
SHOW EXP(4)
```

```
54.59815
```

12 EXPORT to FILEMOVE

This chapter contains the following OLAP DML statements:

- EXPORT
  - EXPORT (to EIF)
  - EXPORT (to spreadsheet)
- EXPTRACE
- EXTBYTES
- EXTCHARS
- EXTCOLS
- EXTLINES
- FCCLOSE
- FCEXEC
- FCOPEN
- FCQUERY
- FCSET
- FETCH
- FILECLOSE
- FILECOPY
- FILEDELETE
- FILEERROR
- FILEGET

- FILEMOVE

EXPORT

The EXPORT command copies workspace objects from your analytic workspace to an external file. You can use EXPORT to copy both data and object definitions from your workspace to an EIF file, or you can use it to copy an OLAP DML worksheet object to a spreadsheet file.

Because the syntax of the EXPORT command is different depending on whether it is being used to produce an EIF file or a spreadsheet file, two separate entries are provided:

- EXPORT (to EIF))
- EXPORT (to spreadsheet)

EXPORT (to EIF)

The EXPORT (to EIF) command copies data and definitions from your Oracle OLAP analytic workspace to an EIF file. EXPORT also copies all dimensions of the exported data, even when you do not specify them in the command. The status of the data's dimensions in Oracle OLAP determines which values are exported. For information on exporting data dimensioned by unnamed composites, see "Unnamed Composites" on page 12-9.

EXPORT (to EIF) is commonly used in conjunction with IMPORT (from EIF) to copy parts of one Oracle OLAP workspace to another. You export objects from the source workspace to an EIF file and then import the objects from the EIF file into the target workspace. The source and target workspaces can reside on the same platform or on different platforms. When you transfer an EIF file between computers, use a binary transfer to overcome file-format incompatibilities between platforms. The EIF file must have been created with the EIFVERSION set to a version that is less than or equal to the version number of the target workspace. See EIFVERSION for information about verifying the target version number.

Syntax

EXPORT *export_item* TO EIF FILE *file-id* [LIST] [NOPROP] -

   [NOREWRITE|REWRITE] [FILESIZE *n* [K, M, or G]] -

   [NOTEMPDATA] [NLS_CHARSET *charset-exp*]

where:

export_item is one of the following:

   name [AS *newname*]

   exp [SCATTER AS *scattername* [TYPE *scattertype*] [EXCLUDING (*concatbasedim* . . .)]

   exp AS *name* [EXCLUDING (*concatbasedim* . . .)]

   ALL

Arguments

name
The name of an analytic workspace object or option to be exported. You can list more than one name for export.

AS *newname*
Specifies a new name for the analytic workspace object or option. When you specify an expression, or a local variable, or a local valueset, then you must use AS *name* to provide a name for the object that IMPORT (from EIF) will use to receive the data

Important: You cannot rename dimensions.

exp
An expression to be computed and exported. You can list more than one name at a time for export.

SCATTER AS *scattername* [TYPE *scattertype*]
When you want to export a large multidimensional object that may require multiple passes to write into memory, then you can use SCATTER AS *scattername* to improve file I/O performance. You must first define one or two new single-dimension text variables (*scattername* and *scattertype*) and assign text values and their corresponding data types to *scattername.* When you use SCATTER AS *scattername*, this tells Oracle OLAP to export the multidimensional expression as separate variables in the *slices* you have specified in *scattername.* When each of the slice variables is to have the same data type, you can simply make *exp* have that data type, in which case you will not need to use TYPE *scattertype*.

EXCLUDING (*concatbasedim* . . .)
The EXCLUDING phrase applies only to a concat dimension that you specify with the *name* argument. The value you specify for *concatbasedim*, specifies the base dimensions of the concat that Oracle OLAP does not export.

ALL
Specifies that Oracle OLAP exports all the objects currently in the status of NAME (and, therefore, not necessarily all objects in the workspace).

TO EIF FILE
Indicates that you want to create an EIF file.

file-id
A text expression that represents the name of the file. The name must be in a standard format for a file identifier.

LIST
Sends to the current outfile the definition of each object as it begins to export it. For dimensions, EXPORT indicates the number of values being exported, and for composites, it lists the number of dimension value combinations. EXPORT also produces a message that shows the total number of bytes read every two minutes and at the end of the export procedure.

NOPROP
Prevents any properties that you have assigned to each object using a PROPERTY from being written to the EIF file.

NOREWRITE
REWRITE
Specifies whether EXPORT will overwrite the target file when it already exists. NOREWRITE (the default) leaves an existing target file intact and sends an error message to the current outfile. REWRITE causes EXPORT to replace the existing file with the new EIF file.

FILESIZE *n* [K|M|G]
Sets the maximum size of each component file (main file and extension files) for EIF files. When a file's size grows beyond the value of FILESIZE or the current disk or location becomes full, Oracle OLAP creates an EIF extension file. See"EIF Extension Files" on page 12-9.

FILESIZE affects component files created after it is set. Previous component files may have various sizes, determined by the FILESIZE setting at the time each one was created or by the size it reached when its disk was full.

When you do not specify `K`, `M`, or `G`, the value you specify for *n* is interpreted as bytes. When you specify `K`, `M`, or `G` after the value *n*, the value is interpreted as kilobytes, megabytes, or gigabytes, respectively.

You can set FILESIZE to any value between 81,920 bytes (80K) and 2,147,479,552 bytes (2G).

NOTEMPDATA
Prevents data in TEMP variables from being written to the EIF file.

NLS_CHARSET *charset-exp*
Specifies the character set that Oracle OLAP will use when exporting text data to the file specified by *file-id*. This allows Oracle OLAP to convert the data accurately into that character set. This argument must be the last one specified. When this argument is omitted, then Oracle OLAP exports the data in the database character set, which is recorded in the NLS_LANG option.

Notes

EIF Options

A number of options determine how EIF files are imported and exported. These options are listed in Table 12–1, " EIF Options" on page 12-7.

Table 12–1 EIF Options

Statements	Description
EIFEXTENSIONPATH	An option that contains a list of directory objects that identify the locations where EIF extension files should be created.
EIFNAMES	An option that contains a list of the names of all the objects imported by the most recent IMPORT (from EIF) command.
EIFSHORTNAMES	An option that controls the structure of the extension of EIF overflow (extension) file names.
EIFTYPES	An option that contains a list of the types of objects that are contained in the list produced by the EIFNAMES option.
EIFUPDBYTES	An option that controls the frequency of updates when you are using the IMPORT (from EIF) command with its UPDATE keyword.
EIFVERSION	Used with the EXPORT (to EIF) and IMPORT (from EIF) commands, an option that specifies the EIF version when copying data between different versions of Express Server or Oracle OLAP.

Relations

When you export a relation, EXPORT sends out the definition and the values in status for the related dimension as well as the dimensions of the relation.

Conjoint Dimensions

When you export a conjoint dimension, make sure that the status of the base dimensions and the status of the conjoint dimension match. Since there is an implicit relation between conjoint and base dimensions, Oracle OLAP exports the base dimensions with the conjoint dimension, but it cannot export all the conjoint dimension values in the current status when the related base values are not also in status.

Concat Dimensions

When you export a concat dimension without using the EXCLUDING phrase or when you implicitly export a concat because you are exporting a variable dimensioned by the concat, an expression that uses the concat, or a concat of which the concat is a component, then Oracle OLAP exports each component of the concat dimension. Oracle OLAP uses the current status of each simple or conjoint component dimension when exporting the component. It does not use the status of the concat dimension when exporting the simple or conjoint components.

When you export a concat dimension using the EXCLUDING phrase, then the definition of the concat dimension that Oracle OLAP exports does not include the base dimensions that you specify with the *concatbasedim* argument. When you also export a variable or expression that uses the concat dimension, then the definition of the exported expression or variable uses the altered concat dimension definition. Oracle OLAP does not export variable or expression values that correspond to the excluded base dimensions.

You cannot use the EXCLUDING phrase with the EXPORT ALL keyword.

Dimension Surrogates

When you export a dimension surrogate, Oracle OLAP also exports the dimension of the surrogate.

Reducing Workspace Size

When you have added and then deleted many objects or dimension values, you might want to use EXPORT (from EIF) in conjunction with the IMPORT (from EIF) command to remove extra space from your analytic workspace. You can make your workspace smaller, perhaps substantially so. To do this, use the EXPORT command with the ALL keyword to put all the data in an EIF file, create another workspace with a different name, and then import the EIF file into the new workspace. You can then delete the old workspace and refer to the new one with the same workspace alias that you used for the original one.

Preserving Conjoint Type

When you export a HASH, BTREE, or NOHASH conjoint dimension to an EIF file, the conjoint type is exported along with its definition in the EIF file. When you then import the conjoint dimension into an analytic workspace, Oracle OLAP preserves the conjoint type when you import into a new dimension or a dimension already using that conjoint type. When you import the dimension into an existing dimension that does not use the same conjoint type, Oracle OLAP does not preserve the original conjoint type that was saved in the EIF file.

Unnamed Composites

When you define variables or other objects with the SPARSE keyword in the dimension list, Oracle OLAP creates an unnamed composite that corresponds to the SPARSE dimension list. When you export or import an object with the unnamed composite in its definition, the composite is automatically exported or imported with the object. Because the unnamed composite is not a regular workspace object, you cannot import or export it independently.

EIF Extension Files

EIF extension file names have the structure *filename*.e*nnn*, where *nnn* is a three-digit number beginning with 001. For example, assume you have an EIF file named `export.eif`, the extension files are named `export.e001`, `export.e002`, and so on. You can set the extension to three characters by using the EIFSHORTNAMES option. Extension files are created in the same directory object as the original EIF file, unless you specify a different one with the EIFEXTENSIONPATH option.

Variable Segments Specified with SEGWIDTH

When you use the SEGWIDTH keyword of the CHGDFN command to specify the length of variable segments, segment information cannot be exported and imported automatically. You can save your SEGWIDTH settings by exporting the entire workspace, creating a new workspace, importing only the workspace objects into the new workspace, specifying segmentation, and then importing the variable data into the new workspace.

Duplicate Object Names

When you want to export two objects that have the same name from two different workspaces, you must rename one of them in the EIF file by exporting it with the AS keyword. Objects in an EIF file cannot have duplicate names.

Permission Programs: Copying to and from Analytic Workspaces When you export PERMIT_READ or PERMIT_WRITE programs which are hidden, they are empty when imported. Additionally, when you outfile PERMIT_READ or PERMIT_WRITE programs which are hidden, then they are empty when infiled.

> **Tip:** Rename PERMIT_READ and PERMIT_WRITE programs before using EXPORT to EIF or OUTFILE After copying the programs to an analytic workspace using IMPORT (from EIF) or INFILE.

TEXT and NTEXT

You can export and import TEXT and NTEXT values. Both data types can be exported to a single EIF file.

- Exported TEXT values are stored in the EIF file using the character set specified for the file in the EXPORT command.
- Exported NTEXT values are stored in the EIF file as NTEXT (UTF8 Unicode).
- NTEXT values imported into TEXT objects are converted into the database character set. This can result in data loss when the NTEXT values cannot be represented in the database character set.
- TEXT values imported into NTEXT objects are converted into the NTEXT (UTF8 Unicode) character set.

Examples

Example 12–1 Exporting Variables

Suppose you want to export the values in status and the dimensions of two variables called `actual` and `budget` from your current Oracle OLAP workspace to a disk file called `finance.eif` in your current directory object. Use the following statement.

```
EXPORT actual budget TO EIF FILE 'finance.eif'
```

Example 12–2 Exporting a Large Object

Suppose you want to export a large, multidimensional object that is likely to require multiple passes to write into memory. To improve file I/O performance, you can create a single-dimension variable to tell Oracle OLAP how to slice the multidimensional variable into smaller pieces.

Suppose the large object is the SALES variable, which is dimensioned by `month`, `product`, and `district`. To specify how `sales` should be sliced, create a single-dimension variable, as shown in the following statement.

```
DEFINE salescatter VARIABLE TEXT <district>
```

Because `salescatter` is dimensioned by `district`, this will tell Oracle OLAP to divide `sales` into `district` slices. Because `district` has six values, `sales` will be divided into six slices. Each slice must be named. To do so, assign values to each `district` in `salescatter`. You can then assign the appropriate data type to each slice, for example, by using a QDR (qualified data reference), when desired.

To export SALES, execute the following statement.

```
EXPORT sales SCATTER AS salescatter TYPE TYPEVAR -
   TO EIF FILE 'slice.eif'
```

To import the variables, specify which of the named slices you want, as in the following statement.

```
IMPORT dist1 dist2 dist3 dist4 dist5 dist6 -
   FROM EIF FILE 'slice.eif'
```

Alternatively, you can import all of the variables.

```
IMPORT ALL FROM EIF FILE 'slice.eif'
```

EXPORT (to spreadsheet)

The EXPORT (to spreadsheet) command copies an Oracle OLAP worksheet object that you have created to a spreadsheet file and automatically translate it into the appropriate format. An analytic worksheet's dimensions form the columns and rows of the spreadsheet file. The current status of these dimensions determines which part of a worksheet is exported.

You can also export an analytic worksheet to an EIF file as described in EXPORT (to EIF). EXPORT (to spreadsheet) is commonly used to copy part of your Oracle OLAP workspace into a file that can be read by other software, such as Lotus 1-2-3, or Symphony.

Syntax

EXPORT *worksheetname* TO {WKS|WK1|WRK|WR1|DIF} FILE *file-id* -

  [STATRANK] [NOREWRITE|REWRITE] [NLS_CHARSET *charset-exp*]

Arguments

worksheetname
An Oracle OLAP worksheet object that you have created. In any one EXPORT (to spreadsheet) command, you can export only one *worksheetname* to one spreadsheet file.

TO WKS
TO WK1
TO WRK
TO WR1
TO DIF
Indicates that you want to export an Oracle OLAP worksheet to a 1-2-3 file, version 1 (`WKS`) or version 2 (`WK1`); a Symphony file, version 1.0 (`WRK`) or version 1.1 (`WR1`); or a data interchange format file (`DIF`).

FILE *file-id*
A text expression that represents the name of the file you are creating. The name must be in a standard format for a file identifier.

STATRANK
Specifies that the row and column numbers exported with worksheet data should be the current status rankings of the `WKSROW` and `WKSCOL` dimensions.

NOREWRITE
REWRITE
NOREWRITE (the default) leaves the existing file intact and displays an error. When you specify REWRITE, EXPORT overwrites the target file when it already exists.

NLS_CHARSET *charset-exp*
Specifies the character set that Oracle OLAP will use when exporting text data to the worksheet file specified by *file-id*. This allows Oracle OLAP to convert the data accurately into that character set. For information about the character sets that you can specify, see the *Oracle Database Globalization Support Guide*. This argument must be the last one specified. When this argument is omitted, then Oracle OLAP exports the data in the database character set, which is recorded in the NLS_LANG option.

Examples

Example 12–3 Limiting Before Exporting

This example exports part of a pricing worksheet by limiting its dimensions, `WKSCOL` and `WKSROW`, before the EXPORT command.

```
LIMIT WKSCOL TO 2 TO 4
LIMIT WKSROW TO 3 TO 4
EXPORT pricing TO WRK FILE 'price1.wrk'
```

EXPTRACE

The EXPTRACE option controls whether system DML programs are traced when the PRGTRACE option is set to `YES`. The EXPTRACE option can be set to `YES` to help debug a user-defined program that calls system programs.

Data type

BOOLEAN

Syntax

EXPTRACE = {YES|NO}

Arguments

YES
All programs are traced, including system DML programs.

NO
System DML programs are not traced. Only programs other than system DML programs are traced.

Notes

How to Identify System DML Programs

Some OLAP DML statements are implemented as system DML programs. To send to the current outfile a list of system DML programs affected by EXPTRACE, issue the following statement.

```
SHOW AW(PROGRAM 'express')
```

Examples

Example 12–4 Tracing System DML Programs

After the following statements are issued, system DML programs such as LISTNAMES and ALLSTAT are traced in addition to user-defined programs.

```
PRGTRACE = YES
EXPTRACE = YES
```

EXTBYTES

The EXTBYTES function extracts a portion of a text expression.

Return Value

TEXT

Syntax

EXTBYTES(*text-expression* [*start* [*length*]])

Arguments

text-expression

The expression from which a portion is to be extracted. When *text-expression* is a multiline TEXT value, EXTBYTES preserves the line breaks in the returned value.

start

An integer that represents the byte position at which to begin extracting. The position of the first byte in *text-expression* is 1. When you omit this argument, EXTBYTES starts with the first byte.

length

An integer that represents the number of bytes to be extracted. When *length* is not specified, or exceeds the number of bytes from *start* to the end of *text-expression,* the part from *start* to the end of *text-expression* is extracted.

Notes

Single-Byte Characters

When you are using a single-byte character set, you can use the EXTCHARS function instead of the EXTBYTES function.

NTEXT Data Type

This function does not accept NTEXT arguments, because it is oriented toward byte-manipulation instead of character manipulation. It always returns values of type TEXT. When you must use this function on NTEXT values, use the CONVERT or TO_CHAR function to convert the NTEXT value to TEXT.

Examples

Example 12–5 Extracting Text Characters Using Bytes

This example shows how to extract portions of text from the TEXT value `'hellotherejoe'`.

- The statement

  ```
  SHOW EXTBYTES('hellotherejoe', 6, 5)
  ```

  produces the following output.

  ```
  there
  ```

- The statement

  ```
  SHOW EXTBYTES('hellotherejoe', 11)
  ```

  produces the following output.

  ```
  joe
  ```

EXTCHARS

The EXTCHARS function extracts a portion of a text expression.

Return Value

TEXT or NTEXT

Syntax

EXTCHARS(*text-expression* [*start* [*length*]])

Arguments

text-expression

The expression from which a portion is to be extracted. When *text-expression* is a multiline text value, EXTCHARS preserves the line breaks in the returned value.

When you specify a TEXT expression, the return value is TEXT. When you specify an NTEXT expression, the return value is NTEXT.

start

An integer that represents the character position at which to begin extracting. The position of the first character in *text-expression* is 1. When you omit this argument, EXTCHARS starts with the first character.

length

An integer that represents the number of characters to be extracted. When *length* is not specified, or exceeds the number of characters from *start* to the end of *text-expression,* the part from *start* to the end of *text-expression* is extracted.

Notes

multibyte Characters

When you are using a multibyte character set, you can use the EXTBYTES function instead of the EXTCHARS function.

Examples

Example 12–6 Extracting Text Characters

This example shows how to extract portions of text from the TEXT value `'hellotherejoe'`.

- The statement

  ```
  SHOW EXTCHARS('hellotherejoe', 6, 5)
  ```

  produces the following output.

  ```
  there
  ```

- The statement

  ```
  SHOW EXTCHARS('hellotherejoe', 11)
  ```

  produces the following output.

  ```
  joe
  ```

EXTCOLS

The EXTCOLS function extracts specified columns from each line of a multiline text value. The function returns a multiline text value that includes only the extracted columns.

Columns refer to the character positions in each line of a multiline text value. The first character in each line is in column one, the second is in column two, and so on.

Return Value

TEXT or NTEXT

Syntax

EXTCOLS(*text-expression* [*start* [*numcols*]])

Arguments

text-expression
The text expression from which the specified columns should be extracted. When *text-expression* is a multiline text value, the characters in the specified columns are extracted from each one of its lines.

When you specify a TEXT expression, the return value is TEXT. When you specify an NTEXT expression, the return value is NTEXT.

start
An`INTEGER`, between `1` and `4000`, that represents the column position at which to begin extracting. The column position of the first character in each line of *text-expression* is `1`.

numcols
An `INTEGER` that represents the number of columns to be extracted. When you do not specify *numcols*, EXTCOLS extracts all the characters from the starting column to the end of each line.

Notes

Number of Lines Returned

EXTCOLS always returns a text value that has the same number of lines as *text-expression,* though some of the lines may be empty.

Start Column Beyond the End of a Line

When you specify a starting column that is to the right of the last character in a given line in *text expression,* the corresponding line in the return value will be empty.

Numcols That Goes Beyond the End of a Line

When you specify a length that exceeds the number of characters that follow the starting position in a given line in *text expression,* the corresponding line in the return value will include only existing characters. EXTCOLS does not return spaces at the end of the line to fill in the missing columns.

Examples

Example 12–7 Extracting Text Columns

In this example, four columns are extracted from each line of `citylist`, starting from the second column.

```
DEFINE citylist VARIABLE TEXT
citylist = 'Boston\nHouston\nChicago'
```

- The statement

  ```
  SHOW citylist
  ```

  produces the following output.

  ```
  Boston
  Houston
  Chicago
  ```

- The statement

  ```
  SHOW EXTCOLS(citylist 2 4)
  ```

  produces the following output.

  ```
  osto
  oust
  hica
  ```

EXTLINES

The EXTLINES function extracts lines from a multiline text expression.

Return Value

TEXT or NTEXT

Syntax

EXTLINES(*text-expression* [*start* [*numlines*]])

Arguments

text-expression
A multiline text expression from whose values one or more lines are to be extracted.

When you specify a TEXT expression, the return value is TEXT. When you specify an NTEXT expression, the return value is NTEXT.

start
An integer that represents the line number at which to begin extracting. The position of the first line in *text-expression* is `1`. When you omit this argument, EXTLINES begins with line 1.

numlines
An integer representing the number of lines to be extracted. When you do not specify *numlines*, or when you specify a number greater than the number of lines from *start* to the end of *text-expression*, all the lines from *start* to the end of *text-expression* are copied.

Examples

Example 12–8 Extracting One Text Line

This example shows how to extract the second line from a multiline text value in a variable called `mktglist`. The `mktglist` variable has the following values.

```
Salespeople
Products
Services
```

The statement

```
SHOW EXTLINES(mktglist 2 1)
```

produces the following output.

```
Products
```

FCCLOSE

The FCCLOSE command closes a forecasting context. When Oracle OLAP closes a forecasting context, only data in the variables specified in the FCEXEC command remain available to applications. Oracle OLAP purges all other data, including temporary pages, associated with the forecast.

You must use the FCCLOSE command in combination with other OLAP DML statements as outlined in "Forecasting Programs" on page 1-16. For additional information about forecasting and forecasting methods, we suggest the latest editions of the books listed in "Further Reading on Forecasting" on page 12-33.

Syntax

FCCLOSE *handle-expression*

Arguments

handle-expression
An INTEGER expression that is the handle to forecast context previously opened using the FCOPEN function.

Examples

For a full example of a forecasting program, see Example 12–10, "A Forecasting Program". on page 43.

FCEXEC

The FCEXEC command executes a forecast based on the parameters options specified by the FCSET command for the forecast. The FCEXEC command implicitly loops over all the dimensions of the expression other than the time dimension.

You must use the FCEXEC command in combination with other OLAP DML statements as outlined in "Forecasting Programs" on page 1-16. For additional information about forecasting and forecasting methods, we suggest the latest editions of the books listed in "Further Reading on Forecasting" on page 12-33.

Syntax

FCEXEC *handle-expression* [*choice*] *time-series-expression*

where:

choice is one or more of the following:

TIME *time-dimension*

TRADINGDAYS *expression*

INTO *name*

SEASONAL *name*

SMSEASONAL *name*

BACKCAST

Arguments

handle-expression
An INTEGER expression that specifies the handle to a forecasting context previously opened using the FCOPEN function.

TIME *time-dimension*
The name of the time dimension. You do not have to specify this parameter when one dimension of the *time-series-expression* is of type DAY, WEEK, MONTH, QUARTER, or YEAR.

TRADINGDAYS *expression*
An INTEGER expression that specifies the number of business days in the unit of time of the time data type (that is, DAY, WEEK, MONTH, or YEAR) of the time-series-expression. By default the value is the total number of days in the unit of time.

INTO *name*
The name of the Oracle OLAP variable in which Oracle OLAP stores the forecast data. This variable must be dimensioned by the time dimension and any other dimensions of the *time-series-expression* that have more than one value in status. (This variable can have additional dimensions. However, in this case, when Oracle OLAP executes the forecast, it limits each of these additional dimensions to the first value in the dimension's status list.).

Important: When you do not specify INTO and the *time-series-expression* names an Oracle OLAP variable, Oracle OLAP populates the input variable with the output data of the forecast, thus overwriting the original data.

SEASONAL *name*
The name of the variable that Oracle OLAP populates with the data that represents seasonal factors. Oracle OLAP produces only one cycle of factors and stores these values into this variable beginning with the first time period in status. This variable must be dimensioned by the time dimension and any other dimensions of the *time-series-expression* that have more than one value in status. (This variable can have additional dimensions. However, in this case, when Oracle OLAP executes the forecast, it limits each of these additional dimensions to the first value in the dimension's status list.)

SMSEASONAL *name*
The name of the variable that Oracle OLAP populates with the data that represents smoothed seasonal factors. Oracle OLAP produces only one cycle of factors and stores these values into this variable beginning with the first time period in status; all other values are set to `NA`. This variable must be dimensioned by the time dimension and any other dimensions of the *time-series-expression* that have more than one value in status. (This variable can have additional dimensions. However, in this case, when Oracle OLAP executes the forecast, it limits each of these additional dimensions to the first value in the dimension's status list.)

BACKCAST

The BACKCAST keyword specifies that Oracle OLAP returns fitted historical data. Typically this data is available only for a subset of the historical periods (sometimes called the "fit window"). Oracle OLAP sets the value of the data that corresponds to the historical time periods that are outside of the fit window to `NA`.

Important: When you specify a value for BACKCAST and do not specify a value for INTO variable, Oracle OLAP populates the source variable with the backcasted data, thus overwriting the original data.

time-series-expression

An expression that specifies the data from which FCEXEC calculates values. The *time-series-expression* must be a numeric expression that is dimensioned by *time-dimension*. The *time-series-expression* may also be dimensioned by other dimensions. In this case, FCEXEC implicitly loops over all the dimensions of the expression other than the time dimension. The maximum status length of the *time-series-expression* is 5000.

Notes

Forecasting a Single Value

The FCEXEC command implicitly loops over all the dimensions of the time-series expression other than the time dimension. When you want to forecast only one value of a multidimensional time-series expression, then you must limit the status of all non-time dimensions to a single value before you execute the FCEXEC command.

Examples

For a full example of a forecasting program, see Example 12–10, "A Forecasting Program". on page 43.

FCOPEN

The FCOPEN function creates a forecasting context and returns a handle to this context.

You must use the FCOPEN function in combination with other OLAP DML statements as outlined in "Forecasting Programs" on page 1-16. For additional information about forecasting and forecasting methods, we suggest the latest editions of the books listed in "Further Reading on Forecasting" on page 12-33.

Return Value

INTEGER

Syntax

FCOPEN(*text-expression* [*prototype-handle*])

Arguments

text-expression
The name of the forecasting context.

prototype-handle
An INTEGER expression that is the handle to a different forecasting context that was previously-created using the FCOPEN function. Oracle OLAP initializes the new forecasting context with the same options as the forecasting context specified by this parameter. (See FCSET for descriptions of the options that specify the characteristics of a forecasting context.)

Examples

For a full example of a forecasting program, see Example 12–10, "A Forecasting Program". on page 43.

FCQUERY

The FCQUERY function queries the results of a forecast created when the FCEXEC command executed.

You must use the FCQUERY function in combination with other OLAP DML statements as outlined in "Forecasting Programs" on page 1-16. For additional information about forecasting and forecasting methods, we suggest the latest editions of the books listed in "Further Reading on Forecasting" on page 12-33.

Return Value

The return value depends on the option that you use as described in the tables for this entry.

Syntax

FCQUERY(HANDLELIST|*handle-expression option* -

[TRIAL *trial-num*] [CYCLE *cycle-num*])

Arguments

HANDLELIST
When you specify the HANDLELIST keyword, the FCQUERY function returns a multiline text expression that is a list of the handles to forecasting contexts that are currently open.

handle-expression
An INTEGER expression that is the handle to forecast context that you want to query and that was previously opened using the FCOPEN function.

option
The specific information that you want to retrieve:

- When you want information about the options specified for the entire forecast, do not use the TRIAL keyword. In this case, *option* can be any of the options that you can specify using the FCSET command and any of the options listed in

Table 12–2, " Options That You Can Specify for the Entire Forecast" on page 12-30.

- When you want information about a specific trial, use the TRIAL *trial-num* phrase. In this case, *option* can be any of the options listed in Table 12–3, " Options That You Can Specify for an Individual Trial" on page 12-30.

Table 12–2 Options That You Can Specify for the Entire Forecast

Keyword	Return type	Description
HANDLEID	TEXT	The name of the forecasting context when a value was specified when the forecasting context was opened using the FCOPEN command; or `NA` when no name was specified at that time.
TRIALSRUN	INTEGER	The number of trials for which data is available; or `NA` when no trials were run.

Table 12–3 Options That You Can Specify for an Individual Trial

Option	Return Value	Description
ALLOCLAST	BOOLEAN	Indicates whether the risk of over-adjustment should be reduced by allocating, instead of forecasting, the last cycle.
ALPHA	DOUBLE	The value of Alpha for this trial of the forecast. Alpha is the level or baseline parameter that is used for the Single Exponential Smoothing, Double Exponential Smoothing, and Holt-Winters forecasting methods.
BETA	DOUBLE	The value of Beta for this trial of the forecast. Beta is the trend parameter that controls the estimate of the trend. Beta is used for the Double Exponential Smoothing and Holt-Winters forecasting methods.
COMPSMOOTH	BOOLEAN	Indicates whether optimization should be done on the median smoothed data series.
CYCDECAY	DOUBLE	The value of the cyclic decay parameter for this trial of the forecast. Cyclical decay pertains to how seriously Oracle OLAP considers deviations from baseline activity when it performs linear and nonlinear regressions.

Table 12–3 (Cont.) Options That You Can Specify for an Individual Trial

Option	Return Value	Description
GAMMA	DOUBLE	The value of Gamma for this trial of the forecast. Gamma is the seasonal parameter that is used for the Holt-Winters forecasting method.
HISTUSED	INTEGER	The number of historical periods actually used, after all leading `NA` values are bypassed.
MAD	DOUBLE	The mean absolute deviation (MAD) for this trial of the forecast.
MAPE	DOUBLE	The mean average percent error (MAPE) for this trial of the forecast.
MAXFCFACTOR	DECIMAL	The upper bound of the forecast data.
METHOD	TEXT	The forecasting method that Oracle OLAP used for this trial of the forecast. See the METHOD option of the FCSET command for descriptions of the various methods.
MINFCFACTOR	DECIMAL	The lower bound of the forecast data.
MPTDECAY	DOUBLE	The value of the parameter that Oracle OLAP used when it adjusted the decay of estimates of base values that were used when it unraveled the predictions on the moving periodic total (MPT) series for this trial of the forecast.
NCYCLES	INTEGER	The number of cycles specified using the PERIODICITY argument to FCSET.
PERIODICITY	INTEGER	The length, in periods, of one or more cycles. The return value depends on the way you call the FCQUERY function: When you specify the CYCLE argument, PERIODICITY returns the number of periods in the specified cycle. When you do not specify the CYCLE argument and FCSET ALLOCLAST is `NO`, PERIODICITY returns the product of all cycle lengths. When you do not specify the CYCLE argument and FCSET ALLOCLAST is `YES`, PERIODICITY returns the product of all cycle lengths leaving out the length of the last (least aggregate) cycle.
RMSE	DOUBLE	The root mean squared error (RMSE) for this trial of the forecast.

Table 12–3 (Cont.) Options That You Can Specify for an Individual Trial

Option	Return Value	Description
SMOOTHING	BOOLEAN	Indicates whether Oracle OLAP smoothed the data for this trial of the forecast. `YES` indicates that Oracle OLAP smoothed the data; `NO` indicates that Oracle OLAP did not smooth the data.
TRANSFORM	TEXT	The data filter that Oracle OLAP used for this trial of the forecast. See the TRANSFORM option of the FCSET command for descriptions of the various filters.
TRENDHOLD	DOUBLE	The value of the trend hold parameter for this trial of the forecast. trend hold parameter that indicates trend reliability in Double Exponential Smoothing and Holt-Winters forecasting methods.

trial-num
An INTEGER expression that is the number of the trial for which you want to retrieve information.

cycle-num
An INTEGER expression that specifies a cycle for which you want information from the PERIODICITY option (see Table 12–3, " Options That You Can Specify for an Individual Trial" on page 12-30). When you specified a series of cycles using the PERIODICITY argument in the FCSET command, then the value of *cycle-num* indicates the position of the cycle of interest in the specified series. For example, assume that `FCSET PERIODICITY <52,7>` was specified. In this case, a *cycle-num* of 1 returns 52 and a *cycle-num* of 2 returns 7. When you did not specify a series of cycles using the PERIODICITY argument in the FCSET command, then it is unnecessary to specify this argument.

Notes

Using Options

You can retrieve information about the options specified for the entire forecast or information about a specific trial.

- When you want information about the options specified for the entire forecast, do not use the TRIAL keyword. In this case, *option* can be HANDLEID, TRIALSRUN, or any of the options that you can specify using the FCSET command.

- When you want information about a specific trial, use the TRIAL *trial-num* phrase. In this case, *option* can be ALPHA, BETA, CYCDECAY, GAMMA, MAD, MAPE, METHOD, MPTDECAY, RMSE, SMOOTHING, TRANSFORM, or TRENDHOLD.

Accessing Dimensioned Data

When more than one time series was in status when the FCEXEC command was executed, then the TRIALSRUN and the NTRIAL-dimensioned data are also be dimensioned by the extra dimensions of the time-series expression. Although Oracle OLAP treats the value returned by the FCQUERY function as a scalar expression, you can access its dimensioned data in any of the following ways:

- In a FOR loop, FCQUERY returns data for the current values of the FOR dimensions
- In a QUAL function, FCQUERY returns data for the specified values of the qualified dimensions.
- In all other cases, FCQUERY returns data for the first value in status of each of its dimensions.

Further Reading on Forecasting

For additional information about forecasting and forecasting methods, we suggest the latest editions of the following books.

- Levenbach, Hans, and Cleary, James P. *The Beginning Forecaster*. Belmont, CA: Lifetime Learning Publications.
- Mosteller, Frederick, and Tukey, John W. *Data Analysis and Regression*. Reading, MA: Addison-Wesley Publishing Co. Inc.
- Makridakis, Spyros, and Wheelwright, Steven C. *Interactive Forecasting*. San Francisco, CA: Holden-Day Inc.

Examples

Example 12–9 Querying a Forecast

The `autofcst` program illustrated in Example 12–10, "A Forecasting Program". on page 43 calls a program named `queryall`. The `queryall` program retrieves the characteristics of the trials of the forecast using the following code.

```
DEFINE queryall PROGRAM
PROGRAM
VARIABLE numtrials INTEGER
VARIABLE loopindx INTEGER
numtrials = FCQUERY(hndl trialsrun)
row numtrials 'TRIALS'
loopindx = 1
WHILE loopindx LE numtrials
  DO
    ROW loopindx 'METHOD' FCQUERY(hndl method trial loopindx)
    ROW loopindx 'TRANSFORM' FCQUERY(hndl transform trial loopindx)
    ROW loopindx 'SMOOTHING' FCQUERY(hndl smoothing trial loopindx)
    ROW loopindx 'ALPHA' FCQUERY(hndl alpha trial loopindx)
    ROW loopindx 'BETA' FCQUERY(hndl beta trial loopindx)
    ROW loopindx 'GAMMA' FCQUERY(hndl gamma trial loopindx)
    ROW loopindx 'TRENDHOLD' FCQUERY(hndl trendhold trial loopindx)
    ROW loopindx 'CYCDECAY' FCQUERY(hndl cycdecay trial loopindx)
    row loopindx 'MPTDECAY' FCQUERY(hndl mptdecay trial loopindx)
    ROW loopindx 'MAD' FCQUERY(hndl mad trial loopindx)
    ROW loopindx 'MAPE' FCQUERY(hndl mape trial loopindx)
    ROW loopindx 'RMSE' FCQUERY(hndl rmse trial loopindx)
    loopindx = loopindx + 1
  DOEND
END
```

A sample report created from the output of the QUERYALL program follows.

```
3 TRIALS
1 METHOD     HOLT/WINTERS
1 TRANSFORM  TRNOSEA
1 SMOOTHING          NO
1 ALPHA             0.2
1 BETA              0.3
1 GAMMA             0.3
1 TRENDHOLD         0.8
1 CYCDECAY           -1
1 MPTDECAY           -1
1 MAD         324.97047
1 MAPE       23.6192147
1 RMSE        389.40202
2 METHOD     HOLT/WINTERS
2 TRANSFORM  TRNOSEA
2 SMOOTHING          NO
2 ALPHA             0.2
2 BETA              0.3
2 GAMMA             0.2
2 TRENDHOLD         0.8
2 CYCDECAY           -1
2 MPTDECAY           -1
2 MAD         324.97047
2 MAPE       23.6192147
2 RMSE        389.40202
3 METHOD     HOLT/WINTERS
3 TRANSFORM  TRNOSEA
3 SMOOTHING          NO
3 ALPHA             0.2
3 BETA              0.3
3 GAMMA             0.1
3 TRENDHOLD         0.8
3 CYCDECAY           -1
3 MPTDECAY           -1
3 MAD         324.97047
3 MAPE       23.6192147
3 RMSE        389.40202
```

FCSET

You must use the FCSET command in combination with other OLAP DML statements as outlined in "Forecasting Programs" on page 1-16. For additional information about forecasting and forecasting methods, we suggest the latest editions of the books listed in "Further Reading on Forecasting" on page 12-33.

Syntax

FCSET *handle-expression*

where

handle expression is one of the following:

> ALLOCLAST {YES | <u>NO</u>}
> ALPHA {MAX | MIN | STEP} *decimal*
> APPROACH {'APPAUTO' | 'APPMANUAL' | 'APPHYBRID'}
> BETA {MAX | MIN | STEP} *decimal*
> COMPSMOOTH {YES | <u>NO</u>}
> CYCDECAY {MAX | MIN} *decimal*
> GAMMA {MAX | MIN | STEP} *decimal*
> HISTPERIODS *integer*
> MAXFACTOR *decimal*
> METHOD *method-text-expression*
> MINFCFACTOR *decimal*
> MPTDECAY {MAX | MIN} *decimal*
> NTRIALS *integer*
> PERIODICITY *cycle-spec*
> RATIO *decimal*
> SMOOTHING {<u>YES</u> | NO}
> TRANSFORM {'TRNOSEA' | 'TRSEA' | 'TRMPT'}
> TRENDHOLD {MAX | MIN | STEP} *decimal*
> WINDOWLEN *integer*

Arguments

ALLOCLAST {<u>NO</u>|YES}
Indicates whether the risk of over-adjustment should be reduced by allocating, instead of forecasting, the last cycle.

- **NO** specifies that the last cycle should be forecast. (Default)
- **YES** specifies that only the average value for one period of the cycle is forecast. That average value is then multiplied by factors to give the remaining points in that period. For example, when the last cycle has 24-hour periods, only an average hourly value is forecast, which is then multiplied by 24 hourly factors to give the value for each hour.

ALPHA {MAX|MIN|STEP} *decimal*
Specifies the value for Alpha.

- **MAX** specifies the maximum value of Alpha. Alpha is the level or baseline parameter that is used for the Single Exponential Smoothing, Double Exponential Smoothing, and Holt-Winters forecasting methods. You can specify any decimal value from 0.0 through 1.0. The default value is 0.3.
- **MIN** specifies the minimum value of Alpha. You can specify any decimal value from 0.0 through 1.0. The default value is 0.1.
- **STEP** specifies the value of the interval that Oracle OLAP uses when it determines the value of Alpha. You can specify any decimal value from 0.05 through 0.2 as long as the value evenly divides the difference between ALPHA MAX and ALPHA MIN. The default value is 0.1.

APPROACH {'APPAUTO'|'APPMANUAL'|'APPHYBRID'}
Specifies the approach that Oracle OLAP takes when the it executes the forecast.

- **'APPAUTO'** indicates that Oracle OLAP tests all of the possible models and options for these models and chooses and uses the model that best fits the data. (Default)
- **'APPMANUAL'** indicates that Oracle OLAP creates a forecast using the values specified in the FCSET commands for this forecasting context.
- **'APPHYBRID'** indicates that, using the options that are specified in the FCSET commands for this forecasting context as the base options, Oracle OLAP tests all of the possible models and options for these models and chooses and uses the model that best fits the data.

BETA {MAX|MIN|STEP} *decimal*
Specifies the value of Beta.

- **MAX** specifies the maximum value of Beta. Beta is the trend parameter that controls the estimate of the trend. Beta is used for the Double Exponential Smoothing and Holt-Winters forecasting methods. You can specify any decimal value from 0.0 through 1.0. The default value is 0.3.

- **MIN** specifies the minimum value of Beta. You can specify any decimal value from 0.0 through 1.0. The default value is 0.1.
- **STEP** specifies the value of the interval that Oracle OLAP uses when it determines the value of Beta. You can specify any decimal value from 0.05 through 0.2 as long as the value evenly divides the difference between BETA MAX and BETA MIN. The default value is 0.1.

COMPSMOOTH {YES|<u>NO</u>}
Indicates whether optimization should be done on the median smoothed data series.

- **NO** specifies that the methods are done using the original historical time series data. (Default)
- **YES** specifies that optimization is done on the median smoothed data series, which results in more smoothed or "baseline" forecasts.

CYCDECAY {MAX|MIN} *decimal*
Specifies the value of the cyclical decay.

- **MAX** specifies the maximum value of the cyclical decay parameter. Cyclical decay pertains to how seriously Oracle OLAP considers deviations from baseline activity when it performs linear and nonlinear regressions. You can specify any decimal value from 0.2 through 1.0 as long as the difference between CYCDECAY MIN and CYCDECAY MAX is evenly divided by 0.4. The default value is 1.0.
- **MIN** specifies the minimum value of the cyclical decay parameter. You can specify any decimal value from 0.2 through 1.0 as long as the difference between CYCDECAY MIN and CYCDECAY MAX is evenly divided by 0.4. The default value is 0.2.

GAMMA {MAX|MIN|STEP} *decimal*
Specifies the value of Gamma.

- **MAX** specifies the maximum value of Gamma. Gamma is the seasonal parameter that is used for the Holt-Winters forecasting method. You can specify any decimal value from 0.0 through 1.0. The default value is 0.3.
- **MIN** specifies the minimum value of Gamma. You can specify any decimal value from 0.0 through 1.0. The default value is 0.1.
- **STEP** specifies the value of the interval that Oracle OLAP uses when it determines the value of Gamma. You can specify any decimal value from 0.05

through 0.2 as long as the value evenly divides the difference between GAMMA MAX and GAMMA MIN. The default value is 0.1.

HISTPERIODS *integer*
The number of historical periods. You can specify any integer value from `1` through `50000`, which is the maximum number of time dimension values that can be present in the *time-series* expression specified in the FCEXEC command.

MAXFCFACTOR *decimal*
Specifies the upper bound on the forecast data. The number you specify indicates a multiple of the largest value in the historical series. For example, when you specify `10.0`, the upper bound will be 10 times the largest value in the historical series. The default value is `100.0`.

METHOD '*method*'
Specifies the method that you want Oracle OLAP to use. You can specify one of the following keywords for *method*:

- **AUTOMATIC** specifies that Oracle OLAP should determine and use the method that is the best fit for the data. (Default)
- **LINREG** specifies the **linear regression** method in which a linear relationship `(y=a*x+b)` is fitted to the data.
- **NLREG1** specifies the **nonlinear regression method 1** in which a linear relationship `(y'=a*x'+b)` is fitted to a transformation of the original data; in this case, `x'=log(x) and y'=log(y)`. This results in the development of a polynomial model between `x` and `y(y=c*x^a)`.
- **NLREG2** specifies the **nonlinear regression method 2** in which a linear relationship `(y'=a*x'+b)` is fitted to a transformation of the original data; in this case, `x'=x and y'=ln(y)`. This results in the development of an exponential model between `x` and `y(y=c*e^ax)`.
- **NLREG3** specifies the **nonlinear regression method 3** in which a linear relationship `(y'=a*x'+b)` is fitted to a transformation of the original data; in this case, `x'=log(x)` and `y'=y`. This results in the development of a logarithmic model between `x` and `y(y=a*log(x)+b)`.
- **NLREG4** specifies the **nonlinear regression method 4** in which a linear relationship `(y'=a*x'+b)` is fitted to a transformation of the original data; in this case, `x'=1/x and y'=1/y`. This results in the development of an asymptotic curve `(y=x/(a+bx))`.

- **NLREG5** specifies the **nonlinear regression method 5** in which a linear relationship `(y'=a'*x+b)` is fitted to a transformation of the original data; in this case, `x'=x and y'=ln(y/(K-y))`. This results in the development of an exponential asymptotic curve `(y=cKe^ax/(1+ce^ax))`.
- **SESMOOTH** specifies the **single exponential smoothing** method in which the current estimate is taken as a geometrically weighted average of past values, and all future values are given this same value. This method is intended for short term forecasts of non-seasonal data.
- **DESMOOTH** specifies the **double exponential smoothing** method in which the current estimate is taken as a geometrically weighted average of past values, and this is added to a trend term calculated by the same method. Single exponential smoothing is therefore applied to both the series and the trend term.
- **HOLT/WINTERS** specifies the **Holt-Winters** method that is used on seasonal data, in which double exponential smoothing methods with trend damping are combined with multiplicative seasonal factors, which are estimated using single exponential smoothing.

MINFCFACTOR *decimal*
Specifies the lower bound on the forecast data. The number you specify indicates a multiple of the smallest value in the historical series. You can specify any decimal value from `0.0` through `1.0`. For example, when you specify `0.5` the lower bound will be half the smallest value in the historical series. The default value is `0.0`.

MPTDECAY {MAX|MIN} *decimal*
Specifies the value of the parameter that Oracle OLAP uses when it adjusts the decay of estimates of base values that it uses when it unravels the predictions on a moving periodic total (MPT) series.

- **MAX** specifies the maximum value of the parameter that Oracle OLAP uses when it adjusts the decay of estimates of base values that it uses when it unravels the predictions on a moving periodic total (MPT) series. You can specify any decimal value from `0.2` through `1.0` as long as the difference between MPTDECAY MIN and MPTDECAY MAX is evenly divided by `0.4`. The default value is `1.0`.
- **MIN** specifies the minimum value of the parameter that Oracle OLAP uses when it adjusts the decay of estimates of base values that it uses when it unravels the predictions on a moving periodic total (MPT) series. You can specify any decimal value from `0.2` through `1.0` as long as the difference between

MPTDECAY MIN and MPTDECAY MAX is evenly divided by `0.4`. The default value is `0.2`.

NTRIALS *integer*
Specifies the number of trials that Oracle OLAP runs to determine the forecast. You can specify any integer value from `1` through `3`. The default value is `3`.

PERIODICITY *cycle-spec*
Specifies either the number of periods for a single cycle or the number of periods in each of a set of nested cycles.

You do not have to specify this parameter when you are using a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR. In this case, Oracle OLAP derives the periodicity from the number of time dimension periods that constitute a year (for example, there are 26 WEEK periods in a year).

When you are not using a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR, the default value for *cycle-spec* is `1`, which specifies that the data is not grouped at all (that is, each period is logically independent).

Cycles are groupings of time periods that repeat through the time span of the data. For example, daily periods can be grouped into a weekly cycle and weekly periods can be grouped into a yearly cycle. In this case, the cycles are said to be nested, with the yearly cycle more aggregate than the weekly cycle, and the weekly cycle more detailed than the yearly cycle. By specifying cycles at a more detailed level, you allow OLAP to conduct a finer-grained search for factors that affect the data.

- To specify a single cycle, set *cycle-spec* to an integer from `1` through `25000`. The integer indicates the number of periods into which the cycle should be divided. For example, the INTEGER `12` specifies that the cycle should be divided into 12 periods.
- To specify a series of nested cycles, set *cycle-spec* to a series of up to six integers enclosed in parentheses and separated by commas. Each value in the series is the number of periods in one of the nested cycles. The cycles are ordered from most aggregate to least aggregate. For example, when *cycle-spec* is `(52,7)`, this indicates two cycles in which the most aggregate cycle is divided into 52 periods and each of those periods is divided into seven periods. In this example, the year is divided into 52 weeks, and each of those weeks is divided into seven days.

RATIO *decimal*
Specifies the ratio of the size of the window that Oracle OLAP uses for smoothing and the total number of historical periods. Oracle OLAP uses this value to

determine the number of backcast periods. You can specify any decimal value from `1/26` through `1/2`. The default value is `1/3`.

SMOOTHING {YES|NO}

Indicates whether Oracle OLAP should smooth the data for the forecast. The default value is `NO`. Specify `YES` when you want Oracle OLAP to smooth the data.

TRANSFORM {'TRNOSEA'|'TRSEA'|'TRMPT'}

The data filter that Oracle OLAP uses when executing the forecast.

- **'TRNOSEA'** indicates that Oracle OLAP will not seasonally adjust the data. (Default)
- **'TRSEA'** indicates that Oracle OLAP will transform using a filter that seasonally adjusts the data.
- **'TRMPT'** indicates that Oracle OLAP will transform using a moving periodic total (MPT) filter.

TRENDHOLD {MAX|MIN|STEP} *decimal*

Specifies the value of the trend.

- **MAX** specifies the maximum value of the trend hold parameter that indicates trend reliability in Double Exponential Smoothing and Holt-Winters forecasting methods. You can specify any decimal value from `0.0` through `1.0`. The default value is `0.8`.
- **MIN** specifies the minimum value of the trend hold parameter. You can specify any decimal value from `0.0` through `1. 0`. The default value is `0.4`.
- **STEP** specifies the value of the interval that Oracle OLAP uses when it determines the value of the trend hold parameter. You can specify any decimal value from `0.1` through `0.2`. The value of TRENDHOLD STEP must evenly divide the difference between TRENDHOLD MAX and TRENDHOLD MIN. The default value is `0.2`

WINDOWLEN *integer*

Specifies the number of points that Oracle OLAP uses when it determines median values when it performs median smoothing. Median smoothing eliminates extreme variations in the data by replacing each data point in a series by the median value of itself and its neighbors. You can specify any integer value from `1` through `13`. The default value is `3`.

Examples

Example 12–10 A Forecasting Program

Suppose you define a program named `autofcst` program to perform a forecast from the data that is in an input variable named `fcin1`. The `fcin1` variable is dimensioned by a time dimension named `timedim`. Assume that you have defined a program named `autofcst` with the following definition and specification.

```
DEFINE autofcst PROGRAM
PROGRAM
" Using the Automatic forecasting method
" Suppose you want to create a forecast from the data in
" an input variable named fcin1 that is dimensionsed by
" a time dimension named timedim.
"
" Open a forecasting context
hndl = FCOPEN('MyForecast')
" Initialize the target variables
fcout1 = NA
fcseas1 = NA
fcsmseas1 = NA
" Specify that the forecast be of the AUTOMATIC type
fcset hndl method 'automatic'
" Execute the forecast
FCEXEC hndl time timedim INTO fcout1 -
      seasonal fcseas1 smseasonal fcsmseas1 backcast fcin1
" Create a report showing the input and output of the forecast
REPORT DOWN timedim fcin1 fcout1 fcseas1 fcsmseas1
" Run a program named queryall to retrieve the characteristics
" of the forecasting trials
QUERYALL
" Close the forecasting context
FCCLOSE hndl
END
```

The `autofcst` program opens a forecasting context, sets the option of the forecast to AUTOMATIC, reports on the forecasted data, and queries and reports the characteristics of the various trials that Oracle OLAP performed to determine the method to use, and closes the forecasting context.

The `autofcst` program contains the following report command that displays a report of the input to and the output from the forecast.

```
REPORT DOWN timedim fcin1 fcout1 fcseas1 fcsmseas1
```

The sample report created by this statement follows.

```
TIMEDIM           FCIN1      FCOUT1    FCSEAS1   FCSMSEAS1
-------------- ---------- ---------- ---------- ----------
Jan97                  NA         NA 1.06725482 1.02926773
Feb97                  NA         NA .978607917 .945762221
Mar97                  NA         NA 1.12699278 .860505188
Apr97                  NA         NA .576219022 .905284834
May97                  NA         NA .920601317 .907019312
Jun97                  NA         NA 0.91118344  1.0580697
Jul97                  NA         NA 1.07886483 1.05597234
Aug97                  NA         NA 1.08101034   1.054612
Sep97                  NA         NA 1.08077427 1.05361672
Oct97               2,914         NA 1.08351799 1.05380407
Nov97               2,500         NA 1.01126778 1.04504316
Dec97               2,504         NA 1.08370549 1.03104272
Jan98               3,333         NA         NA         NA
Feb98               2,512         NA         NA         NA
Mar98               2,888         NA         NA         NA
...                 ...          ...        ...        ...
Jan01                  NA 3,371.7631         NA         NA
Feb01                  NA 2,736.4811         NA         NA
Mar01                  NA 3,408.3656         NA         NA
Apr01                  NA 714.277175         NA         NA
May01                  NA 2,502.9315         NA         NA
Jun01                  NA 3,195.3626         NA         NA
Jul01                  NA 3,911.6058         NA         NA
Aug01                  NA  4,000.651         NA         NA
Sep01                  NA 4,220.2658         NA         NA
Oct01                  NA 3,416.0208         NA         NA
Nov01                  NA 2,827.3943         NA         NA
Dec01                  NA 2,990.8629         NA         NA
```

The `queryall` program and a sample report created from its output is shown in "Querying a Forecast" on page 12-34.

FETCH

The FETCH command specifies how analytic workspace data is retrieved for use in the relational table created by the `OLAP_TABLE` function which you use to access analytic workspace data using SQL.

You can only use the FETCH command in the *OLAP_command* parameter of the `OLAP_TABLE` function; you cannot use it in any other context. For more information on the `OLAP_TABLE` function, see the *Oracle OLAP Reference*.

Within the `OLAP_TABLE` function, the FETCH keyword specifies explicitly how analytic workspace data is mapped to a table object. The FETCH keyword is provided for Express applications that are migrating to the Oracle Database.

Note: Use the FETCH keyword in OLAP_TABLE only when you are upgrading an Express application that used the FETCH command for SNAPI. When you are upgrading an Express application, note that the syntax is the same here as in Express 6.3. You can use the same FETCH commands that you used previously.

When using FETCH as an argument in `OLAP_TABLE`, you must enter the entire statement on one line, without line breaks or continuation marks of any type.

To fetch or import data from an relational table into analytic workspace objects using SQL commands embedded in the OLAP DML, use the OLAP DML SQL command.

Syntax

FETCH *expression*... [TAG *tag-exp*] [LABELED] [*data-order*]

where:

data-order is one of the following:

USING <*order-dim*...>
ACROSS *across-dim*...
DOWN *down-dim*...
ACROSS *across-dim*... DOWN *down-dim*...

Arguments

expression...
One expression for each target column, in the same order they appear in the row definition. Separate expressions with spaces or commas.

TAG *tag-exp*
This keyword is ignored; it is retained in the syntax only for backward compatibility.

LABELED
This keyword is ignored; it is retained in the syntax only for backward compatibility. All fetches are labeled.

USING <*order-dim...*>
Orders the data block according to the dimension list specified in <*order-dim...*>. Specify dimensions or composites or a combination of the two within angle brackets. Dimensions are ordered from fastest to slowest varying, with the first dimension being the fastest varying. When you specify a USING clause, then you cannot specify ACROSS or DOWN.

ACROSS *across-dim...*
Orders the data block in columns and rows and specifies the column dimensions. For *across-dim*, specify a list of one or more dimensions, composites, the NONE keyword, or a combination of these. When you specify two or more ACROSS dimensions, then they vary from slowest to fastest, with the first dimension being the slowest.

When you specify ACROSS but not DOWN, then all unspecified dimensions default to DOWN dimensions, which vary from fastest to slowest in the order that the dimensions appear in the object definitions. However, adding the NONE keyword to the ACROSS dimension list fetches only the first value in status for the unspecified DOWN dimensions.

When you specify an ACROSS clause, then you cannot specify a USING clause.

DOWN *down-dim...*
Orders the data block in columns and rows and specifies the row dimensions. For *down-dim*, specify a list of one or more dimensions, composites, the `NONE` keyword, or a combination of these. When you specify two or more `DOWN` dimensions, then they vary from slowest to fastest, with the first dimension being the slowest.

When you specify DOWN but not ACROSS, then all unspecified dimensions default to ACROSS dimensions, which vary from fastest to slowest in the order that the dimensions appear in the object definitions. However, adding the NONE keyword to the DOWN dimension list fetches only the first value in status for the unspecified ACROSS dimensions.

When you specify a DOWN clause, you cannot specify a USING clause.

Notes

Default Data Order

When you do not specify a USING or DOWN/ACROSS clause, the dimensions of the data vary from fastest to slowest in the order they are listed in the workspace object definitions.

Using Expressions with Different Dimensionality

When you specify multiple expressions with different dimensionality in one FETCH command, the ordering of the dimensions from fastest to slowest varying is not predictable.

Maximum Size of Data Block

You can use MAXFETCH to set an upper limit on the size of a data block generated by FETCH.

Variables Defined with Composites

For variables defined with composites, you can specify the composites in place of the base dimensions in the ACROSS, DOWN, and USING clauses of FETCH. This minimizes the number of NA fields in the resulting data block. When a variable has been defined with a named composite, you can specify the name of the composite after the USING, DOWN or ACROSS keyword. You specify unnamed composites with the syntax used to define them. For example, a variable d.sales with the following definition

```
DEFINE d.sales VARIABLE DECIMAL <month SPARSE<product district>>
```

could be fetched with the expression `SPARSE<product district>` immediately following a USING, DOWN, or ACROSS keyword.

Performance Tip for Variables Dimensioned by Composites

By default, when FETCH explicitly loops over a composite, it sorts the composite values according to the current order of the values in the composite's base dimensions. The task of sorting requires some processing time, so when variables are large, performance can be affected. When your variable is very large, and you are more concerned about performance than about the order in which FETCH output is produced, you can set the SORTCOMPOSITE option to NO.

Examples

For an example of using FETCH in `OLAP_TABLE`, see the *Oracle OLAP Reference*.

FILECLOSE

The FILECLOSE command closes an open file. When the file has not been opened, an error occurs.

Syntax

FILECLOSE *fileunit*

Arguments

fileunit

An INTEGER fileunit number assigned to an open file by a previous call to the FILEOPEN function or by an OUTFILE command.

Notes

LOG Command

You must use the LOG command with the EOF keyword, rather than FILECLOSE, to close a file that was opened with the LOG command.

Examples

Example 12–11 Program That Opens and Closes a File

Suppose you have a program called READFILE that takes a file name as its first argument. The following lines from the program open the file and then close it.

```
fil.unit = FILEOPEN(arg(1), read)
   ... (Commands to read and process data)
FILECLOSE fil.unit
```

FILECOPY

The FILECOPY command copies the contents of one file (the source file) to another file (the target file). When the target file already exists, the file is overwritten with the copy.

Syntax

FILECOPY *source-file-name target-file-name*

Arguments

source-file-name
A text expression specifying the name of the file you want to copy from. Unless the file is in the current directory, you must include the name of the directory object in the name of the file.

> **Note:** Directory objects are defined in the database, and they control access to directories and file in those directories. You can use the CDA command to identify and specify a current directory object. Contact your Oracle DBA for access rights to a directory object where your database user name can read and write files.

target-file-name
A text expression specifying the name of the file you want to copy to. Unless the file is in the current directory, you must include the name of the directory object in the name of the file.

Examples

Example 12–12 Copying a File

The following statement copies the file `log.txt` from your session's current directory object to file `oldlog.txt` in the same directory.

```
FILECOPY 'log.txt' 'oldlog.txt'
```

FILEDELETE

The FILEDELETE command deletes a file from the operating system disk space.

Syntax

FILEDELETE *file-name*

Arguments

file-name
A text expression specifying the name of the file you want to delete. Unless the file is in the current directory, you must include the name of the directory object in the name of the file.

Note: Directory objects are defined in the database, and they control access to directories and file in those directories. You can use the CDA command to identify and specify a current directory object. Contact your Oracle DBA for access rights to a directory object where your database user name can read and write files.

Notes

DELETE and FILEDELETE

The FILEDELETE command differs from the DELETE command in that FILEDELETE deletes a file from the operating system, while DELETE deletes an object in a database.

Examples

Example 12–13 Specifying the File Using a Variable

The following statement deletes the file whose name is stored in a text variable called `filevar`.

```
FILEDELETE filevar
```

FILEERROR

The FILEERROR function returns information about the first error that occurred when you are processing a record from an input file with the data reading statements FILEREAD and FILEVIEW. It can tell you what type of error occurred and where Oracle OLAP was in the record. The keyword you specify as an argument determines the kind of information that is returned.

You should call FILEERROR once to find out the type of error. Then, you can call FILEERROR again to get more details about what caused the error. The return values for the type of error are also FILEERROR keywords. When FILEERROR returns a value other than `NA`, then you would probably call FILEERROR a second time using the return value itself as an argument.

Return Value

Returns various values depending on the type of error that occurred as outlined in Table 12–4, "Types of Errors Returned by FILEERROR" on page 12-52.

Syntax

FILEERROR (TYPE|POSITION|WIDTH|VALUE|DIMENSION)

Arguments

TYPE
Returns a text expression that specifies the type of error that has occurred. The types of errors and their meanings are listed in Table 12–4, "Types of Errors Returned by FILEERROR".

Table 12–4 Types of Errors Returned by FILEERROR

Return Value	Meaning
DIMENSION	The data reading statements tried to set the status of a dimension (through an implicit or explicit MATCH attribute), but the specified position or value did not exist.
NA	No error occurred in the processing of the current record.

Table 12–4 Types of Errors Returned by FILEERROR

Return Value	Meaning
POSITION	The data reading program tried to read from an invalid location in the record. A POSITION error can occur when the field or column is before the beginning of the record or when the field extends past the end of the record. An error beyond the end of the record occurs only for binary or packed data; for symbolic (textual) data, the data reading statements pad short records with blanks.
VALUE	The value could not be converted to the requested data type. For packed data, this means the record had an invalid hexadecimal digit.
WIDTH	The data reading statements specified an invalid field width. Invalid widths depend on the format of the data, which can be symbolic, packed, or binary: ▪ For symbolic format, the width is invalid when it is less than 1 or when it is `NA`. Note that `NA` is acceptable for ID data. ▪ For packed format, the width is invalid when it is less than 1, greater than 8, or `NA`. For binary format, the width requirement depends on whether the data is integer or decimal (floating-point). Integer data must have a width of 1, 2, or 4. Decimal data must have a width of 4 or 8.

POSITION
Returns an INTEGER that is the column number (for RULED records) or field number (for STRUCTURED records) when the error occurred.

WIDTH
Returns an INTEGER that is the current field width. It will return `NA` when `NA` was specified as the width or the error was a POSITION error. A POSITION error stops processing before the width can be evaluated.

VALUE
When the error type is VALUE, it returns a text expression that is the value that could not be converted. When the data is packed, the invalid value is shown as hexadecimal escapes. When the error type is DIMENSION, it returns the value that did not match any existing dimension value. For other error types, it returns `NA`.

DIMENSION
When the error type was DIMENSION, it returns a text expression that is the name of the dimension that had no matching dimension values. For other error types, it returns `NA`.

Notes

Flow of Control

When an error occurs in FILEREAD or FILEVIEW, processing of the current record stops and Oracle OLAP displays an appropriate error message. Then, when your program has a trap label, control branches to the label where you might call FILEERROR to investigate the problem. When you branch back to a FILEREAD or FILENEXT function, processing continues with the next record. When there are more errors in the record, they will not be evaluated.

Error Messages

Set ECHOPROMPT to `YES` in your data reading program when you want error messages to be displayed in the current outfile. When the error occurred during FILEREAD or FILEVIEW, any evaluation by FILEERROR occurs after the error message.

Fileerror Abbreviation

The abbreviation for FILEERROR is FILEERR.

Examples

Example 12–14 Error-Handling with TRAP

This example shows a sample trap label (ERROR:) and the error-handling code that follows it. (For information on error trapping and trap labels, see the TRAP command.) The code checks whether the file has been opened. If so, it checks whether the error that caused the branch is a data reading error. When it is, the program calls FILEERROR in a SHOW command to display information about the error. The body of the program (not shown) contains code that opens the file and assigns a file unit number to the variable `fil.unit`. ERRTYPE is a local variable that is declared at the beginning of the program.

```
error:
IF fil.unit EQ NA
  THEN DO
    POPLEVEL 'save'
    RETURN
  DOEND
IF ERRORNAME NE 'attn'
  THEN DO
    ERRTYPE = FILEERROR(TYPE)
    IF ERRTYPE NE NA
      THEN SHOW JOINCHARS('Error in record ' RECNO(fil.unit) -
         ' in column ' FILEERROR(POSITION) ': ' -
         ERRTYPE ' ' FILEERROR(&ERRTYPE))
    TRAP ON ERROR
    GOTO NEXT
  DOEND
FILECLOSE fil.unit
POPLEVEL 'save'
RETURN
```

FILEGET

The FILEGET function returns text from a file that has been opened for reading. When FILEGET reaches the end of the file, it returns `NA`.

Return Value

TEXT

Syntax

FILEGET(*fileunit* [LENGTH *intexpression*])

Arguments

fileunit
A fileunit INTEGER assigned to a file opened for reading in a previous call to the FILEOPEN function.

LENGTH *intexpression*
An INTEGER expression specifying the number of bytes FILEGET should read from the file. When an end-of-line character is reached in the input file, FILEGET simply starts a new line in the result it is constructing. When LENGTH is omitted, FILEGET reads one line or record regardless of how many bytes it contains.

Notes

Binary Files

When you use the FILEGET function with a binary file, you will get an error.

TEXT, Not NTEXT

All text read with FILEGET is translated into the database character set. FILEGET cannot read data that cannot be represented in the database character set.

Examples

Example 12–15 Program for Reading a File

Suppose you have a program called `readfile` that takes a file name as its argument. It opens the file, reads the lines of the file, adds them to a multiline text variable named `wholetext`, then closes it. `readfile` uses local variables to store the fileunit number and each line of the file as it is read.

```
DEFINE wholetext VARIABLE TEXT
LD Multiline text variable
DEFINE readfile PROGRAM
LD Program to store data from a file in a multiline text variable
PROGRAM
VARIABLE fil.unit INTEGER  "Local Var To Store File Unit
VARIABLE fil.text TEXT     "Local Var To Store Single Lines
FIL.UNIT = FILEOPEN(ARG(1) READ)
FIL.TEXT = FILEGET(fil.unit)        "Read The First Line
WHILE fil.text NE NA                "Test For End-of-file
  DO
  wholetext = JOINLINES(wholetext, fil.text)
  fil.text = FILEGET(fil.unit)      "Read The Next Line
  DOEND
FILECLOSE fil.unit
END
```

FILEMOVE

The FILEMOVE command changes the name or location of a file that you specify. The new file name may be the same or different from the original name.

Syntax

FILEMOVE *old-file-name new-file-name*

Arguments

old-file-name
A text expression specifying the name of the file you want to move or rename. Unless the file is in the current directory, you must include the name of the directory object in the name of the file.

Note: Directory objects are defined in the database, and they control access to directories and file in those directories. You can use the CDA command to identify and specify a current directory object. Contact your Oracle DBA for access rights to a directory object where your database user name can read and write files.

new-file-name
A text expression specifying the new name or location for the file. Unless the file is in the current directory, you must include the name of the directory object in the name of the file.

Examples

Moving a File

The following statement moves the file `log.txt` from your session's current directory object to file `oldlog.txt` in a directory object called `backup`.

```
FILECOPY 'log.txt' 'backup/oldlog.txt'
```

13
FILENEXT to FULLDSC

This chapter contains the following OLAP DML statements:

FILENEXT

The FILENEXT function makes a record available for processing by the FILEVIEW command. It returns `YES` when it was able to read a record and `NO` when it reached the end of the file.

Return Value

BOOLEAN

Syntax

FILENEXT(*fileunit*)

Arguments

fileunit
A fileunit number assigned to a file that is opened for reading in a previous call to the FILEOPEN function or by the OUTFILE command.

Notes

Opening and Closing Files

Before you can get records from a file with FILENEXT, use the FILEOPEN function to open the file for reading (READ mode). When you are finished, close the file with the FILECLOSE command.

Processing Data

After reading a record with FILENEXT, use the FILEVIEW command to process the record. FILEVIEW processes input data and assigns the data to analytic workspace objects or local variables according to a description of each field. You can call FILEVIEW more than once for continued processing of the same record. To process another record, call FILENEXT again.

Automatic Looping

When all the records are being processed in essentially the same way, the FILEREAD command is easier to use because it loops over the records in a file automatically.

Writing Records

To write selected records to an output file, see the FILEPUT command.

Record Numbers

Use the RECNO function to get the current record number for any file that is opened for read-only access.

Reading Binary and Text Files

When you did not specify BINARY for the file when you opened it, FILENEXT reads data up to and including the next newline character. When you specified BINARY for the file when you opened it, you must use FILESET to set LSIZE to the appropriate record length before using the FILENEXT function. Then, FILENEXT reads data one record at a time.

Examples

Example 13–1 Program That Uses FILENEXT

Suppose you receive monthly sales data in a file with the following record layout.

Column	Width	Format	Data
1	1	Text	Division code
2	10	Text	District name
12	10	Text	Product name
30	4	Packed binary	Sales in dollars
34	4	Packed binary	Sales in units

You want to process records only for your division, whose code is A. The following program excerpt opens the file, reads the lines of the file, determines if the data is for division A and, if so, reads the sales data, then closes the file. The file name is given as an argument on the statement line after the program name.

```
VARIABLE fil.unit INTEGER
. . .
fil.unit = FILEOPEN(arg(1) READ)
LIMIT month TO &arg(2)

WHILE FILENEXT(fil.unit)
  DO
    FILEVIEW fil.unit WIDTH 1 rectype
    IF rectype EQ 'A'
      THEN FILEVIEW fil.unit COLUMN 2 WIDTH 10 district -
                             WIDTH 10 product -
                             COLUMN 30 WIDTH 4 BINARY sales -
                             WIDTH 4 BINARY UNITS
  DOEND
FILECLOSE fil.unit
```

FILEOPEN

The FILEOPEN function opens a file, assigns it a fileunit number (an arbitrary integer), and returns that number. You use the fileunit number, rather than a file name, in any further references to the file. When Oracle OLAP cannot open the file, an error occurs.

Return Value

INTEGER

Syntax

FILEOPEN(*file-name* {READ|WRITE|APPEND} [BINARY]) [NLS_CHARSET *charset-exp*]

Arguments

file-name
A text expression specifying the name of the file you want to open. Unless the file is in the current directory, you must include the name of the directory object in the name of the file.

Note: Directory objects are defined in the database, and they control access to directories and file in those directories. You can use the CDA command to identify and specify a current directory object. Contact your Oracle DBA for access rights to a directory object where your database user name can read and write files.

READ
Opens the file for reading. (Abbreviated R)

WRITE
Opens the file for writing. File access begins at the top of the file. Therefore, opening an existing file in WRITE mode erases its contents completely even before anything is written to the file. (Abbreviated W)

APPEND
Opens the file for writing. File access begins at the end of the file, and data is added to the existing contents.

BINARY
Opens a binary-format file (a file with packed or binary data). When you specify BINARY, Oracle OLAP considers every character in the file to be data. Rather than using newline characters to tell when records end, it assumes records of a fixed length, which you can set with FILESET(...LSIZE). The default record length is 80.

NLS_CHARSET *charset-exp*
Specifies the character set that Oracle OLAP will use when reading data from the file specified by *file-name*. When this argument is omitted, then Oracle OLAP handles the data in the file as having the database character set, which is recorded in the NLS_LANG option.

Notes

Multiple File Units

You can open as many files at the same time as your operating system allows.

Access Modes

The mode of access, READ, WRITE, or APPEND, must be appropriate to the file.

OUTFILE Command

The OUTFILE command also opens a file and assigns an arbitrary integer as the fileunit number.

Case Sensitive File Names

When specifying a file name, be sure to match upper, lower, and mixed case exactly with the name of the file you want to open -- unless your operating system makes no case distinctions.

Examples

Example 13–2 FILEOPEN with an Argument Passed into a Program

The following line from a program opens a file whose name was specified as a program argument and saves the fileunit number in the variable `fil.unit`.

```
fil.unit = FILEOPEN(ARG(1), READ)
```

Example 13–3   FILEOPEN with a Binary File

The following statements open a binary file and set the record length.

```
VARIABLE filenum INTEGER
filenum = FILEOPEN('mydata' READ BINARY)
FILESET filenum LSIZE 132
```

FILEPAGE

The FILEPAGE command forces a page break in your output when PAGING is on. FILEPAGE can send the page break conditionally, depending on how many lines are left on the current page

Syntax

FILEPAGE *fileunit* [*n*]

Arguments

fileunit
A fileunit number assigned to a file that is opened in WRITE or APPEND mode by a previous call to the FILEOPEN function or by the OUTFILE command.

n
A positive integer expression that indicates a page break should occur when there are fewer than *n* lines left on the page. When the number of lines left equals or exceeds *n,* or *n* equals zero, no page break occurs. When *n* is greater than PAGESIZE, a page break occurs when LINENUM is not zero. When *n* is negative or omitted, a page break always occurs.

Oracle OLAP calculates the number of available lines left on the page using the values of the options that specify the page size, the current line number, and the bottom margin. The number, which is stored in LINESLEFT, is calculated according to the following formula.

LINESLEFT = PAGESIZE - LINENUM - BMARGIN

Notes

PAGE Command

The PAGE command has the same effect as specifying the FILEPAGE command for the fileunit number OUTFILEUNIT, which is the number of the current outfile destination. The following two statements are equivalent.

```
FILEPAGE OUTFILEUNIT
PAGE
```

Examples

Example 13–4 Using the FILEPAGE Command

In the following program fragment, you might send a FILEPAGE command when you know the next group of products will not fit on the page. The program takes as arguments the name of the output file, and three `month` dimension values.

```
fil.unit = FILEOPEN(ARG(1) WRITE)
LIMIT month TO &ARG(2) &ARG(3) &ARG(4)
COMMAS = NO
DECIMALS = 0
FOR district
  DO
    FILEPAGE fil.unit STATLEN(product)
    FOR product
    DO
      FIL.TEXT = product
      FOR month
        JOINCHARS(fil.text  ' ' CONVERT(sales TEXT))
      FILEPUT fil.unit fil.text
    DOEND
    FILEPUT fil.unit ''
  DOEND
FILECLOSE fil.unit
```

FILEPUT

The FILEPUT command writes data that is specified in a text expression to a file that is opened in WRITE or APPEND mode.

Syntax

FILEPUT *fileunit* {*text-exp*|FROM *infileunit*} [EOL|NOEOL]

Arguments

fileunit
A fileunit number assigned to a file that is opened for writing (WRITE or APPEND mode) by a previous call to the FILEOPEN function or by the OUTFILE command.

text-exp
A text expression that contains data for output.

FROM *infileunit*
Transfers a record read from *infileunit* by the FILENEXT function directly to the file specified by *fileunit.*

EOL
Specifies that a newline character is appended to the output string and written to the file. (Default)

NOEOL
Specifies that no newline character is added to the text written to the file.

Notes

FROM Keyword

The keyword phrase FROM *infileunit* lets you write selected records to an output file while continuing to process data with the FILEVIEW command.

Binary Files

When you use the keyword phrase FROM *infileunit*, you cannot mix binary and non-binary files. When either file was opened with the BINARY keyword, the other must be binary too.

NTEXT Values

When you specify NTEXT data to be written to a file, FILEPUT translates the text to the character set of the file. When that character set cannot represent all of the NTEXT characters, then data is lost.

Examples

Example 13–5 Writing Data to a File Using FILEPUT

Following is an example of a program that writes a file of sales data for three months. The name of the file is the first argument. The following program excerpt opens the file, writes the lines of data to the file, then closes it. This program takes four arguments on the statement line after the program name: the file name of the input data, and three month names.

```
DEFINE salesdata PROGRAM
LD Write Sales Data To File. Args: File Name, 3 Month Names
PROGRAM
VARIABLE fil.unit INTEGER
VARIABLE fil.text TEXT
fil.unit = FILEOPEN(ARG(1) WRITE)
LIMIT month TO &ARG(2) &ARG(3) &ARG(4)
LIMIT product TO ALL
LIMIT district TO ALL
COMMAS = NO
DECIMALS = 0
FOR district
  DO
    FOR product
    DO
      fil.text = product
      FOR month
        fil.text = JOINCHARS(fil.text  ' ' -
          CONVERT(sales TEXT))
      FILEPUT fil.unit fil.text
    DOEND
    FILEPUT fil.unit ''
  DOEND

FILECLOSE fil.unit
END
```

Example 13–6 Preprocessing Data

The following example uses a data file with the 1996 sales figures for the products sold in each district. Only the records that begin with "A" are important right now, but you want to save the rest of the records in a separate file for later processing. The following program excerpt uses FILENEXT to retrieve each record and FILEVIEW to find out what kind of record it is. A second FILEVIEW command processes the record when it is type "A." When not, a FILEPUT command writes it to the output file.

```
DEFINE rectype VARIABLE ID
LD One Letter Code Identifying The Record Type
VARIABLE in.unit INTEGER
VARIABLE out.unit INTEGER
. . .
in.unit = FILEOPEN( GET(TEXT PROMPT 'Input Filename: ') READ)
out.unit = FILEOPEN( GET(TEXT PROMPT 'Output Filename: ') -
   WRITE)

WHILE FILENEXT(in.unit)
   DO
     FILEVIEW in.unit WIDTH 1 rectype
     IF rectype EQ 'A'
       THEN FILEVIEW COLUMN 2 WIDTH 8 district SPACE 2 -
         WIDTH 8 product ACROSS month year Yr96: saleS
       ELSE FILEPUT out.unit FROM in.unit
  DOEND
FILECLOSE in.unit
FILECLOSE out.unit
. . .
END
```

FILEQUERY

The FILEQUERY function returns information about a file. The attribute argument you specify in your FILEQUERY function call determines the type of information that is returned.

Return Value

The data type of the return value depends on the attribute you specify. See Table 13–1, " File Attributes Returned by FILEQUERY" on page 13-14 for more information.

Syntax

FILEQUERY(*file-id attrib-arg*)

Arguments

file-id
A fileunit number or a file name.

- A fileunit number is a number that Oracle OLAP assigned to a file you opened through a previous call to the FILEOPEN function or through the OUTFILE command. You can use the return value of the FILEOPEN function or the value of the OUTFILEUNIT option.
- A file name is a text expression specifying the name of the file you want to move or rename. Unless the file is in the current directory, you must include the name of the directory object in the name of the file.

Note: Directory objects are defined in the database, and they control access to directories and file in those directories. You can use the CDA command to identify and specify a current directory object. Contact your Oracle DBA for access rights to a directory object where your database user name can read and write files.

Some attributes require that you specify a fileunit number; others require the file name. In many cases, you can specify either. Table 13–1, " File Attributes Returned by FILEQUERY" on page 13-14 lists the valid keywords for attrib-arg and, for each keyword, provides a description and indicates whether you specify a file-unit-number of a file-name for the *file-id* argument.

attrib-arg

Specifies the type of information you want to retrieve about the file. The data type of FILEQUERY's return value depends on the attribute you specify. The attribute you specify must be appropriate for the file; otherwise, an error occurs. Table 13–1, " File Attributes Returned by FILEQUERY" on page 13-14 lists the valid keywords for *attrib-arg* and, for each keyword, provides a description and indicates whether you specify a file-unit-number of a file-name for the *file-id* argument.

Table 13–1 File Attributes Returned by FILEQUERY

Keyword	Return Values	Return Data Type	file-id Argument
APPEND	`TRUE` when the file is open for writing at the end (that is, `TRUE` for APPEND and WRITE); `FALSE` when it is not.	`BOOLEAN`	Fileunit number
BMARGIN	The number of blank lines that form the bottom margin.	`INTEGER`	Fileunit number
CHANGED	`TRUE` when the file's archive bit is set; `FALSE` when it is not.	`BOOLEAN`	Fileunit number or file name
DATE	The date that the file was last modified	`DATE`	Fileunit number or file name
EOF	`TRUE` when end-of-file has been reached; `FALSE` when it is not.	`BOOLEAN`	Fileunit number
EXISTS	`TRUE` when the file exists; `FALSE` when it is not.	`BOOLEAN`	Fileunit number or file name
FILENAME	The file name associated with the fileunit.	`TEXT`	Fileunit number
LINENUM	The current line number. Resets after each pagebreak when PAGING is on; keeps incrementing when PAGING is off. When file is currently open in READ mode, returns the current record number.	`INTEGER`	Fileunit number
LINESLEFT	The number of lines left on the page.	`INTEGER`	Fileunit number
LSIZE	For a file that is open for writing, the line length for the standard Oracle OLAP page heading. (See the STDHDR program.) For a fileunit that is open for reading, specifies the record length for binary input files.	`INTEGER`	Fileunit number
NAMELIST	The name of the file (such as `demo`), without an extension.	`TEXT`	Fileunit number or file name
NLS_CHARSET	The character set being used for this fileunit. See the FILEOPEN command for more information.	`TEXT`	Fileunit number
NUMBYTES	The size of the file in bytes.	`INTEGER`	Fileunit number or file name

Table 13–1 (Cont.) File Attributes Returned by FILEQUERY

Keyword	Return Values	Return Data Type	file-id Argument
ORIGIN	The type of computer on which the file was created.	TEXT	Fileunit number
PAGENUM	The current page number. See "Paging Attributes" on page 13-16.	INTEGER	Fileunit number
PAGEPRG	The Oracle OLAP program or statement that produces headings when output is paged. See "Paging Attributes" on page 13-16.	TEXT	Fileunit number
PAGESIZE	The number of lines on each page. See "Paging Attributes" on page 13-16.	INTEGER	Fileunit number
PAGING	TRUE when the output is formatted in pages; FALSE when it is not. See "Paging Attributes" on page 13-16.	BOOLEAN	Fileunit number
PAUSEATPAGEEND	TRUE when Oracle OLAP will pause after each page; FALSE when it will not. See "Paging Attributes" on page 13-16.	BOOLEAN	Fileunit number
R[EAD]	TRUE when the file is open for reading; FALSE when it is not.	BOOLEAN	Fileunit number
RO	TRUE when the file's read-only attribute is set; FALSE when it is not.	BOOLEAN	Fileunit number or file name
SPECLIST	The name and extension of the file.	TEXT	Fileunit number or file name
TABEXPAND	TRUE when the tab characters will be expanded when the file is read by FILEGET or FILEREAD; FALSE when they will not. See "Tab Treatment" on page 13-16.	BOOLEAN	Fileunit number or file name
TIME	The time that the file was last modified.	DATETIME	Fileunit number or file name
TMARGIN	The number of blank lines that form the top margin.	INTEGER	Fileunit number
UNIT	The file unit for the specified file name.	INTEGER	File name
W[RITE]	TRUE when the file is open for writing; FALSE when it is not.	BOOLEAN	Fileunit number

Notes

Related OLAP DML Statements

Before specifying a fileunit with the FILEQUERY function, use FILEOPEN or OUTFILE to open the file.

ORIGIN Attribute

The ORIGIN attribute identifies the computer on which a file was originally created. You must use the FILESET command to set ORIGIN when the file originated on another type of computer. This attribute is relevant only for files that are open for reading.

Listing Open Files

You can use the LISTFILES command to see a list of which open files can be referenced by the FILEQUERY function.

Tab Treatment

When you want tab characters in the source file to be expanded when read by FILEGET or FILEREAD, you can specify the TABEXPAND attribute with the FILESET command. When TABEXPAND is zero, tab characters will not be expanded. A value greater than 0 indicates the distance, in bytes, between tab stops. The default value of TABEXPAND is 8.

Paging Attributes

The paging attributes apply only to files that currently, unless otherwise noted, have PAGING set to `YES` and are open in WRITE mode -- such as files opened with FILEOPEN(...WRITE) or FILEOPEN(...APPEND). You can set any of the paging attributes with the FILESET command.

Using FILEQUERY EOF

Use FILEQUERY with the EOF attribute where you used FILESTATUS in previous Oracle OLAP versions.

File Name Conventions

When specifying a file name, use the file naming conventions for your operating system. For example, it might be necessary to type upper, lower, and mixed case letters to match the name of the file for which you want information.

Wildcard Characters

(Unix only) When querying for Unix file names, wildcard characters (that is, `* ?`) are allowed when searching with the NAMELIST, SPECLIST, and EXISTS attribute arguments.

Examples

Example 13–7 Setting Paging Options for a File Opened for Writing

The following statements show how the paging options are set for a file opened for writing.

```
DEFINE fil.unit INTEGER
fil.unit = FILEOPEN('REPORT' WRITE)
```

- The statement

  ```
  SHOW FILEQUERY(fil.unit PAGING)
  ```

  produces the following output.

  ```
  YES
  ```

- The statement

  ```
  SHOW FILEQUERY(fil.unit PAGESIZE)
  ```

  produces the following output.

  ```
  66
  ```

- The statement

  ```
  SHOW FILEQUERY(fil.unit TMARGIN)
  ```

  produces the following output.

  ```
  5
  ```

- The statement

  ```
  SHOW FILEQUERY(fil.unit LINESLEFT)
  ```

  produces the following output.

  ```
  61
  ```

The following statement closes the file.

```
FILECLOSE fil.unit
```

FILEREAD

The FILEREAD command reads records from an input file and processes data according to action statements that you specify. FILEREAD handles binary and packed decimal data, as well as text. It can handle decimal data written in E-notation (such as `.1E+9`) or M-notation (such as `10M`). It can convert the data to any appropriate data type before storing it in an Oracle OLAP variable, dimension, composite, or relation.

Syntax

FILEREAD *fileunit* [STOPAFTER *n*] [*file-format*] {[*attribute...*] *action-statement1*}

[[*attribute...*] *action-statementN...*]

where:

file-format specifies the format of the records in the input file as follows:

RULED|CSV [DELIMITER *dchar*]|STRUCTURED [TEXTSTART *schar*] -

[TEXTEND *echar*] [DELIMITER *dchar*]

attribute provide information that is used by action statements. For example, attributes can be used to locate a field in the input record, format the data from that field, convert the data to a different data type, or specify how the data should be processed as a dimension value in Oracle OLAP. For information on the placement of attributes in action statements, see "Field Attributes" on page 13-34. An *attribute* can be one or more of the following:

COLUMN *n* | COL *n*

SPACE *n* | SP *n*

FIELD *n* | FLD *n*

WIDTH *n* | W *n*

INTEGER | SHORTINTEGER | DECIMAL | SHORTDECIMAL | NUMBER | TEXT | ID | DATE | VNF | RAW DATE | BOOLEAN

MATCH | APPEND [LAST | FIRST | BEFORE *pos* | AFTER *pos*] | ASSIGN

BINARY | PACKED | SYMBOLIC

TRANSLATE | NOTRANSLATE |

SCALE *n*

ZPUNCH | ZPUNCHL

LSET '*text*'

RSET '*text*'

NOSTRIP | STRIP | LSTRIP | RSTRIP

NAVALUE *val*

NASPELL '*text*'

ZSPELL '*text*'

YESSPELL '*text*'

NOSPELL '*text*'

ZEROFILL

action-statements perform processing, such as assignment statements and IF statements. For example, an action statement can compare dimension values with values retrieved from the input record, assign data to one or more cells in a dimensioned variable, or simply increment a counter. An *action-statement* can be one of the following:

assignment-statement

IF-*statement*

SELECT-*statement*

ACROSS-*statement*: *action-statement*

<*action-statement-group*>

Arguments

fileunit
A fileunit number assigned to a file that is opened for reading (READ mode) by a previous call to the FILEOPEN function.

STOPAFTER *n*
The number of records to read from the input file. When STOPAFTER is left out, or specified with a negative number or an `NA`, FILEREAD processes the whole file. See "STOPAFTER Keyword" on page 13-30.

RULED
Specifies that the record is organized in fixed-width columns, that is, character-by-character or byte-by-byte. All lines must have exactly the same format. RULED is the default file format. Use the COLUMN, SPACE, and WIDTH attributes to specify the location of the data in the records.

CSV [DELIMITER *dchar*]
CSV specifies that the data is in CSV (comma-delimited values) format. You must use the FIELD and SPACE attributes to specify the location of the data in the record.

dchar is a text expression that specifies a single character that you want Oracle OLAP to interpret as the general field delimiter in a structured file. Oracle OLAP uses the general field delimiter to identify both numeric and text fields. The default character is a comma (`,`).

CSV files are a common output format that is generated by spreadsheet programs. Each line of characters in a source file is treated as a single record. Each field in the record is separated by a comma by default. You can use the DELIMITER keyword to specify some other character as field delimiter.

When a group of characters in the input record is enclosed by double quotation marks, all of the following rules apply:

- When the group includes the delimiter character, it is treated as a literal instead of as a delimiter.
- When a double quotation mark (`"`) is included in the group of characters, then it must be followed by another double quotation mark.
- When a linefeed character (`\n`) is included in the group of characters, then it is ignored.
- Any spaces or tabs that occur before or after the double quotation marks that enclose the group of characters will be ignored.

STRUCTURED
Specifies that the record is in "structured prn" format. You must use the FIELD and SPACE attributes to specify the location of the data in the record.

Structured files are a common output format for PC software. They are text files in which the fields are composed of groups of characters. A group of characters is defined by two conditions: text enclosed in double quotes, or a sequence of numbers that is uninterrupted except by a decimal point. This means that an unquoted sequence of numbers containing a decimal point will be stored as a single value; however, an unquoted sequence of numbers containing commas or other delimiters to mark off thousands will be split into several values rather than stored as a single value. Any unquoted, non-numeric characters are ignored, except a minus sign that immediately precedes a number is considered to be part of the number. A space cannot separate the minus sign from the number.

When your file format does not conform to the pattern described here, you can use the TEXTSTART, TEXTEND, and DELIMITER keywords that let you customize the delimiters FILEREAD uses to identify the start and end of each field.

TEXTSTART *schar*
Specifies a single character that you want Oracle OLAP to interpret as the start of a text field in a structured file. *schar* is the value of the character. The default character is a double quote (`"`).

TEXTEND *echar*
Specifies a single character that you want Oracle OLAP to interpret as the end of a text field in a structured file. *echarr* is the value of the character. The default character is a double quote (`"`).

DELIMITER *dchar*
Specifies a single character that you want Oracle OLAP to interpret as the general field delimiter in a structured file. Oracle OLAP uses the general field delimiter to identify both numeric and text fields. *dchar* is the value of the character. The default character is a comma (`,`).

COLUMN *n*
COL *n*
The column in which the field starts in the input record. By default, field 1 begins in column 1 and subsequent fields begin in the column following the previous field. The current field's default column is the sum of the previous field's first column plus its width plus any spaces specified for the current field.

SPACE *n*
SP *n*
The number of spaces between a field and the preceding field. In a structured PRN file, the number of fields between the preceding and current field. The default is `0`.

FIELD *n*
FLD *n*
In a structured PRN file only, the field from which to extract the data.

WIDTH *n*
W *n*
The number of columns the field occupies in the input record. The default is derived from the data type according to the following list:

- BINARY input format with INTEGER, SHORTINTEGER, or SHORTDECIMAL target data type has a default of 4 columns.
- BINARY input format with DECIMAL or NUMBER target data type has a default of 8 columns.
- BINARY input format with BOOLEAN target data type has a default of 2 columns.
- PACKED input format with any type of target data type has no default.
- SYMBOLIC input format with ID target data type has a default of 8 columns.
- SYMBOLIC input format with a target data type that is not ID has no default.

When there is no default, WIDTH must be included for ruled records or FILEREAD generates an error. (Structured records do not require a WIDTH specification.)

The maximum width is 4000 characters for text input.

INTEGER
SHORTINTEGER
DECIMAL
SHORTDECIMAL
NUMBER
TEXT
ID
DATE
VNF
RAW DATE
BOOLEAN
For text data, the data type to which the input is converted before it is stored in your analytic workspace.

- Except for dimensions of type DAY, WEEK, MONTH, QUARTER, and YEAR, the default is the data type of the target object.
- For dimensions of type DAY, WEEK, MONTH, QUARTER, and YEAR, the default is VNF.
- For DATE variables and dimensions of type DAY, WEEK, MONTH, QUARTER, and YEAR, RAW DATE indicates the input values are positive integers that

represent the number of days since December 31, 1899, or negative integers that represent the number of days before December 31, 1899.

- For binary data, the data type of the data in the input record.

See "NTEXT Values" on page 13-34.

MATCH
APPEND
ASSIGN
When the target object is a dimension, these attributes specify whether or not to add new dimension values. For each record processed, the dimension is temporarily limited to the value in the record.

When the target object is a dimension and you do not specify a dimension attribute, then values in the input field must match current dimension values. When the value does not exist, FILEREAD generates an error. This attribute also applies when the target object is a a dimension surrogate.

The field contains new dimension values and may contain existing values as well. New values are added to the dimension list and the status is limited to the current value. The status is set to ALL after FILEREAD finishes. For time dimensions, Oracle OLAP automatically fills in any "missing" periods between the existing ones and the new ones. This attribute does not apply when the target object is a a dimension surrogate because you cannot directly add values to a surrogate.

This attribute applies only to a dimension surrogate. It assigns the new value to the surrogate.

LAST
When the target object is a non-time dimension and the dimension attribute is APPEND, adds the value to the end of the dimension list.

FIRST
When the target object is a non-time dimension and the dimension attribute is APPEND, adds the value to the beginning of the list.

BEFORE *pos*
When the target object is a non-time dimension and the dimension attribute is APPEND, adds the value before the specified value or integer position.

AFTER *pos*
When the target object is a non-time dimension and the dimension attribute is APPEND, adds the value after the specified value or integer position.

SYMBOLIC
Specifies that the format of the input field is ASCII or EBCDIC text.

BINARY
Specifies that the format of the input field is binary.

PACKED
Specifies that the format of the input field is packed decimal.

TRANSLATE
Specifies that Oracle OLAP translates the data from the format of the original operating system, as identified by a FILESET ORIGIN statement

NOTRANSLATE
Specifies that Oracle OLAP does not translate the data from the format of the original operating system, as identified by a FILESET ORIGIN statement.

SCALE *n*
The number of digits to the right of the assumed decimal or binary point. The default is 0. When the input data is text, a decimal point in the input overrides the number specified by SCALE.

ZPUNCH
Specifies that the input is zone overpunched.

ZPUNCHL
Specifies that the input is zone overpunched on the left.

LSET '*text*'
For text input and TEXT or ID target objects, adds text to the left of the value before storing. When text is multiline, only the first line is used. By default, no text is appended

RSET '*text*'
For text input and TEXT or ID target objects, adds text to the right of the value before storing. When text is multiline, only the first line is used. By default, no text is appended.

NOSTRIP
For text input specifies that no spaces (or nulls) are stripped from the input.

STRIP
For text input specifies that spaces (and nulls) are stripped from both left and right of the input.

LSTRIP
For text input specifies that spaces and nulls are stripped from the left of the input.

RSTRIP
For text input specifies that spaces and nulls are stripped from the right of the input.

NAVALUE *val*
For binary or packed input, specifies that when the input is the specified numeric value, `NA` is assigned to the target object.

NASPELL '*text*'
For text input, specifies that Oracle OLAP stores text as NA. When the input is the specified text, NA is assigned to the target object. Text can be a multiline string listing several possible NA values. In addition to the values specified for text, when the input is NA, then NA is assigned to the target object.

ZSPELL '*text*'
For textual numeric input, specifies that Oracle OLAP stores text as 0. When the input is the specified text, zero is assigned to the target object. Text can be a multiline string that lists several possible zero values. In addition to the values specified for text, when the input is `0`, then `0` is assigned to the target object.

YESSPELL '*text*'
For text input that is BOOLEAN, specifies that Oracle OLAP stores text as YES. When the input is text then YES is assigned to the target object. Text can be a multiline string that lists several possible YES values. In addition to the values specified in *text,* when the input is `YES`, ON, or TRUE, `YES` is assigned to the target object.

NOSPELL '*text*'
For text input that is BOOLEAN, specifies that Oracle OLAP stores text as NO. When the input is text then NO is assigned to the target object. Text can be a multiline string that lists several possible NO values. In addition to the values specified in 'text,' when the input is NO, OFF, or FALSE, NO is assigned to the target object.

ZEROFILL
For text numeric input, specifies that Oracle OLAP fills any spaces in the resulting text with zeros. Any spaces in the input are replaced with zeros. The default is no filling with zeros.

action-statement
You may specify one or more action statements to be performed each time a record is retrieved from the input file. Typically, you will use action statements to set dimension status and assign data retrieved from the input record to a target object in Oracle OLAP. However, you may specify action statements that do not reference the data in the input record. For example, one of your action statements might be an assignment statement that simply increments a counter. Alternatively, an action statement might use the input data in some kind of processing, but not actually assign it to a target object in Oracle OLAP.

In your list of action statements, *be sure to process dimensions before variables*. FILEREAD processes each action statement from left to right for each input record. When an action statement performs dimension processing, the resulting status remains in effect for subsequent action statements. When you do not first specify action statements that limit a variable's dimensions, FILEREAD uses the first value in status to target a cell in the variable. Unless you specify an ACROSS phrase, FILEREAD assigns a single value from a field in an input record to a single cell in an Oracle OLAP variable. By default, FILEREAD does *not* loop over a variable's dimensions when assigning data to the variable. See "Field Order" on page 13-29.

Use the VALUE keyword in FILEREAD action statements to represent the value in a particular field of the input record. VALUE represents this data, formatted according to the FILEREAD attributes you have specified. When the field in the record is blank, FILEREAD considers its value to be `NA`. By default, the data type of VALUE is the data type of the target object. However, you can specify a different data type with an attribute keyword.

Note: When you have already specified action statements for use with FILEREAD, you can reuse the code with SQL FETCH and SQL IMPORT by simply adjusting the assignment statements and eliminating the VALUE keyword (if necessary). Most of the FILEREAD attributes (with the exception of the attributes that control dimension processing) are not meaningful for SQL loading and are ignored when executing within SQL FETCH and SQL IMPORT.

assignment-statement

An assignment statement lets you assign a value to an Oracle OLAP object. An assignment statement has the following form.

object [= *expression*]

object is the target where the data will be assigned and stored. The *object* can be an Oracle OLAP variable, dimension, dimension surrogate, composite, or relation.

expression is the source of the data value to be assigned to the target.

Important: In a SQL FETCH or a SQL IMPORT assignment statement, the *expression* component is *not* optional. However, a FILEREAD assignment statement may consist only of an object name. In this case, the input data is assigned directly to *object*. An *expression* in a FILEREAD assignment statement may include the VALUE keyword.

IF*-statement*

An IF statement lets you perform some action depending on whether a Boolean expression is `TRUE` or `FALSE`. An IF statement has the following form.

IF *bool-exp*

THEN *action*

[ELSE *action*]

IF evaluates the Boolean expression. When it is TRUE, the THEN *action* occurs. When it is FALSE, the ELSE *action* (if specified) occurs. When the Boolean expression is `NA`, no *action* occurs.

An *action* can be one of the following:

- NULL (no action occurs)
- An assignment statement
- A SELECT statement
- An IF statement
- A DO … DOEND statement containing *action-statements*

A FILEREAD IF statement may contain invocations of the VALUE keyword. You can use a FILEREAD IF statement to process varying record types (such as records with different structures or different target objects) with one FILEREAD command.

In FILEREAD, the VALUE keyword can be used more than once to represent different values from the same record. For each instance, specify the column from which to read each value.

SELECT *statement*
A SELECT statement lets you perform some action based on the value of an expression. A SELECT statement has the following form.

```
SELECT select-expression
  [WHEN expression1 action
  [WHEN expression2 action . . .]
[ELSE action]
```

SELECT evaluates the SELECT expression and then sequentially compares the result with the WHEN expressions. When the first match is found, the associated *action* occurs. When no match is found, the ELSE *action* (if specified) occurs.

An *action* for a SELECT statement is the same as an *action* for an IF statement.

A FILEREAD SELECT statement may contain invocations of the VALUE keyword. You can use a FILEREAD SELECT statement to process varying record types (such as records with different structures or different target objects) with one FILEREAD command.

ACROSS-statement: *action-statement*
An ACROSS statement causes the following action statement to execute once for every value in status of the ACROSS dimension. When you want the looping to apply to more than one action statement, enclose the action statements in angle brackets. An ACROSS statement has the following form.

```
ACROSS dimension [limit-clause]:
  action-statement
```

limit-clause temporarily changes the status of *dimension*, as long as you are not in a FOR loop over *dimension*. The new status is in effect only for the duration of the SQL FETCH command. The format of *limit-clause* is as follows.

```
[ADD|COMPLEMENT|KEEP|REMOVE|TO] valuelist
```

To specify the temporary status, insert any of the LIMIT command keywords (the default is TO) along with an appropriate list of dimension values or related dimensions. You can use any valid LIMIT clause (see LIMIT command for further information). The following example limits `month` to the last six values, no matter what the current status of `month` is.

```
ACROSS month last 6: units
```

In a FILEREAD ACROSS statement, you can specify attributes to indicate the position in the record where Oracle OLAP will begin reading the fields specified by the ACROSS phrase. To specify the position, use the attributes FIELD, SPACE, and COLUMN. A position attribute is optional when the series of fields specified in the ACROSS phrase begins in the next field for structured records, or the next byte for ruled records.

<action-statement-group>
You can group several action statements together by enclosing them in angle brackets. An *action-statement-group* has the following form.

<*action-statement1* -

[*action-statement2* . . .]>

A typical use for action statement groups is after an ACROSS statement. With the angle bracket syntax, you can cause more than one action statement to execute for every value in status of the ACROSS dimension.

Notes

Reading One Record at a Time
As an alternative to FILEREAD, you can use the FILENEXT function to read one record at a time with one or more FILEVIEW commands to process the fields in the record.

Related OLAP DML Statements
Before you can process data from a file with FILEREAD, use the FILEOPEN function to open the file for reading (READ mode). When you are finished, close the file with the FILECLOSE command.

Field Order
When an input record contains both dimension values and variable data, the dimension values must be the first fields that are read in the record, and the variable data values must be read after those dimension values. To do this, you can either order the fields in the input record itself or you can use FILEREAD attributes to specify the field positions explicitly. (See the description for the *attribute* argument.)

To organize the input records so that you do not need to use position attributes with FILEREAD, put all of the dimension values in the first fields of the record and put the variable data values in the last fields of the record. For example, suppose that you have data for two variables (`units` and `sales`) that share the same dimensions in the same order (`time`, `product`, and `geography`). In this case, the first three fields in the input record should contain dimension values, while the fourth and fifth fields should contain variable data, such as in the following sample input record.

```
Sep99    Snowshoes    Boston    35    5565.95
```

STOPAFTER Keyword

By default, FILEREAD automatically reads all the records in a file in sequential order. When you want to process only the first part of a file, use the STOPAFTER keyword. FILEREAD processes the number of records you specify, then stops. You can then close the file.

When you want to skip the first part of the file and process the remaining records, you can use the STOPAFTER keyword and omit the field descriptions. FILEREAD will read the number of records you specify without processing the data. Then you issue a second FILEREAD command with field descriptions for processing the input. The following program lines illustrate this method.

```
lIMIT district TO 'Boston'
unit = FILEOPEN('bostdata' READ)
FILEREAD unit STOPAFTER 25
FILEREAD unit WIDTH 8 product SPACE 2 ACROSS month 13 TO 24:-
   WIDTH 4 PACKED sales
```

Dimension Maintenance

When the target object of a field description is a dimension, you can specify whether or not to use the data in the file to add values to the dimension. The dimension attributes are MATCH and APPEND. When you are adding values to a dimension with APPEND, you can specify a dimension position attribute (`LAST`, `FIRST`, `BEFORE` *`pos`*, `AFTER` *`pos`*) immediately after APPEND.

In an assignment statement of the form `object=expression`, dimension attributes cannot appear on the right side of the equal sign, but must be specified before the target object. The only exception is when dimensions as target objects also appear on the right side, such as when you are maintaining a conjoint dimension. See Example 13–12, "Maintaining Conjoint Dimensions with File Data" on page 13-38.

Dimension Position Numbers

When your input data consists of dimension position numbers, rather than dimension values, specify the conversion type as INTEGER in the field description, even though the dimension has a type of TEXT, ID, DAY, WEEK, MONTH, QUARTER, or YEAR.

```
FILEREAD unit COLUMN 1 WIDTH 8 INTEGER month
```

When the input contains position numbers, you cannot use the APPEND keyword to add new values to a dimension of type TEXT, ID, DAY, WEEK, MONTH, QUARTER, or YEAR, because the new position numbers have no associated value to be added.

Conjoint Dimension Maintenance

When a conjoint dimension is the target object, you can read its values using one of the two methods:

- **Method One**—When the input contains values or position numbers of the base dimensions, you must specify a dimension list surrounded by angle brackets after the equal sign, as shown in the following two sample lines.

  ```
  FILEREAD unit proddist = <COL 1 W 10 product COL 20 -
     W 8 district>
  FILEREAD unit proddist = <COL 1 W 10 INTEGER product COL 20 -
     W 8 INTEGER district>
  ```

  The preceding examples show values of the `product` and `district` dimensions being used to designate a value of the `proddist` concat dimension You could also use the APPEND attribute when you needed to maintain any of the dimensions. However, when you needed to process the values of `product` or `district` first, so that the syntax would require an equal sign inside the angle brackets, you would have to use an alternative method. (Nested equal signs are not allowed.) For this method you would read and process the base dimension values first, and then use the dimensions, without any field attributes, in the dimension list for the conjoint dimension. For example, to convert the base dimension values of a conjoint dimension to uppercase, use a statement similar to the following.

```
FILEREAD unit COL 14 W 8 product = UPCASE(VALUE) -
   COL 5 W 8 district = UPCASE(VALUE) -
   proddist = <product, district>
```

- **Method Two**—When the input contains position numbers of the conjoint dimension itself, you must specify the INTEGER keyword.

```
FILEREAD unit INTEGER proddist
```

FILEREAD with Variables Dimensioned by Composites

When reading data into a variable dimensioned by a composite, FILEREAD automatically creates any missing target cells that are being assigned non-`NA` values. This process also adds to the composite all the dimension value combinations that correspond to those new cells. Thus, both the target object and the composite might be larger after an assignment.

Variables Dimensioned by Composites and Efficiency

When you use the automatic composite maintenance feature of FILEREAD to load data into variables dimensioned by composites, you should be aware of potential performance problems that might later occur when you attempt to access the variables' data. The position of a composite in the dimension list of a variable indicates whether or not performance might later become an issue.

When the composite appears at the end of the dimension list in the variable's definition (the slowest-varying position), you can use FILEREAD just as you would for a variable whose dimension list does not include composites. For example, you could use the same FILEREAD commands to read data into the variables `newsales` and `newsales.cp` (with the following definitions) without sacrificing efficiency.

```
DEFINE newsales VARIABLE DECIMAL <product district month>
DEFINE newsales.cp VARIABLE DECIMAL <product SPARSE<district month>>
```

`newsales.cp` is dimensioned by three dimensions, the last two of which are in a composite. When, however, you have a variable like `newsales2.cp` (with the following definition) there can be performance implications for accessing data loaded with FILEREAD.

```
DEFINE newsales.cp VARIABLE DECIMAL <SPARSE<district month> product >
```

In this case, you can use one of two methods to avoid performance problems. Refer to "Prevent Performance Problems: Method One" on page 13-33 and "Prevent Performance Problems: Method Two" on page 13-33.

Prevent Performance Problems: Method One

You can use CHGDFN with the SEGWIDTH keyword to change the segment size for the variable before using FILEREAD. CHGDFN SEGWIDTH lets you specify the size of a variable's segments. A segment is a portion of the total number of values a variable holds. The number of segments in a variable affects the performance of data loading and data accessing.

The segment size that you specify with a CHGDFN SEGWIDTH statement is used not only for the variable you designate as *varname*, but also for all other variables and relations that are defined with the same combination of dimensions and composites in the same order.

Prevent Performance Problems: Method Two

Alternatively, you can explicitly add composite values just as you would for a conjoint dimension. You can use this method both for named and unnamed composites. See "Composite Maintenance" on page 13-33.

Composite Maintenance

When you wish to explicitly maintain composites with FILEREAD, use the same syntax that you use to maintain conjoint dimensions. When the composite is unnamed, refer to it with the form `SPARSE<dim1 dim2 ...>`. See "FILEREAD with Variables Dimensioned by Composites" on page 13-32 and "Variables Dimensioned by Composites and Efficiency" on page 13-32 to evaluate the advantages of explicit versus automatic composite maintenance with FILEREAD.

Time Dimensions

When the target object of a field is a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR, the default conversion type is VNF. Therefore, you do not need to specify a conversion type when the input values are formatted according to the VNF of the target dimension (or the default VNF when the dimension does not have a VNF of its own).

When the target object of a field is a DATE variable or a dimension of type DAY, WEEK, MONTH, QUARTER, and YEAR, FILEREAD will interpret the values correctly when they are in a valid input style for dates as described in DATEORDER. For dimensions of type DAY, WEEK, MONTH, QUARTER, and

YEAR, you must specify DATE as the conversion type. For values of a DATE variable, DATE is the default conversion type, so the DATE keyword is optional.

FILEREAD will also interpret values of a time dimension or a DATE variable correctly when they are integers that represent dates (`1 = January 1, 1900`). In this case, you must specify RAW DATE as the conversion type.

Blank Fields

When a field is blank, its value is `NA` and `NA` is assigned to the target variable. Examples of blank fields are a text field filled with spaces, a field that begins beyond the end of the record, or a field in a structured file that has nothing, not even a space, between the field delimiters.

Field Attributes

Normally, the field attributes immediately precede the target object or the expression on the right of the equal sign.

attributes object

However, when you want an attribute to apply to several fields, specify the attribute followed by the list of target objects surrounded by angle brackets. You can also include attributes that apply to one of the objects by typing them inside the brackets before the object to which they apply.

attributes0 <attributes1 object1=expression object2 attributes3 object3>

Angle brackets are also used to surround the base values of a conjoint dimension value.

Error Handling

When FILEREAD encounters an error, you can control what happens with an error trap and appropriate processing. Errors can be caused by attempts to convert data to an incompatible data type or by encountering invalid dimension values. You can use the FILEERROR function to get more information about what caused the error. After processing the error, you can use the TRAP command to turn error trapping back on and GOTO to branch back to the FILEREAD command. Processing continues with the next record. See Example 13–10, "Error Handling" on page 13-37.

NTEXT Values

When you specify a target object of type NTEXT for data from a structured or CSV file, FILEREAD translates the data from the file into the database character set before storing the values (even though they are assigned to an NTEXT object). This can result in data loss when the data from the file cannot be represented in the database character set. For data from a ruled file, which has fixed-width columns, FILEREAD does not translate into the database characters set, so there is no data loss.

Examples

Example 13–8 Dimension Values and Data

Suppose your analytic workspace contains six-character product identification numbers. You need to import both product names and a value for the number of units sold each month. The data file for the last quarter has the following format.

```
Jan951234aa00Chocolate Chip Cookies        123
Jan951099bb00Oatmeal Cookies               145
Jan952355cc00Sugar Cookies                 223
Jan955553ee00Ginger Snap Cookies           233
Feb951234aa00Chocolate Chip Cookies        123
Feb951099bb00Oatmeal Cookies               O145
Feb952355cc00Sugar Cookies                 SS223
Feb955553ee00Ginger Snap Cookies           G233
Mar952355cc00Sugar oCookies                 223
Mar955553ee00Ginger Snap Cookies           233
Mar953222dd00Brownies                      432
```

The dimension and variables have the following definitions.

```
DEFINE month DIMENSION MONTH
DEFINE productid DIMENSION ID
DEFINE productname VARIABLE TEXT <productid>
DEFINE units.sold VARIABLE INTEGER <month productid>
```

The following program uses FILEREAD to add any new values for `month` and `productid` to the analytic workspace and to put the data in the correct variables.You should maintain dimensions in one FILEREAD command, close the file, and process it again to get the associated data.

```
DEFINE read.product PROGRAM
PROGRAM
VARIABLE fi INT
fi = FILEOPEN('Dr.Dat' READ)
FILEREAD fi COLUMN 1 APPEND WIDTH 5 month -
   COLUMN 6 APPEND WIDTH 6 productid
FILECLOSE fi

fi = FILEOPEN('Dr.Dat' READ)
FILEREAD fi COLUMN 1 WIDTH 5 month -
   COLUMN 6 WIDTH 6 productid -
   COLUMN 12 WIDTH 30 productname -
   COLUMN 44 WIDTH 22 units.sold
FILECLOSE fi
END
```

Example 13–9 Dimension Surrogate Values

This example uses one FILEREAD operation to add a value to the `product` dimension and assign a value to `prodnum`, which is a NUMBER dimension surrogate for the `product` dimension. It uses a second FILEREAD to assign a value to the `units` variable, which is dimensioned by `month`, `product`, and `district`. The data file for the dimension and surrogate values has the following format.

```
Kiyaks400
```

The following statements define a fileunit, open the file, read its contents and append a value to the `product` dimension and assign a value to the `prodnum` surrogate, and close the file.

```
DEFINE funit INT
funit = FILEOPEN('Ds.Dat' READ)
FILEREAD funit COL 1 APPEND W 6 product COL 7 ASSIGN W 3 prodnum
FILECLOSE funit
```

The data file for the variable value has the following format.

```
Jan02400Boston416
```

The following statements open the file, read its contents, match the value of the `prodnum` surrogate and assign a value to the `units` variable, and close the file.

```
funit = FILEOPEN('Var.Dat' READ)
FILEREAD funit COL 1 W 5 month COL 6 MATCH W 3 prodnum -
  COL 9 W 6 district COL 15 W 3 INTEGER units
FILECLOSE funit
```

Example 13–10   Error Handling

When your input file has data that does not match the format specifications, or when it has a dimension value that is not part of the analytic workspace when you are using the default MATCH attribute, you will get an error. You can use error processing at the trap label to check for that kind of error, skip the bad record, and continue processing the file. You can also use the FILEPUT command to store the bad records in a separate file (see FILEPUT).

In the following example, the statements at the trap label check whether the file was successfully opened (`fil.unit` has an integer value) and whether the user interrupted the program. When these are not the reason for the error, the program assumes it encountered a bad record, resets the trap, and branches back to the FILEREAD command to continue processing with the next record.

```
DEFINE read.price PROGRAM
PROGRAM
VARIABLE fil.unit INTEGER
TRAP ON ERROR
fil.unit = FILEOPEN( ARG(1) READ)
LIMIT month TO &ARG(2)
NEXT:
FILEREAD fil.unit -
  WIDTH 8 product -
  WIDTH 4 BINARY price
FILECLOSE fil.unit
RETURN
error:
IF fil.unit EQ NA
  THEN RETURN
IF ERRORNAME NE 'attn' AND ERRORNAME NE 'quit'
  THEN DO
    SHOW JOINCHARS('Record ' RECNO(fil.unit) ' is Invalid.')
    TRAP ON ERROR
    GOTO NEXT
  DOEND
FILECLOSE fil.unit
END
```

Example 13–11 Preprocessing File Data Before Assigning to a Workspace Object

You can also process the data in each field before assigning it to a variable or dimension in the analytic workspace. Suppose your data file has product identifiers that are six-digit numbers, and your analytic workspace has a `product` dimension whose values are these same product numbers, preceded by a "P." You can process the identifiers in the file by adding a "P" at the beginning of each value.

```
FILEREAD unit COLUMN 1 WIDTH 6 APPEND LSET 'p' product
```

Example 13–12 Maintaining Conjoint Dimensions with File Data

To maintain a conjoint dimension with FILEREAD, you first maintain its base dimensions by appending any new values from the input file. Then you assign the resulting combination of base dimension values to the conjoint dimension. The following example gets base dimension values from two separate fields, appends the values to the base dimensions, then appends the combination to the conjoint dimension.

```
FILEREAD unit APPEND proddist = <W 8 product, W 8 district>
```

In the preceding statement, the angle brackets automatically cause APPEND to apply to all three dimensions. When you do not want to add new values to the base dimensions, but want only to add new conjoint dimension values, you must explicitly state the keyword MATCH or change the order of the target objects, as shown in the two following statements.

```
fileread unit APPEND proddist = <W 8 MATCH product,W 8 MATCH district>
```

or

```
FILEREAD unit W 8 product W 8 district APPEND proddist = <product, district>
```

Example 13–13   Reading Data From a Structured PRN File

Suppose you want to read data from a structured PRN file with values of the `product` dimension in field two, values of the `district` dimension in field three, and several months of sales values beginning in field six. You could read the first 10 records in the file with the following statement.

```
FILEREAD unit STOPAFTER 10 STRUCTURED FIELD 2 product -
   district FIELD 6 ACROSS month: sales
```

FILESET

The FILESET command sets the paging attributes of a specified fileunit.

Syntax

FILESET *fileunit attrib-arg1 exp1* [*attrib-argN expN* ...]

where:

attrib-arg is one of the following:

BMARGIN

LINENUM

LSIZE

ORIGIN

PAGENUM

PAGEPRG

PAGESIZE

PAGING

PAUSEATPAGEEND

TABEXPAND

TMARGIN

Arguments

fileunit
A fileunit number that is assigned to a file opened in a previous call to the FILEOPEN function or by the OUTFILE command. You can set attributes only for an open file. An attribute argument specifies the file characteristic to change. The attribute must be appropriate for the fileunit specified; otherwise, Oracle OLAP returns an error. You can set several attributes in one FILESET command by listing the attribute name and its new value in pairs.

BMARGIN
Specifies the number of blank lines that make up the bottom margin.

LINENUM
Specifies the current line number. Resets after each pagebreak when PAGING is on; otherwise, keeps incrementing.

LSIZE
Specifies the maximum line length for text output files, or the record length for binary input files.

ORIGIN
Specifies the type of system on which the file was created. See "ORIGIN Attribute" on page 13-42.

PAGENUM
Specifies the current page number.

PAGEPRG
Specifies the OLAP DML program that produces page titles and headings when output is paged.

PAGESIZE
Specifies the number of lines on each page.

PAGING
Specifies if the output is formatted in pages.

PAUSEATPAGEEND
Specifies if Oracle OLAP should pause after each page.

TABEXPAND
Specifies if tab characters should be expanded. See "Tab Treatment" on page 13-42.

TMARGIN
Specifies the number of blank lines that make up the top margin.

exp
An expression that contains the new value for the attribute being set. The data type of the expression must be the same as the data type of the attribute.

Notes

OUTFILE Command

When you use an OUTFILE *filename* command, it is easier to set paging attributes for the file by using the regular Oracle OLAP paging options from the command line instead of FILESET. When you prefer FILESET, you can identify the file by simply using the OUTFILEUNIT option. For example, these statements

```
OUTFILE FILENAME
PAGING = YES
```

are equivalent to these statements,

```
OUTFILE FILENAME
FILESET OUTFILEUNIT PAGING YES
```

Multiple Open File Units

You can have as many files open at the same time as your operating system allows.

Tab Treatment

When you want tab characters in the source file to be expanded when read by FILEGET or FILEREAD, you can specify the TABEXPAND attribute. When TABEXPAND is zero, tab characters will not be expanded. A value greater than 0 indicates the distance, in bytes, between tab stops. The default value of TABEXPAND is `8`.

ORIGIN Attribute

The default value of the ORIGIN attribute reflects the system you are currently working on, so you must set ORIGIN when the file originated on a different system. The setting of ORIGIN affects how data reading statements interpret the files. For example, data reading statements use this information to decide whether bytes of binary data need to be reversed, and so forth. Table 13–2, " Values for ORIGIN Clause of FILESET" will help you make the right choice. When your system is not listed, try using PC or HP as the value of ORIGIN. When one value does not work, the other one should.

Table 13–2 Values for ORIGIN Clause of FILESET

Value	Hardware or Operating System
ALPHA	Any DEC workstation using an Alpha processor
AVMS	A DEC Alpha processor running on VM
HP	HP MPE XL
HPS700	HP Series 700 Workstation
HPS800	HP Series 800 Workstation
IBMPC	An Intel processor running DOS, Windows, or Windows N
INTEL5	Any Intel5 processor running Unix
MIPS	Any MIPS machine
MVS	IBM MVS/TSO
NTALPHA	A DEC Alpha processor running Windows NT
PC	An Intel processor running DOS, Windows, or Windows NT
RS6000	Any IBM RS6000 processor running IBM AIX
SOLARIS2	Any workstation running Solaris2
SUNOS4	Any workstation running SunOS4
VAX	VAX VMS (floating point in G format only)
VM	VM/CMS

Examples

Example 13–14 Setting Paging for a Report

When you are sending output to a report in a disk file, you might set the following attributes to indicate that the report is organized in pages and that the first page is 1.

```
DEFINE fil.unit INTEGER
fil.unit = FILEOPEN('REPORT' WRITE)
FILESET fil.unit PAGING YES PAGENUM 1
```

FILEVIEW

The FILEVIEW command works in conjunction with the FILENEXT function to read one record at a time of an input file, process the data, and store the data in Oracle OLAP dimensions and variables according to the descriptions of the fields. Use FILENEXT to read the record, then use one or more FILEVIEW commands to process the fields as needed. FILEVIEW has the same attributes as FILEREAD for specifying the format of the input and the processing of the output.

Syntax

FILEVIEW *fileunit* [*field-desc*...]

Arguments

fileunit
A fileunit number that is assigned to a file opened for reading (READ mode) in a previous call to the FILEOPEN function.

field-desc
A field description describes how to process one or more fields in each input record. Attributes in the field description specify how to format the input data. FILEVIEW reads each field according to the format specification and assigns the input data to the specified object. You can assign the data to the object directly or you can specify an expression to manipulate the data before you assign it. One field description can assign data from one input field to one Oracle OLAP object. Alternately you can use the ACROSS keyword to assign several values in the input record to a variable that is dimensioned by the fastest varying dimension. Because field attributes include the column number in the input record, you can process input fields in any order.

The format for the field description is as follows.

[[*pos*] ACROSS *dim* [*limit-clause*]:] [*attribs*] *object* [= *exp*]

pos
One or more attributes that specify the position in the record where Oracle OLAP will begin reading the fields specified by the ACROSS description. To specify the position, use the attributes FIELD, SPACE, and COLUMN (see FILEREAD). The *pos* argument is optional when the series of fields specified in the ACROSS phrase begins in the next field for structured records, or the next byte for ruled records.

ACROSS *dim* [*limit-clause*]:
Specifies the dimension of one or more data fields in the input record. FILEVIEW assigns the data in the fields to a variable according to the values in the current status of *dim*. Typically, each field description processes one value. However, using the ACROSS keyword, you can process one input value for each dimension value currently in the status.

limit-clause lets you temporarily change the status of the fastest varying dimension, as long as you are not in a FOR loop over the that dimension. The new status is in effect only for the duration of the FILEVIEW command. The format of *limit-clause* is as follows.

[ADD|COMPLEMENT|KEEP|REMOVE|<u>TO</u>|INSERT] *valuelist*

To specify the temporary status, insert any of the LIMIT keywords (the default is TO) along with an appropriate list of dimension values or related dimensions. You can use any valid LIMIT clause (see the LIMIT command for further information). The following example limits month to the last six values, no matter what the current status of month is.

```
across month last 6: units
```

attribs
One or more attributes that tell Oracle OLAP the position in the record and the format of the input data. (See FILEREAD for an explanation of the available attributes.)

***object* [= *exp*]**
An Oracle OLAP variable, dimension, or relation to which the input data is assigned. When = *exp* is missing, the data is assigned implicitly to the object. When = *exp* is present, the data is processed according to the expression and then assigned to *object*.

You can use the keyword VALUE to represent the value in a particular field of a record. VALUE represents the data from the file, formatted according to the FILEREAD attributes you use. When the field in the record is blank, FILEREAD considers its value to be `NA`. By default, the data type of VALUE is the data type of the target object. However, you can specify a different data type with an attribute keyword. VALUE can be used more than once to represent different values from the same record. For each instance, specify the column from which to read each value, as shown in the following example code.

```
sales = if col 1 w 1 text value eq 'A' then col 2 w 8 value -
   else col 10 w 8 value
```

In this example, the default data type of VALUE is decimal, which is the data type of the target object `sales`. However, the first instance of VALUE is compared to a text expression, so you must use the attribute TEXT to specify its data type.

SELECT *exp*
The SELECT field-description keyword processes varying record types (such as records with different structures or different target objects) with one FILEVIEW command. Within a field description, you can use the following syntax:

```
SELECT exp -
[WHEN exp action [WHEN exp action ...]] -
[ELSE action]
IF bool-exp THEN action [ELSE action]
DO
  field-desc
  [field-desc]
  ...
DOEND
```

The *action* argument is one of the following:

- NULL (no action occurs)
- *field-description*, including nested IF and SELECT statements.

SELECT evaluates the first expression, which may contain invocations of the VALUE keyword, and which has a default data type of TEXT. SELECT then sequentially compares the result with the WHEN expressions. When the first match is found, the associated action occurs. When no match is found, the ELSE action (if specified) occurs.

IF *bool-exp*
The IF field-description keyword processes varying record types (such as records with different structures or different target objects) with one FILEVIEW command. Within a field description, you can use the following syntax:

```
IF bool-exp THEN action [ELSE action]
```

action is the same as described for SELECT.

IF evaluates the Boolean expression, which may contain invocations of the VALUE keyword. IF performs the THEN action when the expression is TRUE or the ELSE action, if specified, when the expression is FALSE. No action occurs when the expression is NA.

Notes

Related OLAP DML Statements

Before you can process data from a file with FILEVIEW, use the FILEOPEN function to open the file for reading (READ mode). Use the FILENEST function to read a record for processing. When you are finished, close the file with the FILECLOSE command.

Record Order

FILEVIEW can process the fields in a record in any order. List the field descriptions in the order you want to process them, identifying the fields with explicit column numbers. You can also use several FILEVIEW commands on the same record to do different processing depending on the data you find in the record.

Alternative OLAP DML Statement

When you want to process all the records in a file in the same way, without complicated optional processing, the FILEREAD command is easier to use.

Dimension Values

When the target object of a field description is a dimension, you can specify whether the data in the file will be used to add values to the dimension or not. The dimension attributes are MATCH and APPEND:

- MATCH -- Any value encountered in a field must already be a value of the dimension. FILEVIEW temporarily limits status to that value. When it is not already a dimension value, FILEVIEW generates an error. After the FILEVIEW command, the dimension status is the same as before the command.
- APPEND -- The values in the field can already exist or they can be new. When the value exists, FILEVIEW limits status to that value; when it does not, FILEVIEW adds the value and then limits status. The dimension is limited to ALL when FILEVIEW is finished.

For more information about handling dimensions, see FILEREAD.

Error Handling

When FILEVIEW encounters an error, you can control what happens with an error trap and appropriate processing. Errors can be caused by attempts to convert data to an incompatible data type or by encountering invalid dimension values. You can use the FILEERROR function to find out what type of error occurred. After processing the error, you can use GOTO to branch back to the FILEVIEW command.

Attribute List

For a complete list of the attributes for FILEVIEW and FILEREAD and for more information about processing `NA` values, reading date values, reading multidimensional data, storing NTEXT values, and specifying attributes, see FILEREAD.

FILEVIEW with Composites

The discussions of composites and variables dimensioned by composites in FILEREAD also apply to FILEVIEW.

Examples

Example 13–15 Varying Months

The following program processes an input file that contains sales data for a variable number of months. The file has the following records:

Record 1 -- Title (to be ignored).

Record 2 -- Column labels. Month names are used to set the status of `month`. The number of months is unknown before processing the file.

Record 3 -- Dashes underlining column labels (to be ignored).

Record 4 -- Blank.

Record 5 to end -- There are three record types for Record 5—one for each type of line to be read.

One record type for Record 5 represents a detail line with the contents shown in the following table.

Column	Width	Format	Data
1	8	Symbolic	District name or blank (When the district name is blank on a detail line, the most recent line containing a district determines the current district.)
10	10	Symbolic	Product name

Column	Width	Format	Data
21	10	Symbolic	Sales for first month
33	10	Symbolic	Sales for second month
45	To end of record	Symbolic	Sales for additional months

Another record type in Record 5 represents a totals line with the contents shown in the following table.

Column	Width	Data
1	18	Blank
21	To end of record	Totals

A third record type of Record 5 contains dashes or equal signs as row separators as illustrated in the following table.

Column	Width	Data
1	18	Blank
21	To end of record	Dashes (--) or equal signs (==)

This is a report of the sample file.

```
                          This is the Title
                  Jan95      Feb95      Mar95      Apr95
                  ---------- ---------- ---------- ----------

Boston   Tents       32,153.52  32,536.30  43,062.75  57,608.39
         Canoes      66,013.92  76,083.84  91,748.16 125,594.28
         Racquets    52,420.86  56,837.88  58,838.04  69,338.88
         Sportswear  53,194.70  58,913.40  62,797.80  67,869.10
         Footwear    91,406.82  86,827.32 100,199.46 107,526.66
                  ---------- ---------- ---------- ----------
                  295,189.82 311,198.74 356,646.21 427,937.31
                  ---------- ---------- ---------- ----------
Atlanta  Tents       40,674.20  44,236.55  51,227.06  78,469.37
           .
           .
           .
         Footwear    53,284.54  57,331.30  59,144.76  70,516.98
                  ---------- ---------- ---------- ----------
                  231,780.46 245,812.33 275,622.68 355,784.92
                  ---------- ---------- ---------- ----------
                   1,813,326  1,985,731  2,185,174  2,638,409
                  ========== ========== ========== ==========
```

The program figures out which months are covered in the file, then reads the detail lines and assigns the sales data to the appropriate district and month. The program ignores total lines and underlines when FILEVIEW finds columns 1 through 19 blank. The program takes the name of the data file as an argument.

```
DEFINE salesdata PROGRAM
LD Store Several Months of Sales Data in an Analytic Workspace
PROGRAM
VARIABLE fil.unit INTEGER
VARIABLE flag BOOLEAN
VARIABLE mname TEXT
VARIABLE label TEXT
VARIABLE savedist TEXT

TRAP ON error NOPRINT
PUSH month district
fil.unit = FILEOPEN(ARG(1) READ)

IF FILENEXT(fil.unit) NE YES    "Skip Record 1
  THEN SIGNAL noread
IF FILENEXT(fil.unit) NE YES    "Process Record 2
  THEN SIGNAL noread
FILEVIEW fil.unit COLUMN 21 ACROSS month: -
  WIDTH 10 mname = JOINLINES( mname VALUE)
LIMIT month TO mname
IF FILENEXT(fil.unit) NE YES     "Skip Record 3
  THEN SIGNAL noread
IF FILENEXT(fil.unit) NE YES     "Skip Record 4
  THEN SIGNAL noread

WHILE FILENEXT(fil.unit)  "Process Record 5 To End Of File
   DO
   "Store Value In Local Label Variable
   FILEVIEW fil.unit COLUMN 1 WIDTH 18 label
   IF label NE NA         "Check For NA (Blank Field)
     THEN DO              "Get District Value If Present
      IF EXTCHARS(label, 1, 8) NE '      '
         "Set District Status
         THEN savedist = BLANKSTRIP(EXTCHARS(label, 1, 8))
       FILEVIEW fil.unit -
         COLUMN 1 WIDTH 8 district = IF VALUE NE NA THEN -
            VALUE ELSE savedist -
         COLUMN 10 WIDTH 10 product -
         COLUMN 19 ACROSS month: WIDTH 10 SPACE 2 -
            SCALE 2 newsales
      DOEND
NEXT:
  DOEND

FILECLOSE fil.unit
POP month district
RETURN
error:
IF fil.unit EQ NA
  THEN SHOW JOINCHARS('Can\'t Open Data File ' ARG(1) '.')
ELSE IF ERRORNAME NE 'attn' AND ERRORNAME NE 'QUIT'
  THEN DO
    SHOW JOINCHARS('RECORD ' RECNO(fil.unit) ' is invalid.')
    GOTO NEXT
  DOEND
ELSE IF ERRORNAME EQ 'noread'
  THEN DO
    SHOW 'File Too Short.'
    FILECLOSE fil.unit
  DOEND
ELSE DO
  SHOW 'Data Import Interrupted.'
  FILECLOSE fil.unit
DOEND
POP month district
RETURN
```

Example 13–16   Additional Processing

When you want to save the dimension value that FILEVIEW read for display or further processing, you can read the field again and save the value in a variable. These lines in a program display the name of the month that FILEVIEW read. The FILEVIEW command saves the month value in column 1 in a variable called `mname`.

```
WHILE FILENEXT(fil.unit)
DO
  FILEVIEW fil.unit WIDTH 8 month WIDTH 5 INTEGER units -
       COLUMN 1 WIDTH 8 mname
  SHOW mname PROMPT
DOEND
```

Example 13–17   Using the VALUE Keyword as a Function

Suppose you want to read and report data from a disk file similar to the following, named `numbers.dat`, which has columns 15 characters wide.

```
            1.0            2.0            3.0            4.0            5.0
           -1.0           -2.0           -3.0           -4.0           -5.0
```

```
         0.0              0.0              1.43900000E+03                    1.39900000E+03
```

You can read this data using the VALUE keyword as a function with FILEVIEW in a program similar to the following one (named `try`). However, this first example does not work. The FILEVIEW command will skip fields. The reason for the data skipping is that each time FILEREAD fetches a field from the current record, it updates the column pointer to point *past* the field. When the next fetch does not specify a position (using the COLUMN, SPACE, or FIELD attribute), data will be read from the default position established by the previous fetch. This usually desirable behavior will not work when more than one fetch is needed to perform a single assignment. This happens when the VALUE function is coded twice in the same IF...THEN...ELSE block, as shown here. The NAMELIST and DIRLIST attributes return one value for multiple versions of a particular file name in the directory. The NAMELIST attribute also returns only one value for multiple files in the directory with the same root file name but different file types.

```
DEFINE try PROGRAM
PROGRAM
VARIABLE funit INTEGER
DEFINE dvar VARIABLE DECIMAL <year>
PUSH year
LIMIT year TO LAST 5
TRAP ON ERROR
funit=FILEOPEN('numbers.dat' R)

WHILE FILENEXT(funit)
   DO
   FILEVIEW funit ACROSS year: W 15 TEXT dvar = -
      IF FINDCHARS(VALUE, 'e') EQ 0 -  "Incorrect Use of Value
      THEN CONVERT(VALUE, dec) -       "Results in Skipped
      ELSE -9999.99                    "Fields
      REPORT DOWN year dvar
   DOEND
error:
FILECLOSE funit
DELETE dvar
POP year
END
```

When you execute the `try` program,

```
try
```

the output skips numbers, as in the following.

```
YEAR              DVAR
-------------  ----------
Yr93                 2.00
Yr94                 4.00
Yr95                   NA
Yr96            -9,999.99
Yr97            -9,999.99

YEAR              DVAR
-------------  ----------
Yr93                -2.00
Yr94                -4.00
Yr95                   NA
Yr96            -9,999.99
Yr97            -9,999.99

YEAR              DVAR
-------------  ----------
Yr93                 0.00
Yr94            -9,999.99
Yr95            -9,999.99
Yr96            -9,999.99
Yr97            -9,999.99
```

However, when the SPACE attribute is used to make the second VALUE back up some distance so it reads the same field that the first VALUE read, everything works fine. SPACE can be used in the preceding sample program by changing the THEN clause to the following:

```
THEN CONVERT(SPACE -15 VALUE, dec) -
```

Now when you execute the program,

```
try
```

the output will look like this.

```
YEAR                DVAR
-------------  ----------
Yr93                 1.00
Yr94                 2.00
Yr95                 3.00
Yr96                 4.00
Yr97                 5.00

YEAR                DVAR
-------------  ----------
Yr93                -1.00
Yr94                -2.00
Yr95                -3.00
Yr96                -4.00
Yr97                -5.00

YEAR                DVAR
-------------  ----------
Yr93                 0.00
Yr94                 0.00
Yr95            -9,999.99
Yr96            -9,999.99
Yr97            -9,999.99
```

FILTERLINES

The FILTERLINES function applies a filter expression that you create to each line of a multiline text expression.

Return Value

TEXT or NTEXT

This function accepts TEXT values and NTEXT values as arguments. The data type of the return value depends on the data type of the values specified for the arguments:

- When all arguments are TEXT values, the return value is TEXT.
- When all arguments are NTEXT values, the return value is NTEXT.
- When the arguments include both TEXT and NTEXT values, the function converts all TEXT values to NTEXT before performing the function operation, and the return value is NTEXT.

Syntax

FILTERLINES(*source-expression filter-expression*)

Arguments

source-expression
A multiline text expression whose lines should be modified according to *filter-expression*.

filter-expression
An expression to be applied as a filter to each line of *source-expression*. The terms of the filter expression dictate the processing that FILTERLINES will perform on each line of the source expression.

The filter expression may produce `NA`, which means that there is no line in the resulting text expression corresponding to the current line of the source expression.

You can use the keyword VALUE in your filter expression to represent the current line of the source expression.

Notes

The Result of FILTERLINES

FILTERLINES returns a text expression composed of the lines that result from the action of the filter expression on each line of the source expression. The filter expression may return multiline text for any or all of the input source lines. None of these lines will be acted on again by the filter expression.

Examples

Example 13–18 Removing Extension From File Names

The following example shows how FILTERLINES could be used on a list of file names to produce a list of those same file names without extensions.

With a multiline text variable named `filelist` that evaluates to

```
myfile1.txt
file2.txt
myfile3
file4.txt
```

the statement

```
SHOW FILTERLINES(FILELIST -
   IF FINDCHARS(VALUE '.') GT 0 -
      THEN EXTCHARS(VALUE 1 FINDCHARS(VALUE '.') -1) -
      ELSE VALUE)
```

produces the following output.

```
myfile1
file2
myfile3
file4
```

FINDBYTES

The FINDBYTES function returns the byte position of the beginning of a specified group of bytes within a text expression.

Return Value

INTEGER

Syntax

FINDBYTES(*text-expression*, *bytes* [*starting-pos* [LINENUM]])

Arguments

text-expression
The text expression in which you are searching for the specified bytes. The value of *text-expression* can be a multiline value. In this case, FINDBYTES searches all lines for the specified bytes. The match must be exact, including a match of upper- and lowercase characters.

bytes
The group of bytes for which you are searching. When *bytes* is a multiline value, FINDBYTES ignores all lines except the first one.

When *bytes* is not found in *text-expression,* FINDBYTES returns zero. When the group of bytes occurs more than once, FINDBYTES returns the position of its first occurrence.

starting-pos
An INTEGER expression that specifies the byte position where the search in *text-expression* should start. The default is at position 1 (the first byte) in *text-expression*.

LINENUM
Specifies that FINDBYTES should return the line number instead of the byte position of the beginning of the specified text.

Notes

Single-Byte Characters

When you are using a single-byte character set, you can use the FINDCHARS function instead of the FINDBYTES function.

NTEXT Data Type

This function does not accept NTEXT arguments, because it is oriented toward byte-manipulation instead of character manipulation. It always returns values of type TEXT. When you must use this function on NTEXT values, use the CONVERT or TO_CHAR function to convert the NTEXT value to TEXT.

Examples

Example 13–19 Finding the Starting Position of a Byte Group

This example shows how to find the starting position of various groups of bytes in the literal TEXT value `hellotherejoe`.

The statement

```
SHOW FINDBYTES('hellotherejoe', 'joe')
```

produces the following output.

```
11
```

The statement

```
SHOW FINDBYTES('hellotherejoe', 'al')
```

produces the following output.

```
0
```

FINDCHARS

The FINDCHARS function returns the character position of the beginning of a specified group of characters within a text expression.

Return Value

INTEGER

Syntax

FINDCHARS(*text-expression*, *characters* [*starting-pos* [LINENUM]])

Arguments

text-expression
The text expression in which you are searching for the specified characters. *Text-expression* can be a multiline value. In this case, FINDCHARS searches all lines for the specified characters. The match must be exact, including a match of upper- and lowercase characters. See "TEXT and NTEXT" on page 13-61.

characters
The group of characters for which you are searching. When *characters* is a multiline value, FINDCHARS ignores all lines except the first one.

When *characters* is not found in *text-expression*, FINDCHARS returns zero. When the group of characters occurs more than once, FINDCHARS returns the position of its first occurrence.

starting-pos
An INTEGER expression that specifies the character position where the search in *text-exp* should start. The default is at position 1 (the first character) in *text-exp*.

LINENUM
Specifies that FINDCHARS should return the line number instead of the character position of the beginning of the specified text.

Notes

multibyte Characters

When you are using a multibyte character set, you can use the FINDBYTES function instead of the FINDCHARS function.

TEXT and NTEXT

FINDCHARS accepts TEXT values and NTEXT values as arguments. When only one argument is NTEXT, then FINDCHARS automatically converts the other argument to NTEXT before performing the function operation.

Examples

Example 13–20 Finding the Starting Position of a Character Group

This example shows how to find the starting position of various groups of characters in the literal TEXT value `hellotherejoe`.

The statement

```
SHOW FINDCHARS('hellotherejoe', 'joe')
```

produces the following output.

```
11
```

The statement

```
SHOW FINDCHARS('hellotherejoe', 'al')
```

produces the following output.

```
0
```

FINDLINES

The FINDLINES function determines the position of one or more lines in a multiline text expression.

Return Value

INTEGER

Syntax

FINDLINES(*text-expression*, *lines*)

Arguments

text-expression
A text expression within whose values you want to locate a certain line or group of lines. FINDLINES searches *text-expression* for the specified lines. The match must be exact, including a match of uppercase and lowercase characters. See "TEXT and NTEXT" on page 13-63.

lines
A second text expression containing the line(s) for which you are searching. When *lines* is not found in *text-expression,* FINDLINES returns 0. When *lines* occurs more than once, FINDLINES returns the line number of its first occurrence.

Notes

Finding Multiple Lines

When you specify two or more lines, FINDLINES searches for all the specified lines as a single continuous block in *text-expression.* When all the lines occur in *text-expression,* but are not in a continuous block, FINDLINES returns 0 (not found).

NA Values

When the value of *text-expression* is `NA`, FINDLINES returns `NA`.

TEXT and NTEXT

FINDLINES accepts TEXT values and NTEXT values as arguments. When only one argument is NTEXT, then FINDLINES automatically converts the other argument to NTEXT before performing the function operation.

Examples

Example 13–21 Finding Two Sequential Lines

This example shows how to find the location of the two lines "products" and "services" in a multiline value in a TEXT variable called `newlist`. The `newlist`T variable has the following values.

```
salespeople
products
services
regions
priorities
```

The characters "`\n`" in the *lines* argument to the following FINDLINES function call indicates a line break to show that "product" and "services" are separate lines.

```
SHOW FINDLINES(newlist, 'products\nservices')
```

The result of this statement is

```
2
```

FINTSCHED

The FINTSCHED function calculates the interest portion of the payments on a series of fixed-rate installment loans that are paid off over a specified number of time periods. For each time period, you specify the amount of the loans incurred during that time period and a single interest rate that will apply to those loans over their lifetime.

Return Value

DECIMAL

Syntax

FINTSCHED(*loans*, *rates*, *n*, [*time-dimension*])

Arguments

loans

A numeric expression that contains the initial amounts of the loans. When *loans* does not have a time dimension, or when *loans* is dimensioned by more than one time dimension, the *time-dimension* argument is required.

rates

A numeric expression that contains the interest rates charged for *loans.* When *rates* is a dimensioned variable, it can be dimensioned by any dimension, including a different time dimension. When *rates* is dimensioned by a time dimension, you specify the interest rate in each time period that will apply to the loans incurred in that period. The interest rate for the time period in which a loan is incurred applies throughout the lifetime of that loan. The rates are expressed as decimals; for example, a 5 percent rate is expressed as `.05`.

n

A numeric expression that specifies the number of payments required to pay off the loans in the series. The *n* expression can be a dimensioned variable, but it cannot be dimensioned by the time dimension argument. One payment is made in each time period of the time dimension by which *loans* is dimensioned or in each time period of the dimension specified in the *time-dimension* argument. For example, one payment is made each month when *loans* is dimensioned by MONTH.

time-dimension

The name of the dimension along which the interest payments are calculated. When the time dimension has a type of DAY, WEEK, MONTH, QUARTER, or YEAR, the *time-dimension* argument is optional, unless *loans* has more than one time dimension.

Notes

The Dimensions of the Result

The result returned by the FINTSCHED function is dimensioned by the union of all the dimensions of *loans*, *rates*, *n*, and the dimension used as the *time-dimension* argument.

Time Period Results

FINTSCHED calculates the result for a given time period as the sum of the interest due on each loan that is incurred or outstanding in that period.

NA Mismatch Error

When *loans* has a value other than `NA` and the corresponding value of *rates* is `NA`, an error occurs.

NASKIP Option

FINTSCHED is affected by the NASKIP option. When NASKIP is set to `YES` (the default), and a loan value is `NA` for the affected time period, the result returned by FINTSCHED depends on whether the corresponding interest rate has a value of `NA` or a value other than `NA`. Table 13–3, " Effect of NASKIP When Loan or Rate Values are NA for a Time Period" illustrates how NASKIP affects the results when a loan or rate value is `NA` for a given time period.

Table 13–3 Effect of NASKIP When Loan or Rate Values are NA for a Time Period

Loan Value	Rate Value	Result When NASKIP = YES	Result When NASKIP = NO
Non-`NA`	`NA`	Error	Error
`NA`	Non-`NA`	Interest values (`NA` loan value is treated as zero)	`NA` for the affected time periods
`NA`	`NA`	`NA` for affected time periods	`NA` for the affected time periods

As an example, suppose a loan expression and a corresponding interest expression both have NA values for 1997 but both have values other than NA for succeeding years. When the number of payments is 3, FINTSCHED returns NA for 1997, 1998, and 1999. For 2000, FINTSCHED returns the interest portion of the payment due for loans incurred in 1998, 1999, and 2000.

Time Dimensions

The FINTSCHED calculation begins with the first time dimension value, regardless of how the status of that dimension may be limited. For example, suppose *loans* is dimensioned by `year`, and the values of `year` range from `Yr95` to `Yr99`. The calculation always begins with `Yr95`, even when you limit the status of `year` so that it does not include `Yr95`.

However, when *loans* is not dimensioned by the time dimension, the FINTSCHED calculation begins with the first value in the current status of the time dimension. For example, suppose *loans* is not dimensioned by `year`, but `year` is specified as *time-dimension*. When the status of `year` is limited to `Yr97` to `Yr99`, the calculation begins with `Yr97` instead of `Yr95`.

Related Functions

The VINTSCHED function, which calculates the interest portion of the payments on a series of variable-rate loans, and the FPMTSCHED and VPMTSCHED functions, which calculate the payment schedules (principal plus interest) for fixed-rate and variable-rate loans.

Examples

Example 13–22 Calculating Interest

The following statements create two variables called `loans` and `rates`.

```
DEFINE loans DECIMAL <year>
DEFINE rates DECIMAL <year>
```

Suppose you assign the following values to the variables `loans` and `rates`.

```
YEAR              LOANS      RATES
-------------- ---------- ----------
Yr95               100.00       0.05
Yr96               200.00       0.06
Yr97               300.00       0.07
Yr98                 0.00       0.00
Yr99                 0.00       0.00
```

For each year, `loans` contains the initial value of the fixed-rate loan incurred during that year. For each year, the value of `rates` is the interest rate that will be charged for any loans incurred in that year; for those loans, this same rate is charged each year until the loans are paid off.

The following statement specifies that each loan is to be paid off in three payments, calculates the interest portion of the payments on the loans,

```
REPORT W 20 HEADING 'Payment' FINTSCHED(loans, rates, 3, year)
```

and produces the following report.

```
YEAR                        Payment
--------------   --------------------
Yr95                           5.00
Yr96                          15.41
Yr97                          30.98
Yr98                          18.70
Yr99                           7.48
```

The interest payment for 1995 is interest on the loan of $100 incurred in 1995, at 5 percent. The interest payment for 1996 is the sum of the interest on the remaining principal of the 1995 loan, at 5 percent, plus interest on the loan of $200 incurred in 1996, at 6 percent. The 1997 interest payment is the sum of the interest on the remaining principal of the 1995 loan, at 5 percent; interest on the remaining principal of the 1996 loan, at 6 percent; and interest on the loan of $300 incurred in 1997, at 7 percent. Since the 1995 loan is paid off in 1997, the payment for 1998 represents interest on the remaining principal of the 1996 and 1997 loans. In 1999, the interest payment is on the remaining principal of the 1997 loan.

FLOOR

The FLOOR function returns the largest whole number equal to or less than a specified number.

Return Value

NUMBER

Syntax

FLOOR(*n*)

Arguments

n
A whole number (NUMBER data type) that you specify.

Examples

Example 13–23 Displaying the Largest Integer Equal to or Less Than a Number

The following statements show results returned by the FLOOR function.

- The following `SHOW FLOOR` statement produces the result that follows it.

  ```
  SHOW FLOOR(15.7)

  15
  ```

- The following `SHOW FLOOR` statement produces the result that follows it.

  ```
  SHOW FLOOR(4)

  4
  ```

- The following `SHOW FLOOR` statement produces the result that follows it.

  ```
  SHOW FLOOR(-6.457)

  -7
  ```

FOR

The FOR command specifies one or more dimensions whose status will control the repetition of one or more statements. These statements, along with the FOR command itself, are often called a FOR loop. You can use the FOR command only within programs.

Syntax

FOR *dimension...*

statement

Arguments

dimension

One or more dimensions whose current status controls the repetition of one or more statements. The statements are repeated for each combination of the values of the specified dimensions in the current status. When two or more dimensions are specified, the first one varies the slowest. You can specify a composite instead of a dimension.

statement

The statement to be repeated. To repeat two or more statements, enclose them between DO and DOEND.

```
DO
  statement1
     ...
  statementN
DOEND
```

When you are repeating only one statement after FOR, you can omit DO and DOEND.

Notes

FOR Dimension

A FOR statement loops over the values in status of the specified dimension. After the last dimension value, dimension status is restored to what it was before the loop, and execution of the program resumes with the next statement.

Status Inside a Loop

The TEMPSTAT command limits the dimension you are looping over inside a FOR loop or inside a loop that is automatically generated by the REPORT command.

No Sorting

Because current status defines and controls a FOR loop, you cannot sort the FOR dimension within the loop.

Assignment Statements and Other Looping Statements

An OLAP DML assignment statement (SET), and some other OLAP DML statements automatically loop over dimension status and do so more efficiently than a FOR loop. Be careful not to cause extra looping by putting an assignment statement or one of these statements in a FOR loop.

Branching

You can use the BREAK, CONTINUE, and GOTO commands to branch within, or out of, a FOR loop, thereby altering the sequence of statement execution.

Nested FOR Commands

FOR commands can be nested within a FOR loop to any depth, as long as matching DO and DOEND commands are supplied where appropriate.

Related Statements

See also DO ... DOEND, IF...THEN...ELSE, WHILE, and RETURN.

Examples

Example 13–24 Repeating ROW Commands

In a report program, you want to show the unit sales of tents for each of three months. Use the following FOR command with a DO/DOEND sequence to repeat ROW commands and BITAND commands for each value of the `month` dimension.

```
LIMIT product TO tents
LIMIT month TO 'Jan96' TO 'Mar96'
ROW district
ROW UNDER '-' VALONLY name.product
BLANK
FOR month
    DO
      ROW INDENT 5 month WIDTH 6 UNITS
      BLANK
    DOEND
```

The program lines produce the following report.

```
BOSTON
3-Person Tents
--------------

     Jan96             307
     Feb96             209
     Mar96             277
```

Example 13–25 Using the FOR Command for Looping Over Values

The FOR command executes the commands in the loop for each value in the current status of the dimension. You must limit the dimension to the desired values before executing the FOR command. For example, you can produce a series of output lines that show the price for each product.

```
LIMIT month TO FIRST 1
LIMIT product TO ALL
FOR product
SHOW JOINCHARS('Price for ' product ': $' price)
```

Each output line has the following format.

```
Price for TENTS: $165.50
```

When your data is multidimensional, you can specify more than one dimension in a FOR command to control the order of processing. For example, you can use the following command to control the order in which dimension values of the `units` data are processed.

```
FOR month district product
   units = ...
```

When this assignment statement is executed, the `month` dimension varies the slowest, the `district` dimension varies the next slowest, and the `product` dimension varies the fastest. Thus, a loop is performed over all products for the first district before doing the next district, and over all districts for the first month before doing the next month.

Within the FOR loop, each specified dimension is temporarily limited to a single value while it executes the commands in the loop. You can therefore work with specific combinations of dimension values within the loop.

Example 13–26 Using DO/DOEND in a FOR Loop

When actual figures for unit sales are stored in a variable called `units` and projected figures for unit sales are stored in a variable called `units.plan`, then the code in your loop can compare these figures for the same combination of dimension values.

```
LIMIT month TO FIRST 1
LIMIT product TO ALL
LIMIT district TO ALL
FOR district product
   DO
     IF (units.plan - units)/units.plan GT .1
     THEN SHOW JOINCHARS(-
       'Unit sales for ' product ' in ' -
       district ' are not within 10% of plan.')
   DOEND
```

These lines of code are processed in the following manner.

1. The data is limited to a specific month.
2. All the districts and products are placed in status, and the FOR loop is entered.

3. In the FOR loop, the actual figure is tested against the planned figure. When the unit sales figure for `Tents` in `Boston` is more than 10 percent below the planned figure, then the following message is sent to the current outfile.

   ```
   Unit sales for TENTS in BOSTON are not within 10% of plan.
   ```

4. After processing all the products, the FOR loop is complete for the first district.
5. The loop is executed for the second district, and so on.

   Note that while the FOR loop executes, each dimension that is specified in a FOR command is limited temporarily to a single value. When you specify `district` in the FOR loop, but not `product`, then all the values of `product` are in status while the FOR loop executes. The IF...THEN...ELSE command then tests data for only the first value of the `product` dimension.

FORECAST

Use the FORECAST command to forecast data by one of three methods: straight-line trend, exponential growth, or Holt-Winters extrapolation. FORECAST performs the calculation according to the method you specify and optionally stores the result in a variable in your analytic workspace.

You can then execute FORECAST.REPORT to produce a standard report of the forecast. You can also use the INFO function to obtain portions of the results for use in your own customized reports or for further analysis.

Note: Most applications forecast data using a forecasting context rather than using the FORECAST command. See "Using a Forecasting Context" on page 13-77 for more information.

Syntax

FORECAST [LENGTH *n*] -

[METHOD {TREND|EXPONENTIAL|WINTERS PERIODICITY *p* [*argument...*]}] -

[TIME *dimension*] [FCNAME *name*] *time-series*

where:

argument is one or more of the following:

ALPHA *n*

BETA *n*

GAMMA *n*

STSMOOTHED *n* STSEASONAL *n-series* STTREND *n*

FCSMOOTHED *name*

FCSEASONAL *name*

FCTREND *name*

Arguments

LENGTH *n*
Specifies the number of periods to forecast. The default is zero. When you supply a LENGTH, you must also supply the FCNAME option.

METHOD TREND
Specifies that the forecasting technique is a straight-line extrapolation of historical data. (Default)

METHOD EXPONENTIAL
Specifies that the forecasting technique is an extrapolation of historical data using a constant period-to-period percentage growth.

METHOD WINTERS
Specifies that the forecasting technique is the Holt-Winters method, an extrapolation method that allows for both a linear trend and seasonal fluctuations in the data. Oracle OLAP first constructs three statistically related series for each time period of the historical data. (See "Holt-Winters Constructed Series" on page 13-78.) Then, Oracle OLAP produces a forecast from the three series for the specified number of periods into the future.

You can supply several arguments that affect the results of the Holt-Winters forecast. The only required one is PERIODICITY. For the others, Oracle OLAP chooses a reasonable value based on the data available.

PERIODICITY *p*
The length of the seasonal cycle, where *p* is an expression that specifies an integer greater than or equal to 2. For example, when the data you are analyzing has monthly values, then *p* is 12.

PERIODICITY is required when you use the METHOD WINTERS keyword.

ALPHA *n*
BETA *n*
GAMMA *n*
Smoothing constants for the first three series calculated for the Holt-Winters forecast (See "Holt-Winters Constructed Series" on page 13-78). ALPHA is for the smoothed data series; BETA is for the seasonal index series; and GAMMA is for the trend series. The value *n* is a decimal expression greater than 0 and less than or equal to 1. Each value is optional. When you omit one, Oracle OLAP calculates an optimal smoothing constant for that series that minimizes the Mean Absolute Percent Error of the one-period-ahead forecasts in the historical time periods.

STSMOOTHED *n* STSEASONAL *n-series* STTREND *n*
STSMOOTHED specifies the starting value of the smoothed data series (See "Holt-Winters Constructed Series" on page 13-78). The value *n* is a decimal expression greater than 0. When you specify STSMOOTHED, you must also specify STSEASONAL and STTREND. When you omit it, Oracle OLAP calculates a starting value.

STSEASONAL specifies the starting values for the seasonal index series (See "Holt-Winters Constructed Series" on page 13-78). *N-series* is an array of decimal values, one for each period in a seasonal cycle. The number of values needed is the same as the number specified for PERIODICITY (See "Holt-Winters Starting Values" on page 13-79). When you specify STSEASONAL, you must also specify STSMOOTHED and STTREND. When you omit it, Oracle OLAP calculates the starting values.

STTREND specifies the starting value of the trend series (See "Holt-Winters Constructed Series" on page 13-78). *N* is a decimal value. When you specify STTREND, you must also specify STSMOOTHED and STSEASONAL. When you omit it, Oracle OLAP calculates a starting value.

FCSMOOTHED *name*
FCSEASONAL *name*
FCTREND *name*
Numeric variables in which Oracle OLAP can store the data calculated for the smoothed data series, the seasonal index series, and the trend series (See "Holt-Winters Constructed Series" on page 13-78). The variable specified by *name* must have the TIME *dimension* as one of its dimensions. The series calculations produce DECIMAL results, but Oracle OLAP will convert the values to the data type of *name* before storing them. You can save any or all of the preliminary series. When you do not save a series, Oracle OLAP discards the values after completing the forecast.

TIME *dimension*
The name of the dimension considered to be the time dimension. The current status of *dimension* determines the number of periods of historical data used to calculate the forecast. The status of the time dimension must be an increasing, consecutive range of values. LENGTH specifies how many values immediately beyond this range will be forecast.

When *time-series* has only one dimension, the time dimension will default to that. When *time-series* has more than one dimension, and one of the dimensions has a type of DAY, WEEK, MONTH, QUARTER, or YEAR, the time dimension will default to that. Otherwise, you must specify the time dimension, even when the additional dimensions are limited to a single value. FORECAST only uses the first value in the status for dimensions other than the time dimension.

FCNAME *name*
The name of a numeric variable in which to store the values calculated by FORECAST. *Name* must be dimensioned by the time dimension; it can have other dimensions as well. When the data type of *name* is not decimal, FORECAST converts the values to the appropriate data type.

Fitted values, which correspond to the historical data, are stored in *name* for the current status of the time dimension. Forecasted values are stored in *name* for the number of periods specified by LENGTH. These forecasted periods immediately follow the current status of the time dimension.

For the Holt-Winters method, the fitted values are one-period-ahead forecasts calculated at the previous period. The final forecasted values are extrapolated from the fitted data.

For the TREND and EXPONENTIAL methods, FORECAST obtains the fitted values by evaluating the regression equation over the current status of the time dimension.

time-series
An expression that specifies the time series to be forecast. *Time-series* must be a numeric expression that is dimensioned by the time dimension. When *time-series* has other dimensions, FORECAST uses the first value only in their current status. The *time-series* is the historical data from which FORECAST calculates fitted and forecasted values. (See the explanation for FCNAME.)

Notes

Using a Forecasting Context
Instead of calculating a simple forecast using the FORECAST command, you can perform more complex forecasting using a forecasting context that you manipulate with the following OLAP DML statements:

1. FCOPEN function -- Creates a forecasting context.
2. FCSET command -- Specifies the characteristics of a forecast.
3. FCEXEC command -- Executes a forecast and populates Oracle OLAP variables with forecasting data.
4. FCQUERY function -- Retrieves information about the characteristics of a forecast or a trial of a forecast.

5. FCCLOSE command -- Closes a forecasting context.

Forecasting Multidimensional Expressions

When you want to forecast all the values of a multidimensional expression, you can use a program that puts the FORECAST command inside one or more FOR loops to loop over all the remaining dimensions of the expression.

Obtaining Portions of Results

YOu can obtain portions of the results of FORECAST for your own reports or further analysis, using an INFO statement.

Order of Arguments

You can specify the arguments for FORECAST in any order, except that *time-series,* the expression specifying the data to be forecast, must be last.

Time-series Data Handling

Each method has its own criteria for handling the input data specified in *time-series.*

- TREND -- Requires at least two values that are not `NA`; accepts zero and negative values; ignores `NA` values
- EXPONENTIAL -- Requires at least two positive values; ignores zero, negative, and `NA` values
- WINTERS -- Accepts zero and negative values; fills in `NA` values by calculating a weighted moving average

Zero Values

All methods allow zero values in the historical data, specified by *time-series,* but those time periods are excluded from the Mean Absolute Percent Error (MAPE) calculation.

Holt-Winters Constructed Series

The Holt-Winters forecasting method constructs three statistically related series, which are used to make the actual forecast. These series are:

1. The smoothed data series, which is the original data with seasonal effects and random error removed.
2. The seasonal index series, which is the seasonal effect for each period. A value greater than one represents a seasonal increase in the data for that period, and a value less than one is a seasonal decrease in the data. The Holt-Winters method

allows seasonal effects to vary over time, so there is a seasonal index value for every historical period.

3. The trend series, which is the change in the data for each period with the seasonal effects and random error removed. The Holt-Winters method allows the trend effect to vary over time, so there is a trend value for every historical period.

Holt-Winters Omitted Arguments

For the Holt-Winters method, when you omit the STSMOOTHED, STTREND, and STSEASONAL phrases, Oracle OLAP calculates the necessary starting values using an algorithm from *Statistical Methods for Forecasting* by Abraham and Ledolter (See "Further Reading on Forecasting" on page 12-33). You should let Oracle OLAP calculate the starting values when you have little experience with Holt-Winters forecasting.

Holt-Winters Starting Values

When you specify starting values, Oracle OLAP obtains the STSEASONAL starting values by unraveling the values to make a list. The list must have at least the number of values as specified by PERIODICITY. Any more values are ignored; fewer values cause an error. The STSEASONAL expression can be multidimensional and does not have to have the same dimensions as the historical data. (For information about the order of the list when a dimensioned expression is unraveled, see UNRAVEL.)

Getting Calculated Values

You can find out the values that Oracle OLAP calculates for ALPHA, BETA, and GAMMA and for STSMOOTHED, STSEASONAL, and STTREND by using the INFO function.

Further Reading

For additional information about forecasting and forecasting methods, we suggest the latest editions of the books listed in "Further Reading on Forecasting" on page 12-33.

Examples

Example 13–27 Using the EXPONENTIAL Method

The following statements create a variable called `fcst.sales`, limit the dimensions of the `sales` variable, use the EXPONENTIAL method to forecast sportswear sales for the Chicago district for 1997, and store the results of the calculation in `fcst.sales`.

```
DEFINE fcst.sales DECIMAL <month>
LIMIT product TO 'Sportswear'
LIMIT district TO 'Chicago'
LIMIT month TO 'Jan95' TO 'Dec96'
FORECAST LENGTH 12 METHOD EXPONENTIAL FCNAME fcst.sales -
time month sales
```

You can now execute FORECAST.REPORT as illustrated in "Report of Forecast Using the EXPONENTIAL Method" on page 13-81 to see the values that have been generated.

Example 13–28 Using the WINTERS Method

The following statements limit the `month` dimension, then calculate a forecast that takes into account seasonal influences, using the WINTERS method.

```
DEFINE fcst.sales DECIMAL <montH>
LIMIT month TO year 'Yr95' 'Yr96'
FORECAST LENGTH 12 METHOD WINTERS -
PERIODICITY 12, ALPHA .5, BETA .5, GAMMA .5 -
time month, FCNAME fcst.sales, sales
```

You can now execute FORECAST.REPORT as illustrated in "Report of Forecast Using the WINTERS Method" on page 13-82 to see the values that have been generated.

FORECAST.REPORT

The FORECAST.REPORT program produces a standard report of a forecast created using the FORECAST command.

The report shows the parameters of the forecast, including the forecast formula and Mean Absolute Percent Error, followed by a display of the forecasted values. To produce this report, type the following.

Syntax

FORECAST.REPORT

Examples

Example 13–29 Report of Forecast Using the EXPONENTIAL Method

Assume that you have performed the forecast illustrated in "Using the EXPONENTIAL Method" on page 13-80. Running the FORECAST.REPORT program for that forecast produces the following report.

```
                          Forecasting Analysis
                          ====================

                     Variable to Forecast: SALES
                       Forecast dimension: MONTH
                          Forecast method: EXPONENTIAL
              Mean absolute percent error: 16.64%

        Forecast Equation: SALES = 87718.0009541883 *
                                   (1.00553383457899 ** MONTH)

MONTH                   Actual Value    Fitted Value
--------------------    ------------    ------------
Jan95                      72,123.47       88,203.42
Feb95                      80,071.75       88,691.52
Mar95                      78,812.69       89,182.33
Apr95                      97,413.26       89,675.85
May95                      94,406.65       90,172.10
  ...                         ...             ...
Dec96                      72,095.02      100,140.38
  ...                         ...             ...
```

Example 13–30 Report of Forecast Using the WINTERS Method

Assume that you have performed the forecast illustrated in Example 13–28, "Using the WINTERS Method" on page 13-80. Running the FORECAST.REPORT program for that forecast produces the following report.

```
                          Forecasting Analysis
                          ====================

                     Variable to Forecast: SALES
                       Forecast dimension: MONTH
                          Forecast method: WINTERS
                                    Alpha: 0.50
                                     Beta: 0.50
                                    Gamma: 0.50
                              Periodicity: 12
             Mean absolute percent error: 0.20%

MONTH                        Actual Value    Fitted Value
--------------------         ------------    ------------
Jan95                           72,123.47       72,154.67
Feb95                           80,071.75       80,027.51
Mar95                           78,812.69       79,171.08
Apr95                           97,413.26       97,200.81
May95                           94,406.65       94,464.71
 ....                              ...             ...
Dec97                                           77,867.23
```

FPMTSCHED

The FPMTSCHED function calculates a payment schedule (principal plus interest) for paying off a series of fixed-rate installment loans over a specified number of time periods. For each time period, you specify the amount of the loans incurred during that time period and a single interest rate that will apply to those loans over their lifetime.

Return Value

DECIMAL

Syntax

FPMTSCHED(*loans*, *rates*, *n*, [*time-dimension*])

Arguments

loans
A numeric expression that contains the initial amounts of the loans. When *loans* does not have a time dimension, or when *loans* is dimensioned by more than one time dimension, the *time-dimension* argument is required.

rates
A numeric expression that contains the interest rates charged for *loans.* When *rates* is a dimensioned variable, it can be dimensioned by any dimension, including a different time dimension. When *rates* is dimensioned by a time dimension, you specify the interest rate in each time period that will apply to the loans incurred in that period. The interest rate for the time period in which a loan is incurred applies throughout the lifetime of that loan. The rates are expressed as decimals; for example, a 5 percent rate is expressed as `.05`.

n
A numeric expression that specifies the number of payments required to pay off the loans in the series. The *n* expression can be dimensioned, but it cannot be dimensioned by the time dimension argument. One payment is made in each time period of the time dimension by which *loans* is dimensioned or in each time period of the dimension specified in the *time-dimension* argument. For example, one payment each month is made when *loans* is dimensioned by `month`.

time-dimension
The name of the dimension along which the interest payments are calculated. When the time dimension for *loans* has a type of DAY, WEEK, MONTH, QUARTER, or YEAR, the *time-dimension* argument is optional, unless *loans* has more than one time dimension.

Notes

Dimensions of the Result

The result returned by the FPMTSCHED function is dimensioned by the union of all the dimensions of *loans* and *rates* and the dimension used as the *time-dimension* argument.

Time-Period Payment Calculation

FPMTSCHED calculates the payment for a given time period as the sum of the principal and interest due on each loan that is incurred or outstanding in that period.

NA Mismatch Error

When *loans* has a value other than `NA` and the corresponding value of *rates* is `NA`, an error occurs.

NASKIP Option

FPMTSCHED is affected by the NASKIP option. When NASKIP is set to `YES` (the default), and a loan value is `NA` for the affected time period, the result returned by FPMTSCHED depends on whether the corresponding interest rate has a value of `NA` or a value other than `NA`. Table 13–3, " Effect of NASKIP When Loan or Rate Values are NA for a Time Period" on page 13-65 illustrates how NASKIP affects the results when a loan or rate value is `NA` for a given time period.

As an example, suppose a loan expression and a corresponding interest expression both have `NA` values for 1997 but both have values other than `NA` for succeeding years. When the number of payments is 3, FPMTSCHED returns `NA` for 1997, 1998, and 1999. For 2000, FPMTSCHED returns the payment due for loans incurred in 1998, 1999, and 2000.

Time Dimensions

The FPMTSCHED calculation begins with the first time dimension value, regardless of how the status of that dimension may be limited. For example, suppose *loans* is dimensioned by `year`, and the values of `year` range from `Yr95` to `Yr99`. The calculation always begins with `Yr95`, even when you limit the status of `year` so that it does not include `Yr95`.

However, when *loans* is not dimensioned by the time dimension, the FPMTSCHED calculation begins with the first value in the current status of the time dimension. For example, suppose *loans* is not dimensioned by `year`, but `year` is specified as *time-dimension*. When the status of `year` is limited to `Yr97` to `Yr99`, the calculation begins with `Yr97` instead of `Yr95`.

Related Functions

The VPMTSCHED function, which calculates the payment schedule for a series of variable-rate loans, and the FINTSCHED and VINTSCHED functions, which calculate the interest portion of the payments on fixed-rate and variable-rate loans.

Examples

Example 13–31 Calculating a Payment Schedule

The following statements create two variables called `loans` and `rates`.

```
DEFINE loans DECIMAL <year>
DEFINE rates DECIMAL <year>
```

Suppose you assign the following values to the variables `loans` and `rates`.

```
year                loans        rates
-------------- ---------- ----------
Yr95               100.00       0.05
Yr96               200.00       0.06
Yr97               300.00       0.07
Yr98                 0.00       0.00
Yr99                 0.00       0.00
```

For each year, `loans` contains the initial value of the fixed-rate loan incurred during that year. For each year, the value of `rates` is the interest rate that will be charged for any loans incurred in that year; for those loans, this same rate is charged each year until the loans are paid off.

The following statement specifies that each loan is to be paid off in three payments, calculates the schedule for paying off the principal and interest on the loans,

```
REPORT W 20 HEADING 'Payment' FPMTSCHED(loans, rates, 3, year)
```

and produces the following report.

```
YEAR                          Payment
-------------- -------------------
Yr95                            36.72
Yr96                           111.54
Yr97                           225.86
Yr98                           189.14
Yr99                           114.32
```

The payment for 1995 is the principal due on the loan of $100 incurred in 1995, plus interest on the loan at 5 percent. The payment due in 1996 is the sum of the second payment on the loan incurred in 1995 (principal plus 5 percent interest), plus the first payment on the loan of $200 incurred in 1996 (principal plus 6 percent interest). The 1997 payment is the sum of the third and final payment on the loan incurred in 1995, the second of the three payments on the 1996 loan, and the first payment on the loan of $300 incurred in 1997 (principal plus 7 percent interest). Since the 1995 loan is paid off in 1997, the payment for 1998 covers the principal and interest for the 1996 and 1997 loans. The payment for 1999 is the final payment of principal and interest for the 1997 loan.

Example 13–32 Determining Monthly Payments

The following statement determines what the monthly payments would be on a $125,000 loan with an 8.75 percent annual interest rate,

```
SHOW FPMTSCHED(125000, .0875/12, 360, month)
```

and produces the following output.

```
983.38
```

FULLDSC

The FULLDSC program produces a report that lists the definition of one or more workspace objects, including the properties and triggers of the object(s).

Syntax

FULLDSC [*names*]

Arguments

names

The names of one or more workspace objects, separated by spaces or commas. FULLDSC shows the full definition of each object specified. When you omit this argument, FULLDSC shows the definition of all objects in the current status of the NAME dimension.

Notes

Output of FULLDSC

The FULLDSC program is an extension to the DESCRIBE command. That is, the object definition that you list with FULLDSC includes the definition components that are listed by the DESCRIBE command, followed by any properties that are assigned to the object. Each property is listed on its own line with the word PROPERTY, the name of the property, and its value.

Limiting the Objects Described

Normally, the status of NAME is ALL, so FULLDSC with no argument produces a report that includes the definitions of all objects in your current workspace. However, you can use the LIMIT command in combination with FULLDSC to report the definitions of a particular group of objects in your workspace. Use LIMIT first to limit the status of the NAME dimension to the names of the objects whose definitions you want to see. Then execute a FULLDSC command with no arguments to list the definitions.

Paginated Output

You can produce paginated output with the FULLDSC command by setting PAGING to `YES` before using FULLDSC.

Creating Objects with FULLDSC Output

You can use the output from the FULLDSC command to create objects in other workspaces, because each line of the output is a valid statement. For example, you can execute an OUTFILE command to send subsequent output to a file, and then execute the FULLDSC command. You can then access another workspace, and use the INFILE command to read the FULLDSC output. The same object will be created in that workspace.

The output produced by FULLDSC might not exactly reproduce the original PROPERTY commands that created the properties of the object because the original name and value expressions are not saved. In addition, FULLDSC sets the DECIMALS option to 255, which drops trailing zeros. See "Listing the Properties of a Variable" on page 13-88**.**

Examples

See Also: Example 24–5, "Describing Triggers" on page 24-15

Example 13–33 Listing the Properties of a Variable

This example produces a report of the full definition of the `actual` variable, to which the properties DECPLACE and REPPRG have been added. The statement

```
FULLDSC actual
```

produces the following output.

```
DEFINE ACTUAL VARIABLE DECIMAL <LINE DIVISION MONTH>
LD Actual $ Financials
PROPERTY 'DECPLACE' 4
PROPERTY 'REPPRG' 'qtrrep'
```

Suppose the DECPLACE property had been specified with the following statement, where PRPNAME is a variable whose value is DECPLACE.

```
PROPERTY prpname 4.00
```

The output from FULLDSC would be the same as that shown in the preceding example; the value 4.00 would be shown as 4. Therefore, when you created an object using the INFILE technique with the FULLDSC output, the newly created property value would have a type of INTEGER (based on the value 4) even though the original property value had a type of DECIMAL (based on the value 4.00). In most cases, this difference is immaterial, because the appropriate conversions are performed when the property values are used.

14

GET to IMPORT

This chapter contains the following OLAP DML statements:

GET

The GET function requests input from the current input stream. The input may be a single item of data, a dimension value, an analytic workspace object, or simply the next item in the input stream. The simplest form of the GET function requests a value of a certain data type.

GET(*datatype*)

GET also provides several arguments that verify the input.

Because GET is a function, it must be used in a command. It also may be used in an assignment statement to store the input in a variable for later use, or in a LIMIT command to set the status of a dimension. GET can be used in programs to request information necessary for the completion of the program.

Return Value

The return value depends on the input that you request, as described in the syntax.

Syntax

GET({RAW TEXT|[NEW|VALID|POSLIST] *input*} -

  [VERIFY *condition-exp* [IFNOT *result-exp*]])

where:

input is one of the following:

  dim-name

  NAME

  datatype

Arguments

dim-name

A text expression specifying the name of a dimension. When you specify *dim-name*, GET requests a value of this dimension as input and verifies that the input is a valid value of the dimension.

RAW TEXT
Specifies that GET should return the next item in the input stream exactly as it is entered. See "GET with RAW TEXT" on page 14-6.

NEW *dim-name*
The NEW keyword with the *dim-name* argument causes GET to request a new value for the dimension. When requesting a dimension value with NEW, GET verifies that the input is not already a value of the dimension.

VALID *dim-name*
The VALID keyword with the *dim-name* argument causes GET to request either a new value or an existing value of the dimension. When requesting a dimension value with VALID, GET verifies that the input is either an existing dimension value or a valid new dimension value.

POSLIST *dim-name*
The POSLIST keyword with the *dim-name* argument causes GET to request a dimension value identified by its position in the dimension. When requesting a dimension value with POSLIST, GET verifies that the input is an existing position number in the dimension. See "GET with POSLIST" on page 14-6.

NAME
Indicates that GET is requesting the name of an object in the current analytic workspace. When you specify NAME, GET verifies that the input is an object that exists in the current analytic workspace. The object name must not be enclosed in single quotes, and it must follow the rules for valid object names explained in the main DEFINE entry. GET automatically converts the object name to uppercase.

NEW NAME
The NEW NAME keywords cause GET to request a name for a new analytic workspace object. When requesting an analytic workspace object name with NEW, GET verifies that the input is not already the name of an object in any attached analytic workspace (including `EXPRESS.DB`).

VALID NAME
The VALID NAME keywords cause GET to request a name for an analytic workspace object. When requesting an analytic workspace object name with VALID, GET verifies that the input follows the rules for valid object names, even when there is no current analytic workspace and regardless of whether the name already exists.

POSLIST NAME

The POSLIST NAME keywords cause GET to request an analytic workspace object name identified by its position in the NAME dimension. When requesting an analytic workspace object name with POSLIST, GET verifies that the input is an existing position number in the NAME dimension.

datatype

Specifies the type of data being requested by GET. This can be any of the Oracle OLAP data types: INTEGER, SHORTINTEGER, DECIMAL, SHORTDECIMAL, BOOLEAN, ID, TEXT, or DATE. GET accepts a value of `NA` when requesting any data type.

VERIFY *condition-exp* [IFNOT *result-exp*]

With VERIFY, you can specify a Boolean condition that must be satisfied by the input to GET. The keyword VALUE may be used in *condition-exp* to test the input before any assignment is made. For example, when requesting a value of LSIZE, the Boolean condition might be as follows.

```
VALUE NE NA AND VALUE GE 1 AND VALUE LE 80
```

The IFNOT clause specifies a text expression to provide for occasions when the input does not satisfy *condition-exp*. For example, you might jump to an error-handling routine in your program. When you do not use IFNOT and an error occurs, GET produces an error message and then resumes waiting for input.

Notes

Current Input Stream

Oracle OLAP obtains statements for processing from the current input stream. You can override your default input stream with an INFILE command. INFILE causes Oracle OLAP to read input from a file. Each line of the infile must contain a single statement.

Input from INFILE

When the GET function is in an infile, Oracle OLAP considers the next line in the infile to be the input to GET. You must be sure you supply the expected input for GET in the line or lines following the statement that invokes the GET function.

For example, suppose your infile contains a line invoking a report program that calls GET to obtain the number of decimal places to use. The infile then continues with other statements. When you do not put the desired number of decimal places on the line following the program call, GET will examine line after line in the infile looking for the expected numeric response, rather than executing those lines as commands. See "Using GET to Obtain a Password" on page 14-6.

INTEGER Dimension Values

When GET requests a value of an INTEGER dimension, the input should usually be in the form of a dimension-value position number

Non-INTEGER Dimension Values

Non-integer dimension values must be entered in uppercase and enclosed in single quotes.

Time Dimension Values

Values of time dimensions may be entered in the format of the dimension's VNF (or in the format of the default VNF when the dimension does not have a VNF of its own) or as a date. See VNF for an explanation of how to enter values in a VNF format. See DATEORDER for an explanation the valid input styles for entering values as dates.

Whether you use the VNF format or specify the value as a date, you need to specify only the date components that are relevant for this type of time dimension. For example, for a MONTH dimension, you need to supply only the month and year.

TEXT or ID Values

TEXT and ID values provided as input to GET retain the case in which they were entered. You do not need to enclose TEXT and ID values in quotes unless they begin with single or double quotes, or contain embedded blanks or escape sequences, such as `\dnnn` or `\n`. (Remember to precede any single quote in the value with a backslash (`\'`) so Oracle OLAP will interpret it literally.)

DATE Values

When GET requests a DATE value, you can provide the input in any of the valid styles for dates, as explained in DATEORDER. Oracle OLAP uses the current value of the DATEORDER option to resolve any ambiguity in the DATE value.

Numeric Values

GET rounds a SHORTDECIMAL or DECIMAL value when converting it into an INTEGER value. When GET requests an INTEGER or SHORTINTEGER value and the input is a number beyond the range for that data type, GET produces an error message and resumes waiting for input.

GET with RAW TEXT

When GET requests RAW TEXT input and no input is provided, GET returns a null string (`''`). For any type of information other than RAW TEXT, GET waits until input is provided.

GET with POSLIST

When you use the POSLIST keyword with the GET function, Oracle OLAP requires that you enter a position value to identify the dimension value rather than the dimension name. The syntax for the POSLIST keyword depends on whether you are using the GET function with either an assignment statement created using an assignment statement or the LIMIT command. When you want to set a variable equal to the result of a GET function, use the following syntax.

expression = GET(POSLIST *dimension*)

When you want to limit a dimension to a value returned by a GET function, you specify the POSLIST keyword twice, as shown in the following syntax.

LIMIT *dimension* TO POSLIST GET(POSLIST *dimension*)

Examples

Example 14–1 Using GET to Obtain a Password

Suppose you have written an Oracle OLAP program called `myconn`. This program contains a call to GET that requests a password.

```
DEFINE myconn PROGRAM
PROGRAM
...
PASSWORD = GET(TEXT)
...
END
```

GOTO

Within an OLAP DML program, the GOTO command alters the sequence of statement execution within a program.

Syntax

GOTO *label*

Arguments

label

The name of a label elsewhere in the program constructed following the "Guidelines for Constructing a Label" on page 14-7. Execution of the program branches to the line directly following the specified label.

Note that *label*, as specified in GOTO, must *not* be followed by a colon. However, the actual label elsewhere in the program must end with a colon.

Notes

Guidelines for Constructing a Label

When you use control structures to branch to a particular location, you must provide a label for the location in order to identify it clearly. When creating a label, follow these guidelines:

- The first character in the label must be a letter, period (.), or underscore (_).
- The remaining characters in a label can be any combination of letters, numbers, periods, or underscores.
- A label must be followed immediately by a colon (:).
- Make sure that the first eight bytes in the label are unique. (Remember that, in your character set, a byte might or might not be equivalent to one character.) A label can contain up to 3999 bytes (the maximum length of a text line minus 1 byte for the colon that identifies a label). However, because only the first eight bytes of a label name are used, you can experience problems with label names greater than eight bytes when the first eight bytes are not unique.

Missing GOTO Label

When an actual label that corresponds to *label* does not exist elsewhere in the same program, execution stops with an error.

GOTO with IF and WHILE

The GOTO command can be used with IF...THEN...ELSE and WHILE to set up conditional branching, using the following syntax.

```
IF boolean-expression
  THEN GOTO label1
  ELSE GOTO label2
```

However, to preserve the clarity of your programming logic, you should minimize your use of GOTO. You can often replace GOTO with one or more statements executed conditionally using FOR, IF...THEN...ELSE, or WHILE. You can also use the SWITCH command to handle different cases within the same program.

GOTO with FOR

You can use the GOTO command in a FOR loop to branch within, or out of, the loop. This changes the sequence of statement execution, depending on where the GOTO command and the label are positioned.

- A GOTO in a FOR loop that branches to a label within the same loop makes execution continue at the label without affecting the current dimension status. Subsequent repetitions of the loop continue normally. To branch to the end of the loop, just before the DOEND command, you should consider using the CONTINUE command instead.
- A GOTO in a FOR loop that branches to a label outside the loop terminates the effect of the FOR command. Execution continues at the specified label and dimension status is restored to what it was before the loop. To branch to the statement immediately following the DOEND of a loop, you should consider using the BREAK command instead.

When you use a GOTO command outside a FOR loop to branch into the loop (that is, to a label inside the loop), an error occurs after execution passes through the rest of the loop once.

TEMPSTAT Command and GOTO Command

Within a FOR loop of a program, when a DO ... DOEND phrase follows TEMPSTAT, status is restored when the DOEND, BREAK, or GOTO is encountered.

Alternatives to the GOTO Command

While GOTO makes it easy to branch within a program, frequent use of it can obscure the logic of your program, making it difficult to follow its flow. This is particularly true when you have a complex program with several labels and GOTO commands that skip over large portions of code.

To keep the logic of your programs clear, minimize your use of GOTO.

Sometimes a GOTO command is the best programming technique, but often there are better alternatives. For example:

- Instead of using GOTO commands in an FOR command, you can often place your alternative sets of commands between DO ... DOEND commands within the IF...THEN...ELSE command itself.
- When each set of commands is long or you want to use them in more than one place in your program, then you might consider placing them in subprograms. Then, you can use the IF...THEN...ELSE command to choose between two different programs, or use the SWITCH command to choose among many different programs.

"Using the FOR Command for Looping Over Values" on page 13-71 illustrates how the FOR command loops over values. "Using DO/DOEND in a FOR Loop" on page 13-72 illustrates using DO ... DOEND within a FOR loop.

Examples

Example 14–2 Using GOTO with IF

This example shows a program that will produce a report for one of three areas, depending on what argument the user supplies when running the program. When the user specifies `EAST`, `WEST`, or `CENTRAL`, execution branches to a corresponding label, and the statements following it (statement group 1, 2, or 3) are executed.

When the user specifies anything else, execution branches to the `argerror` label, after which statements will handle the error.

```
DEFINE flexrpt PROGRAM
PROGRAM
IF NOT INLIST('East\nWest\nCentral', UPCASE(ARG(1)))
   THEN GOTO argerror

SWITCH &UPCASE(ARG(1))
DO
CASE 'EAST':
   ..." (statement group 1)
   BREAK
CASE 'WEST':
   ... "(statement group 2)
   BREAK
CASE 'CENTRAL':
   ..." (statement group 3)
   BREAK
DOEND

argerror:
   ..." statements to handle error)

END
```

GREATEST

The GREATEST function returns the largest expression in a list of expressions. All expressions after the first are implicitly converted to the data type of the first expression before the comparison.

To retrieve the smallest expression in a list of expressions, use LEAST.

Return Value

The data type of the first expression.

Syntax

GREATEST (*expr* [, *expr*]...)

Arguments

expr
An expression.

Examples

Example 14–3 Finding the Longest Text Expressions

The following statement selects the longest string.

```
SHOW GREATEST ('Harry', 'Harriot', 'Harold')
Harriot
```

Example 14–4 Finding the Largest Numerical Expression

The following statement selects the number with the greatest value.

```
SHOW GREATEST (5, 3, 18)
18
```

GROUPINGID

The GROUPINGID command populates a previously-defined object with the grouping ids for the values of a hierarchical dimension. A grouping id is a numeric value that corresponds to a level of a hierarchical dimension. The grouping id for the lowest-level of the hierarchy is `0` (zero).

Grouping ids are especially useful for identifying values of different levels of a hierarchical dimension. Dimension values in the same level of the hierarchy have the same value for their grouping id. Selecting dimension values for a specific level is easier with grouping ids because the desired values can be identified with a single condition of *groupingid* = *n*. Typically, you use the GROUPINGID command when you are planning on accessing in analytic workspace data in SQL using the `OLAP_TABLE` function.

See also: The `GROUPING_ID` function in for more information on grouping ids.

Syntax

GROUPINGID [*family-relation*] INTO *destination-object* -

{USING *level-relation*} [INHIERARCHY {*inh-variable* | *inh-valueset*}] [LEVELORDER *lo-valueset*]

where *destination-object* is one of the following:

grouping-relation
grouping-variable
grouping-surrogate

Arguments

family-relation
A self-relation for a hierarchical dimension. This self-relation is dimensioned by a hierarchical dimension. The values of the self-relation are the parents of each value in the hierarchical dimension. The *family-relation* argument is optional *only* when you use the GROUPINGID statement to populate a surrogate and the GROUPINGID statement includes a LEVELORDER clause.

grouping-relation
The name of a previously-defined relation. One of the dimensions of *grouping-relation* must be the hierarchical dimension. The values of *grouping-relation*

are calculated and populated when the GROUPINGID statement executes. See DEFINE RELATION for information on defining relations.

grouping-variable
The name of a previously-defined numeric variable. One of the dimensions of *grouping-variable* must be the hierarchical dimension. The data type of *grouping-variable* can be any numeric type including NUMBER. The values of *grouping-variable* are calculated and populated when the GROUPINGID statement executes.See DEFINE VARIABLE for information on defining variables.

grouping-surrogate
The name of a previously-defined surrogate for the hierarchical dimension. The values of *grouping-surrogate* are calculated and populated when the GROUPINGID statement executes. See DEFINE SURROGATE for information on defining surrogates.

USING
Specifies that the level of the values of the hierarchical dimension are to be considered when creating grouping ids.

level-relation
A relation that is dimensioned by the hierarchical dimension. For each value of the hierarchical dimension, the relation has its value the name of the level for the dimension's value.

INHIERARCHY
Specifies that only some of the values of the hierarchical dimension are to be considered when creating grouping ids.

inh-variable
A BOOLEAN variable that is dimensioned by the hierarchical dimension and, when the hierarchical dimension is a multi-hierarchical dimension, by a dimension that is the names of the hierarchies. The values of the variable are `TRUE` when the dimension value is in a hierarchy and `FALSE` when it is not.

inh-valueset
The name of a valueset object whose values identify the hierarchical dimension values to be considered when creating grouping ids. Values not included in the valueset are ignored.

LEVELORDER
Specifies the top-down order of the levels when creating grouping ids.

lo-valueset
The name of a valueset object whose values are the names of the levels to be used when creating grouping ids. The order of the values in the valueset object determine the grouping id assigned.

Notes

GROUPINGID with the OLAP_TABLE Function

Typically, you use the GROUPINGID command when you are planning on accessing analytic workspace data in SQL using the `OLAP_TABLE` function. For more information on the `OLAP_TABLE` function see the *Oracle OLAP Reference*.

Examples

Example 14–5 Using GROUPINGID to Populate a Variable with Grouping Ids

Assume that you have the following objects in your analytic workspace.

```
DEFINE geography DIMENSION TEXT WIDTH 12
LD Geography Dimension Values
DEFINE geography.parent RELATION geography <geography>
LD Child-parent relation for geography
DEFINE geography.hierarchyid DIMENSION INTEGER
LD Dimension whose values are ids for hierarchies in geography
```

To create a grouping id variable for the `Standard` hierarchy of geography, define a child-parent relation of only those values that are in the hierarchy whose grouping ids you want to generate, and define a variable to hold the grouping ids. Examples of these definitions follow.

```
DEFINE geog.gid INTEGER VARIABLE <geography>
DEFINE geography.newparent RELATION geography <geography>
```

Then populate these variables using statements similar to these.

```
AW DETACH myaw
AW ATTACH myaw ro
PUSH OKNULLSTATUS
OKNULLSTATUS = TRUE
" Populate the child-parent relation for hierarchy 1
geography.newparent = geography.parent(geography.hierarchyid 1)
" Populate the grouping id variables
GROUPINGID geography.newparent INTO geog.gid
" Save changes to analytic workspace
POP OKNULLSTATUS
ALLSTAT
UPDATE
COMMIT
```

Reports for the new objects created by this code (geography.newparent and GEOG.GID) follow.

```
REPORT geography.newparent

GEOGRAPHY          GEOGRAPHY.NEWPARENT
---------------- ----------------
World            NA
Americas         World
Canada           Americas
Toronto          Canada
Montreal         Canada
Ottawa           Canada
Vancouver        Canada
Edmonton         Canada
Calgary          Canada
Usa              Americas
Boston           Usa
Losangeles       Usa
Dallas           Usa
Denver           Usa
Newyork          Usa
Chicago          Usa
Seattle          Usa
Mexico           Americas
...              ...
Japan            Asia
Tokyo            Japan
Osaka            Japan
Kyoto            Japan
China            Asia
Beijing          China
Shanghai         China
...              ...
India            Asia
Ireland          Europe
Taiwan           Asia
Thailand         Asia

REPORT geog.gid
GEOGRAPHY            GEOG.GID
---------------- ----------------
World                           7
Americas                        3
Canada                          1
```

```
Toronto                         0
Montreal                        0
Ottawa                          0
Vancouver                       0
Edmonton                        0
Calgary                         0
Usa                             1
Boston                          0
Losangeles                      0
Dallas                          0
Denver                          0
Newyork                         0
Chicago                         0
Seattle                         0
Mexico                          1
...                             ...
Japan                           1
Tokyo                           0
Osaka                           0
Kyoto                           0
China                           1
Beijing                         0
Shanghai                        0
...                             ...
India                           1
Ireland                         1
Taiwan                          1
Thailand                        1
```

GROWRATE

The GROWRATE function calculates the growth rate of a time-series expression, based on the first and last values of the series.

Return Value

DECIMAL

Syntax

GROWRATE(*expression* [*time-dimension*])

Arguments

expression
A numeric expression for which you want to calculate the growth rate. The expression must be dimensioned by a time dimension.

time-dimension
The name of the time dimension by which *expression* is dimensioned. When the time dimension has a type of DAY, WEEK, MONTH, QUARTER, or YEAR, the *time-dimension* argument is optional, unless *loans* has more than one time dimension.

Notes

Dimensions of the Result

The result returned by GROWRATE is dimensioned by all the dimensions of *expression* except the time dimension.

Understanding the Calculation

GROWRATE bases its calculation on the values of *expression* that correspond to the first and last values in the status of the time dimension. The intervening values of *expression* are ignored. GROWRATE uses the following calculation.

$$\text{GROWRATE} = (\text{last}/\text{first})^{1/(n-1)} - 1$$

In the exponent, *n* is the number of values in the status of the time dimension.

The Expression Argument

The following rules apply to the first and last values of *expression*:

- The first value of *expression* cannot be zero. (This is to avoid a division by zero in the GROWRATE calculation.)
- The first and last values of *expression* must both be positive or both negative. (Or the last value of *expression* can be zero, regardless of whether the first value is positive or negative.)
- Neither the first value nor the last value of *expression* can be NA.

Examples

Example 14–6 Determining Growth Rate

The following statements limit the dimensions of the actual variable and produce a report.

```
LIMIT month TO 'Dec95' TO 'Mar96'
LIMIT line TO 'net.income'
REPORT DOWN division ACROSS month: actual
```

These statements produce the following report.

```
LINE: NET.INCOME
               ------------------ACTUAL-------------------
               -------------------MONTH-------------------
DIVISION         Dec95      Jan96      Feb96      Mar96
-------------- ---------- ---------- ---------- ----------
Camping          4,378.09  19,915.13  22,510.38  34,731.63
Sporting         6,297.02  13,180.29  17,429.17  18,819.14
Clothing        87,471.74 107,257.85 133,566.01 127,132.55
```

The statement `REPORT W 20 GROWRATE(actual)` produces a report that shows the growth rate of the actual net income in the `demo` workspace between December 1995 and March 1996.

```
               --GROWRATE(ACTUAL)--
               --------LINE--------
DIVISION            NET.INCOME
-------------- --------------------
Camping                        0.99
Sporting                       0.44
Clothing                       0.13
```

HEADING

The HEADING command produces titles and column headings for a report. The heading output is sent to the current outfile. The form of the HEADING command is the same as that of the ROW command. When you use HEADING, however, Oracle OLAP does not add any numeric values from the heading to column subtotals or grand totals.

Frequently, HEADING commands are used in a PAGEPRG program to produce titles or column headings on each page of a report.

Syntax

HEADING [*attribs*] {*expression1*|SKIP}, [*attribs*] {*expressionN*|SKIP}

Arguments

attribs
The attributes that specify the format for each column. (See ROW command for a list and detailed explanation of the available attributes.)

expression
The text to be used as a column heading. To use literal text for a column heading, enclose the text in single quotes. (See ROW command for more information on using expressions, attributes, and ACROSS groups to produce columns.)

SKIP
Used in place of an expression to indicate that the column is to be left blank.

Notes

Creating Titles

To create a title or subtitle in a report, use HEADING to produce a single "column" with a width equal to the setting of the LSIZE option. You can then center your text within this "column" to produce a centered title.

Maximum Heading Width

The maximum width of any line in a report, including a heading line, is 4000 characters.

Improving Report Performance

When you know ahead of time that you will not need the subtitling capability of the ROW command, using the HEADING command instead of ROW to produce the lines of your report can provide a time savings, since Oracle OLAP will not be keeping track of subtotals.

ROW Command Notes

The notes for the ROW command also apply to the HEADING command (with the exception of the note on row and column arithmetic in ROW).

Examples

Example 14–7   Producing Column Headings

In a report, you want to have headings for your columns. You can use a HEADING command such as the following in your program.

```
HEADING UNDER '-' CENTER <WIDTH 15 'Product' -
   ACROSS district FIRST 3: district>
```

This command produces the following result.

```
    Product      Atlanta     Boston    Chicago
--------------- ---------- ---------- ----------
```

HIDE

The HIDE command hides the text of a program, so that you cannot display it using the DESCRIBE command, the EDIT command, or the OBJ function. You can perform all other actions on the program, including executing, compiling, renaming, or exporting.

When you hide a program, you supply a seed expression, which Oracle OLAP uses to encrypt the program text. You can use this seed expression later with the UNHIDE command to make the text visible.

Syntax

HIDE *prog-name seed-exp*

Arguments

prog-name
The name of the program whose text you want to hide. Do not enclose the program name in quotes.

seed-exp
A single-line text expression to be used as a seed value in the encryption of the program text. Do not specify `NA` for this value.

Keep a record of this seed expression, so that you can use it later with the UNHIDE command. The seed expression you specify in the UNHIDE command must be byte-for-byte the same value as you used in this command. Also, the seed expression is case-sensitive, so record uppercase and lowercase characters carefully.

Notes

Exporting and Importing with the Seed

When you export and import a hidden program, the text remains hidden in the analytic workspace in which it is imported. It retains the same seed expression for use with the UNHIDE command.

Forgetting the Seed Expression

When you want to use the UNHIDE command on a program but you have forgotten the seed expression, you can call Oracle OLAP Products Technical

Support for help in solving your problem. Before calling, make a connection to Oracle OLAP from OLAP Worksheet and attach the analytic workspace that contains the hidden program.

Examples

Example 14–8 Hiding Program Text

The following example hides the text of a program called `sales_rpt`.

```
HIDE sales_rpt 'Crystal'
```

HIERCHECK

The HIERCHECK program checks the parent relation of a hierarchical dimension to make sure it has no loops. A hierarchical dimension's parent relation specifies the parent for each of the dimension's values. A loop will occur when a dimension value has inadvertently been specified as its own ancestor or descendant in the parent relation. When you execute a ROLLUP command or a AGGREGATE command that uses a parent relation with a loop, an error message will be returned when the loop is identified.

You can call HIERCHECK as a command or as a Boolean function. When called as a function, HIERCHECK returns `YES` when the parent relation "passes" the check (for example, it contains no loops), and `NO` when it fails the check (it does contain loops).

Return Value

BOOLEAN

Syntax

When Used as a Command

HIERCHECK *relation-name* [NOSTATUS]

When Used as a Function or with CALL

HIERCHECK ('*relation-name*' [NOSTATUS])

Arguments

relation-name
A text expression indicating the name of the parent relation to be checked.

You can use OLAP DML statements to create a parent relation. To do so, you define a relation that relates a dimension to itself, and then you can specify the parent of each dimension value in the relation. This makes the dimension hierarchical.

NOSTATUS
Specifies that the current status of any extra dimensions on a parent relation is ignored, so that all the hierarchies of a multi-dimensional parent relation will be checked for infinite loops.

When a parent relation has been defined with one or more extra dimensions (that is, with dimensions other than the required embedded total dimension), you can create and name more than one hierarchy within the parent relation. Each of the values of the extra dimension(s) can represent a different hierarchy. The hierarchies use the same dimension values, but the way in which those dimension values relate to each other is different in each hierarchy within the relation.

You can use the LIMIT command on the extra dimension(s) of a parent relation to select which of the parent relation's multiple hierarchies are in status. When a parent relation has multiple hierarchies and the current status of the extra dimension(s) on the parent relation does not include all of those hierarchies, NOSTATUS ignores the current status and checks every hierarchy of the parent relation for loops.

Notes

Why You Should Use HIERCHECK

It is a good strategy to use HIERCHECK at the time you build your hierarchies as a way to verify that they are valid. In other words, you should not attempt to roll up a variable's data unless you have already verified that its dimensions' hierarchies are structured correctly.

Why ROLLUP and AGGREGATE Use HIERCHECK

The ROLLUP command and the AGGREGATE command both use HIERCHECK in order to prevent infinite looping once the command has been executed.

When to Use HIERCHECK

You should check a parent relation for loops after you set up the levels of a hierarchical dimension, before you load data into any variable that is dimensioned by the hierarchical dimension, or before you use the ROLLUP or AGGREGATE command for the first time with a variable. Although it is possible to roll up a variable without first having checked the parent relations of all of its hierarchical dimensions with HIERCHECK, you should make it a practice to use HIERCHECK first.

For example, suppose you accidentally create a hierarchy that is invalid. You then fail to use HIERCHECK to check that hierarchy. Now, suppose you submit a rollup program as a batch job to run overnight. When you check on the batch job the next morning, you will see that the job failed to run because when the ROLLUP command was called, HIERCHECK detected a loop, which prevented the ROLLUP command from running to completion. In this case, you will have lost a night's work because you did not use HIERCHECK at the time when you created your hierarchies. In other words, using HIERCHECK yourself (instead of waiting for the ROLLUP or the AGGREGATE command to do it for you) will save you time and effort.

Using HIERCHECK as a Function

You may use HIERCHECK as a function. When the parent relation has no loops, the return value is `YES`. When HIERCHECK detects a loop, the return value is `NO`. When you call it as a function, HIERCHECK does not signal an error when it finds a loop.

Using HIERCHECK as a Command

When you use HIERCHECK as a command or with the CALL command, it signals an error when it finds a loop in the parent relation. The error message identifies the dimension values that are involved in the loop, the name of the hierarchy (referred to as the "extra dimension values") in which the loop occurs (when the parent relation has one or more named hierarchies), and the name of the parent relation in which the loop was found. When a parent relation has no loops, no message is displayed. See Example 14–9, "Checking for Loops" on page 14-27.

Checking the Result

When you call HIERCHECK as a function, you get the result as a Boolean return value. When you call HIERCHECK as a command or with the CALL command, you can check a Boolean variable called HIERCHK.LOOPFND to determine the result. When a loop is found, the value is `YES`. When HIERCHECK did not terminate normally (for example, because a bad argument was passed in), the value is `NA`. When HIERCHECK runs successfully and no loops are found, the value is `NO`.

The Problem Dimension Values

When HIERCHECK finds a loop in a parent relation, the names of all dimension values that are involved in that loop are stored in a variable named `HIERCHK.LOOPVALS`. You can check the value of this variable and use this information to determine where the looping problem lies.

The Problem Hierarchy

When HIERCHECK finds a loop and your parent relation has more than one hierarchy, the name of the hierarchy in which a loop is found is stored in a variable called `HIERCHK.XTRADIMS`. You can check the value of this variable to find out which hierarchy you should check for the looping problem.

Multiple Loops

HIERCHECK detects the presence of loops, but it does not report multiple loops. While the name of every dimension value involved in a loop will be stored in `HIERCHK.LOOPVALS`, that does not mean that those dimension values are all part of the same loop; they may be involved in separate loops. Once you have detected and fixed a looping problem, it is important to use HIERCHECK again to check the parent relation until it is loop-free.

Examples

Example 14–9 Checking for Loops

This example shows how to create a parent relation and check it for loops. You would begin by defining a dimension and adding values to it.

```
DEFINE geography DIMENSION ID
MAINTAIN geography ADD 'U.S.'
MAINTAIN geography ADD 'East' 'Central' 'West'
MAINTAIN geography ADD 'Boston' 'Atlanta' 'Chicago' 'Dallas' 'Denver' 'Seattle'
```

Next, relate the dimension to itself. The following statement defines a parent relation called GEOG.GEOG, which relates the GEOGRAPHY dimension to itself.

```
define geog.geog RELATION geography <geography>
```

You would then specify the hierarchy of the dimension values. In this example, there will be three levels in the hierarchy: country, regions, and cities. When you specify the hierarchy, you assign parent dimension values (such as `East`) to child dimension values (such as `Boston`) for every level except the highest level. To do this, you store values in the relation. First, group the children together with a LIMIT command, then assign a parent to those children.

```
LIMIT geography TO 'East' 'Central' 'West'
geog.geog = 'U.S.'
LIMIT geography TO 'Boston' 'Atlanta'
geog.geog = 'East'
LIMIT geography TO 'Chicago' 'Dallas'
geog.geog = 'Central'
LIMIT geography TO 'Denver' 'Seattle'
geog.geog = 'West'
```

Now you can check for loops in the parent relation `geog.geog`, as shown by the following statement.

```
HIERCHECK geog.geog
```

In this case, HIERCHECK produces no message output, which means there are no loops in `geog.geog`. It sets HIERCHK.LOOPFND to `NO`, and leaves HIERCHK.LOOPVALS and HIERCHK.XTRADIMS set to `NA`.

Now suppose the following mistake had been made in the storing of values in the relation.

```
LIMIT geography TO 'East' 'Central' 'West'
geog.geog = 'East'
```

The preceding statements inadvertently make `East` its own parent, which would cause a ROLLUP command to loop infinitely. When you now check the `geog.geog` relation for loops, the following statement produces the following error message.

```
HIERCHECK geog.geog
ERROR: HIERCHECK has detected one or more loops in the hierarchy represented by
GEOG.GEOG. The values involved are 'East'.
```

HIERHEIGHT command

The HIERHEIGHT command populates a previously-defined relation with the values of a specified hierarchical dimension by level. Typically, you use the HIERHEIGHT command when you are preparing an analytic workspace for access using the `OLAP_TABLE` function.

To retrieve the value of a node (by level) for the value of a hierarchical dimension, use the HIERHEIGHT function.

Syntax

HIERHEIGHT *familyrelation* [(*qdrlist*)] INTO{*hierheight-relation* -

[USING *level-relation*[A | D]] [INHIERARCHY { *inh-variable*| *inh-valueset*}]

Arguments

family-relation
A child-parent self-relation for the hierarchical dimension. This relation can have multiple dimensions; however, one of the dimensions of *family-relation* must be the hierarchical dimension. The values of the *family-relation* are the values of the hierarchical dimension that is the parent of each set of dimension values

qdrlist
A list of QDRs that limits the values of *family-relation*. Specify the QDRs as described in "Form of a Qualified Data Reference" on page 3-33. When you do not specify a value for *qdrlist*, HIERHEIGHT uses the values of *family-relation* that are in current status.

hierheight-relation
A previously -defined relation that the HIERHEIGHT command populates when it executes. This relation can have multiple dimensions; however, it must be dimensioned by the dimensions of *family-relation* and one other dimension that represents the levels of the hierarchical dimension. The actual constuct of the dimension that represents the levels of the hierarchical dimension varies depending on whether or not the HIERHEIGHT statement includes the USING phrase:

- When the HIERHEIGHT statement includes the USING phrase, the dimension that represents the levels of the hierarchical dimension is a dimension that contains the names of the levels.

- When the HIERHEIGHT statement does not include the USING phrase, the dimension that represents the levels of the hierarchical dimension is an INTEGER dimension that has as values the depth of the level.

When *hierheight-relation* is populated before the HIERHEIGHT command executes, the command depopulates it before computing new values.

level-relation
A relation that is a dimensioned by the hierarchical dimension and (when the hierarchical dimension is a multi-hierarchical dimension) by a dimension that is the names of the hierarchies. The values of the relation are values of a dimension that represents the levels of the hierarchy. This dimension typically is a TEXT or ID dimension that has the names of the levels as values.

A
Ascending order.

D
Descending order. (Default)

inh-variable
A BOOLEAN variable that is dimensioned by the hierarchical dimension and, when the hierarchical dimension is a multi-hierarchical dimension, by a dimension that is the names of the hierarchies. The values of the variable are `TRUE` when the dimension value is in a hierarchy and `FALSE` when it is not.

inh-valueset
The name of a valueset object whose values are the hierarchical dimension values to be considered when creating grouping ids. Values not included in the valueset are ignored.

Notes

HIERHEIGHT with the OLAP_TABLE Function

Typically, you use the HIERHEIGHT command when you are preparing an analytic workspace for access using the `OLAP_TABLE` function.

Examples

Example 14–10 Creating a Relational Representation of a Geography Hierarchy

Assume that there is an analytic workspace named `myaw` that has a Geography hierarchy defined with analytic objects with the following definitions.

```
DEFINE geog.hierdim DIMENSION TEXT
LD Hierarchy names for Geography hierarchies

DEFINE geog.leveldim DIMENSION TEXT
LD List of levels for GEOGRAPHY hierarchies

DEFINE geography DIMENSION TEXT WIDTH 12
LD Values for the Geography hierarchies

DEFINE geog.levelrel RELATION geog.leveldim <geography geog.hierdim>
LD Level of each value in the Geography hierarchies

DEFINE geog.parent RELATION geography <geography geog.hierdim>
LD Child-parent relation for the Geography hierarchies

DEFINE geog.familyrel RELATION geography <geography geog.leveldim geog.hierdim>
LD Geography values by level and hierarchy
```

These objects have the following structures.

```
GEOGRAPHY
------------------
World
Americas
Canada
USA
Toronto
Montreal
Boston
LosAngeles

GEOG.HIERDIM
------------------
Standard
Consolidated
```

```
GEOG.LEVELDIM
------------------
World
Continent
Country
City
Consolidated
Continent
Consolidated
Country

                   ------------GEOG.LEVELREL------------
                   ------------GEOG.HIERDIM-------------
GEOGRAPHY               Standard         Consolidated
------------------ ------------------ ------------------
World              World              NA
Americas           Continent          Consolidated
                                      Continent
Canada             Country            Consolidated
                                      Country
USA                Country            Consolidated
                                      Country
Toronto            City               NA
Montreal           City               NA
Boston             City               NA
LosAngeles         City               NA

                   -------------GEOG.PARENT-------------
                   ------------GEOG.HIERDIM-------------
GEOGRAPHY               Standard         Consolidated
------------------ ------------------ ------------------
World              NA                 NA
Americas           World              NA
Canada             Americas           Americas
USA                Americas           Americas
Toronto            Canada             NA
Montreal           Canada             NA
Boston             USA                NA
LosAngeles         USA                NA
```

To create a family relation of the Geography hierarchy you define an analytic workspace object with the following definition.

```
DEFINE geog.familyrel RELATION geography <geography geog.leveldim geog.hierdim>
LD Geography values by level and hierarchy
```

Then you use the HIERHEIGHT command as illustrated in the following statement to populate the object.

```
HIERHEIGHT geog.parent INTO geog.familyrel USING geog.levelrel
```

By issuing the REPORT command, you can display the relational representations of both the `Standard` and `Consolidated` hierarchies of the `geography` dimension.

```
REPORT DOWN geography geog.familyrel

GEOG.HIERDIM: Standard
             -------------------------------GEOG.FAMILYREL--------------------------------
             --------------------------------GEOG.LEVELDIM--------------------------------
                                                                 Consolidated Consolidated
GEOGRAPHY       World      Continent     Country        City      Continent     Country
------------ ------------ ------------ ------------ ------------ ------------ ------------
World        World        NA           NA           NA           NA           NA
Americas     World        Americas     NA           NA           NA           NA
Canada       World        Americas     Canada       NA           NA           NA
USA          World        Americas     USA          NA           NA           NA
Toronto      World        Americas     Canada       Toronto      NA           NA
Montreal     World        Americas     Canada       Montreal     NA           NA
Boston       World        Americas     USA          Boston       NA           NA
LosAngeles   World        Americas     USA          LosAngeles   NA           NA

GEOG.HIERDIM: Consolidated
             -------------------------------GEOG.FAMILYREL--------------------------------
             --------------------------------GEOG.LEVELDIM--------------------------------
                                                                 Consolidated Consolidated
GEOGRAPHY       World      Continent     Country        City      Continent     Country
------------ ------------ ------------ ------------ ------------ ------------ ------------
World        NA           NA           NA           NA           NA           NA
Americas     NA           NA           NA           NA           Americas     NA
Canada       NA           NA           NA           NA           Americas     Canada
USA          NA           NA           NA           NA           Americas     USA
Toronto      NA           NA           NA           NA           NA           NA
Montreal     NA           NA           NA           NA           NA           NA
Boston       NA           NA           NA           NA           NA           NA
LosAngeles   NA           NA           NA           NA           NA           NA
```

HIERHEIGHT function

The HIERHEIGHT function returns the value of a node at a specified level for the first value in the current status list of a hierarchical dimension.

To populate a previously-defined relation with the values of a specified hierarchical dimension by level, use the HIERHEIGHT command.

Syntax

HIERHEIGHT(*family-relation* [,] *level*)

Return Value

The data type returned by HIERHEIGHT is the data type of the dimension value of *family-relation*.

Arguments

family-relation
A child-parent self-relation for the hierarchical dimension. The values of *family-relation* are the parents.

level
An `INTEGER` value that represents a level of the hierarchical dimension. The value `1` (one) represents the lowest-level of the hierarchical dimension.

Notes

Limiting the Hierarchical Dimension
The HIERHEIGHT function always returns a single value of the hierarchical dimension. When you do not limit the hierarchical dimension to a single value before calling the HIERHEIGHT function, the HIERHEIGHT function executes against the first value in the current status list of the dimension. Typically, you either limit the hierarchical dimension to a single value before you call the HIERHEIGHT function or you use the HIERHEIGHT function after FOR command in order to execute the HIERHEIGHT function for each value of the hierarchical dimension.

Examples

Example 14–11 Using HIERHEIGHT as a Simple Command

Assume that your analytic workspace has a hierarchical dimension named `geography` and a relation named `g0.stanparent` that is a self-relation of the `geography` values for the `Standard` hierarchy of `geography`.

```
DEFINE g0.newparent RELATION geography <geography>
LD Parent-child when hierarchy of geography is 1
```

Issuing a report command like `REPORT g0.stanparent` displays the values in `g0.stanparent`.

```
GEOGRAPHY          G0.STANPARENT
---------------- ----------------
World            NA
Americas         World
Canada           Americas
Toronto          Canada
Montreal         Canada
Ottawa           Canada
...              ...
USA              Americas
Boston           USA
LosAngeles       USA
...              ...
Mexico           Americas
Mexicocity       Mexico
Argentina        Americas
BuenosAires      Argentina
Brazil           Americas
Saopaulo         Brazil
Colombia         Americas
Bogota           Colombia
Australia        World
East.Aust        Australia
Sydney           East.Aust
Madrid           Spain
Budapest         Hungary
Athens           Greece
Vienna           Austria
Melbourne        East.Aust
Central.aust     Australia
Tai-pei          Taiwan
Singapore        Asia
Adelaide         Central.Aust
Bangkok          Thailand
West.aust        Australia
Newdelhi         India
Perth            West.Aust
Bombay           India
Malaysia         Asia
Europe           World
France           Europe
```

```
Caen             France
Paris            France
```

Now you limit GEOGRAPHY to the value `Americas` by issuing the following OLAP DML statement.

```
LIMIT geography TO 'Americas'
```

When you use the HIERHEIGHT function to find the node for `Americas` for the lowest-level of the hierarchy (level 1) by issuing the following OLAP DML statement.

```
REPORT HIERHEIGHT(g0.stanparent 1)
```

The following report is produced.

```
HIERHEIGHT(G0.STANPARENT
COUNTER)
------------------------------
NA
```

When you use the HIERHEIGHT function to find the node for `Americas` for the highest-level of the hierarchy (level 4) by issuing the following OLAP DML statement.

```
REPORT HIERHEIGHT(g0.stanparent 4)
```

The following report is produced.

```
HIERHEIGHT(G0.STANPARENT
COUNTER)
------------------------------
World
```

When you use the HIERHEIGHT function to find the node for `Americas` for the levels `2` and `3` of the hierarchy by issuing the following OLAP DML statements.

```
REPORT HIERHEIGHT(g0.stanparent 2)
REPORT HIERHEIGHT(g0.stanparent 3)
```

The following reports are produced.

```
HIERHEIGHT(G0.STANPARENT
COUNTER)
------------------------------
NA

HIERHEIGHT(G0.STANPARENT
COUNTER)
------------------------------
Americas
```

Notice that the output for each level corresponds in between the values that are created for a relation created using HIERHEIGHT command. For example, assume you created a relation named `geog.stanhierrel` for the standard hierarchy for `geography` and limit `geography` to '`Americas`. A report of `geog.stanhierrel` would show the same `geography` values for each level.

```
LIMIT geography TO 'AMERICAS'
REPORT DOWN geography geog.stanhierrel

                 ---------------------------GEOG.STANHIERREL--------------------
                 ----------------------------GEOG.LVLDIM------------------------
GEOGRAPHY               1                2                3                4
---------------- ---------------- ---------------- ---------------- ------------
Americas         NA               NA               Americas         World
```

Example 14–12 Using HIERHEIGHT After a FOR Command

Assume that your analytic workspace has a program named `findnodes` that finds the nodes of all of the `geography` values in status.

```
DEFINE FINDNODES PROGRAM
PROGRAM
VARIABLE level INTEGER
FOR geography
DO
counter = 1
WHILE counter LE statlen(geog.lvldim)
DO
REPORT HIERHEIGHT(g0.stanparent level)
level = level + 1
DOEND
DOEND
END
```

Assume also that you limit `geography` to `Americas` and `Asia` and call the HIERHEIGHT function for each level of the `Standard` hierarchy by issuing the following OLAP statements.

```
LIMIT geography TO 'Americas', 'Asia'
CALL findnodes
```

The output of the `findnodes` program for the `geography` values `Americas` and `Asia` is follows. The program first reports on the value of each level for `Americas` is provided. Then it reports on the value of each level for `Asia`.

```
HIERHEIGHT(G0.STANPARENT
COUNTER)
------------------------------
NA

HIERHEIGHT(G0.STANPARENT
COUNTER)
------------------------------
NA

HIERHEIGHT(G0.STANPARENT
COUNTER)
------------------------------
Americas

HIERHEIGHT(G0.STANPARENT
COUNTER)
------------------------------
World

HIERHEIGHT(G0.STANPARENT
COUNTER)
------------------------------
NA

HIERHEIGHT(G0.STANPARENT
COUNTER)
------------------------------
NA

HIERHEIGHT(G0.STANPARENT
COUNTER)
------------------------------
Asia

HIERHEIGHT(G0.STANPARENT
COUNTER)
------------------------------
World
```

Notice that the output for each level corresponds in between the values that are created for a relation created using the HIERHEIGHT command

```
LIMIT geography TO 'Americas' 'Asia'
REPORT DOWN geography geog.stanhierrel

                 ----------------------------GEOG.STANHIERREL--------------------
                 ----------------------------GEOG.LVLDIM-------------------------
GEOGRAPHY               1                2                3                4
---------------- ---------------- ---------------- ---------------- ------------
Americas         NA               NA               Americas         World
Asia             NA               NA               Asia             World
```

IF...THEN...ELSE

The IF...THEN...ELSE command executes one or more commands in a program when a specified condition is met. Optionally, it also executes an alternative command or group of commands when the condition is not met. You can use IF only within programs.

You can also use IF as a conditional operator in an expression. See "IF as a Conditional Operator" on page 14-43.

Syntax

IF *boolean-expression*

  THEN *statement1*

  [ELSE *statement2*]

Arguments

boolean-expression
Any valid Boolean expression that returns either `TRUE` or `FALSE`.

THEN *statement1*
Oracle OLAP executes the *statement1* argument when the Boolean expression is `TRUE`. The *statement1* must be on the same line as THEN.

ELSE *statement2*
Oracle OLAP executes the *statement2* argument when the Boolean expression is `FALSE`. The *statement2* must be on the same line as ELSE. When you omit the ELSE phrase, execution continues with the statement after the whole IF...THEN... command in the program.

Notes

IF with DO

You can use the IF command for conditional execution of two or more statements by following the THEN or ELSE (or both) keywords with a DO ... DOEND sequence. See Example 14–13, "Using IF...THEN...ELSE" on page 14-43.

IF as a Conditional Operator

When you use IF as a conditional operator in an expression, it has the following format. ELSE is required when IF is used as an operator.

```
IF boolean-exp THEN exp1 ELSE exp2
```

In most cases, *exp1* and *exp2* must be of the same basic data type (numeric, text, or Boolean). The value of the whole expression is the value of either *exp1* or *exp2*.

However, when the data type of either *exp1* or *exp2* is DATE, it is possible for the other expression to have a numeric or text data type. Because Oracle OLAP expects both data types to be DATE, it will convert the numeric or text value to a DATE.

Single or Multiple Lines

When IF is used as an expression, the THEN and ELSE keywords must be on the same line as IF. When used as a command, THEN and ELSE must be on separate lines.

Examples

Example 14–13 Using IF...THEN...ELSE

The following lines from a program illustrate the use of IF...THEN...ELSE.... When the Boolean expression `ANY(DOLLARS LT 200000)` is `TRUE`, the statements following THEN (`statement group 1`) are executed. When the expression is `FALSE`, the statements following ELSE (`statement group 2`) are executed instead.

```
IF ANY(DOLLARS LT 200000)
THEN DO
  ... " (statement group 1)
  DOEND
ELSE DO
  ... "(statement group 2)
   DOEND
```

Example 14–14 Using IF as a Conditional Operator

In a program that produces a report, you would like to report a previous year's actual expenses or the current year's budget, depending on the year passed to the program as an argument. A conditional expression in a JOINCHARS function produces a heading with the word Actual or Budget. Another conditional expression selects the variable to report. The program would include the following lines.

```
ARGUMENT cur.year year

LIMIT month TO year cur.year
REPORT -
   HEADING JOINCHARS( 'Expenses: ' -
                        IF cur.year LT 'Yr95' -
                        THEN 'Actual FOR ' -
                        ELSE 'Budget FOR ', -
                       cur.year ) -
   IF cur.year LT 'Yr95' THEN actual ELSE budget
```

IMPORT

The IMPORT command transfers data to an analytic workspace from a text file, a spreadsheet, or another analytic workspace from an EIF file.

Because the syntax of the IMPORT command is different depending on where the data to be imported is located, separate topics are provided for different types of source files:

- IMPORT (from EIF)
- IMPORT (from text)
- IMPORT (from spreadsheet)

IMPORT (from EIF)

You can use the IMPORT (from EIF) command to copy data and definitions into your Oracle OLAP analytic workspace from an EIF file. IMPORT also copies any dimensions of the imported data that do not already exist in your workspace, even when you do not specify them in the command. For information on importing variables dimensioned by composites, see "Unnamed Composites" on page 14-54.

IMPORT (from EIF) is commonly used in conjunction with EXPORT (to EIF) to copy parts of one Oracle OLAP analytic workspace to another; you export objects from the source workspace to an EIF file and then import the objects from the EIF file into the target workspace. The source and target workspaces can reside on the same platform or on different platforms. When you transfer an EIF file between computers, you use a binary transfer to overcome file-format incompatibilities between platforms. The EIF file must have been created with the EIFVERSION set to a version that is less than or equal to the version number of the target workspace. Use EVERSION to verify the target version number.

You can also use IMPORT to store information in the EIFNAMES and EIFTYPES options.

Syntax

IMPORT *import_item* FROM EIF FILE *file-id* [INTO *workspace*] -

  [MATCH [STATUS]|<u>APPEND</u>|REPLACE [DELETE]] [LIST [ONLY]] [<u>DATA</u>] -

  [DFNS] [UPDATE] [NOPROP] [NASKIP] [NLS_CHARSET *charset-exp*]

where:

import_item is one of the following:

name [AS *newname*]

ALL

Arguments

***name* [AS *newname*]**

The name of an analytic workspace object to be imported from an EIF file to an attached workspace. You cannot specify a qualified object name for the object, because the object is not yet in any workspace. You can list more than one name at a time. See the INTO *workspace* argument for information about where the object will be imported.

AS *newname* can be used to rename any type of object being imported except dimensions.

When you have exported a multidimensional object as separate variables, list all the variable names. (See the SCATTER AS keyword in the EXPORT (to EIF).)

ALL
Indicates that you want to import all the objects contained in the EIF file. (Default)

See the INTO *workspace* argument for information about where the objects will be imported.

INTO *workspace*
A workspace name that identifies the attached workspace into which objects will be imported. When the objects exist in the specified workspace, then their data will be overwritten by the imported data. When the objects do not already exist, IMPORT creates them it in the specified workspace. IMPORT ignores identically named objects when they exist in other attached workspaces.

When you do not specify this argument, then Oracle OLAP does the following:

- When you have not previously defined the objects being imported in an attached workspace, then IMPORT defines them automatically in the current workspace.
- When the objects already exist in any attached workspace, then IMPORT overwrites the data they contain with the imported data.

FROM EIF FILE *file-id*
Identifies the file you want to import. *File-id* is a text expression that represents the name of the file. The name must be in a standard format for a file identifier.

MATCH [STATUS]
Indicates that the IMPORT command should bring in only the data associated with dimension values that match those already in the target workspace. For dimensions other than time dimensions, be sure that corresponding dimension values are spelled and capitalized identically in the EIF file and your target workspace when you want them to match; for example, `Tents` does not match `TENTS`. For time dimensions, Oracle OLAP identifies dimension values by the dates they represent rather than by the way they are displayed. Therefore, time dimension values in the EIF file will automatically match time dimension values in your workspace when they represent the same time periods. When you specify MATCH STATUS, IMPORT

only imports data associated with the values included in the current status of that dimension. When the dimension is limited in the target workspace, Oracle OLAP ignores any data in the EIF file associated with the values excluded from the status.

APPEND
Indicates that the IMPORT command should bring in all the dimension values, along with associated data, regardless of whether or not the dimension values match those already present in the target workspace. APPEND adds those that do not match to those already present; it adds new values to the end of the list of dimension values. For time dimensions, APPEND also adds dimension values to fill in any gaps between the dimension values in your target workspace and the new ones. (Default)

REPLACE [DELETE]
Indicates that, for objects already defined in the workspace, IMPORT should keep the existing dimension values that match the dimension values in the EIF file. IMPORT deletes dimension values (and their data) that do *not* match dimension values in the EIF file. IMPORT replaces the associated data for the dimension values kept as part of the new dimension when the associated data variables are included in the EIF file. For text dimensions, the *order* of the dimension values in the EIF file is also adapted.

When you specify REPLACE DELETE, no matching takes place. Before importing a dimension, Oracle OLAP performs a MAINTAIN DELETE ALL, which discards all data associated with the existing dimension, as well as the dimension values.

Important: Be careful when using the REPLACE keyword. When you replace the values of a dimension, all variables and relations in the target workspace dimensioned by it are affected. When a variable or relation is not being imported at the same time, replacing the values of one of its dimensions could result in the loss of its data.

LIST
LIST ONLY
Produces a list of the definitions. For dimensions, the output lists the number of values in each dimension, as they are imported into the target workspace. For composites, the output lists the number of dimension value combinations. IMPORT also indicates the number of bytes read and the elapsed time every two minutes or, in any case, at the end of the import procedure.

When you define a conjoint or composite that uses an index type other than the default, the IMPORT LIST command displays the index type. When you use the default index type (HASH for conjoints, BTREE for composites), that information is not displayed.

EXPORT (to EIF) sends the list to the current outfile. When you specify LIST ONLY, you get *only the listing* without actually importing anything.

ONLY
Causes Oracle OLAP to place the correct values in the EIFNAMES and EIFTYPES options without actually importing them. However, Oracle OLAP does not produce a full listing of the object definitions. To produce the list, specify the LIST keyword before the ONLY keyword.

DATA
Indicates that, for objects that already exist in the target workspace, IMPORT should update only the data associated with those objects. For formulas that already exist, IMPORT updates their EQ expressions. Objects that IMPORT creates in the target workspace are created with their full definitions, as well as any associated data. You can specify both DATA and DFNS, but when neither is specified, the default is DATA.

DFNS
Indicates that, for objects that already exist in the target workspace, IMPORT should just update definitions and leave data unchanged. The components of the definition affected by IMPORT DFNS are: LD Command, VNF, and PROPERTY. Objects that IMPORT creates in the target workspace still get their data. You can specify both DATA and DFNS, but when neither is specified, the default is DATA.

UPDATE
Indicates that IMPORT should execute an UPDATE command after importing each object. This can be useful when importing large EIF files that would otherwise cause Oracle OLAP to run out of memory. To control the frequency of updates, use the EIFUPDBYTES option.

NOPROP
Prevents any properties that you have assigned to each object from being read from the EIF file.

NASKIP
Specifies that composite tuples (indexes) that contain only `NA` data should not be imported into the target workspace. This argument has no effect on tuples that already exist in the workspace.

NLS_CHARSET *charset-exp*
Specifies the character set that Oracle OLAP will use when importing text data from the file specified by *file-id*. Normally, an EIF file contains its own specification of its character set, so that this argument is not needed. However, when the EIF file specifies the character set incorrectly or is missing the character set specification, then you must use this argument to specify the character set correctly. For information about the character sets that you can specify, see the *Oracle Database Globalization Support Guide*.

This argument must be the last one specified. When this argument is omitted, and Oracle OLAP is unable to determine the character set from the EIF file itself, then Oracle OLAP imports the data using the database character set, which is recorded in the NLS_LANG option.

Notes

EIF Options
A number of options determine how EIF files are imported and exported. These options are listed in Table 12–1, " EIF Options" on page 12-7.

Separate IMPORT Commands
The MATCH, APPEND, REPLACE, DATA, and DFNS arguments you specify affect all the objects you name to be imported. When you want to treat different objects in different ways, use separate IMPORT commands.

Relations
When you are importing a relation, IMPORT also brings in the definition and values for the related dimension as well.

Concat Dimensions
When you import a concat dimension into an analytic workspace and the concat dimension and none of its component dimensions already exist in the analytic workspace, then Oracle OLAP imports the concat dimension, its component dimensions, and the definitions of all of the dimensions.

When you import a concat dimension that does not already exist but one or more of its component dimensions already exist in the analytic workspace, then Oracle OLAP imports the concat dimension and any new component dimensions and their definitions. For the component dimensions that already exist in the analytic workspace, Oracle OLAP imports the component dimensions as it does other dimensions, obeying any MATCH, APPEND, REPLACE specifications in the IMPORT command.

When you import a concat dimension with a name and a definition of a concat dimension that already exists in the analytic workspace, then Oracle OLAP imports the concat dimension as it does other dimension.

When you import a concat dimension with the same name as one that already exists in the analytic workspace but the definition of the imported concat dimension is different than the definition of the existing concat dimension, then the definition of the existing concat dimension does not change and the definitions of the component dimensions of the existing concat dimension do not change. Only the component dimensions of the imported concat dimension that are also component dimensions of the existing concat dimension are imported. When the imported concat dimension does not share any component dimensions with the existing concat dimension, an error condition occurs. When you are importing any objects that are dimensioned by the concat dimension, then Oracle OLAP imports only the values of the object that correspond to the values of the imported dimensions.

Dimension Surrogates

You can import or export a dimension surrogate to or from an Express Interchange File (EIF). In those operations, a dimension surrogate behaves like a variable that is dimensioned by the dimension of the surrogate. In an EXPORT operation, the dimension for which the surrogate is defined is also exported. In an IMPORT operation, the dimension for which the surrogate is defined is imported but you can use the MATCH, STATUS, DATA, DFNS, APPEND, and REPLACE keywords to affect which values are imported.

Importing a dimension surrogates also imports the definition and values for the dimension for which it is a surrogate. When a dimension with the same definition already exists in the current analytic workspace, then the effects of the IMPORT keywords such as MATCH, APPEND, REPLACE, DATA, and DFNS are the same for the surrogate as they would be for a variable dimensioned by the dimension. When the name and definition of the imported surrogate is the same as a dimension surrogate that already exists in the current analytic workspace and when the imported surrogate has a value that is identical to a value in the existing surrogate, an error condition occurs.

You can import an INTEGER dimension surrogate when no object of the same name exists in the current analytic workspace or when you use the DFNS keyword. Importing an INTEGER dimension surrogate affects existing INTEGER dimension surrogates when the implicit importing of the dimension of the imported surrogate changes the values of the existing dimension.

APPEND Versus REPLACE

When you are importing an INTEGER dimension that already exists in your target workspace, the following considerations apply.

- When the imported INTEGER dimension is larger than the existing one, APPEND and REPLACE have the same effect. The dimension will end up with the number of values in the larger, imported dimension.
- When the imported INTEGER dimension is smaller, REPLACE drops the appropriate dimension values from the end of the dimension, *along with any associated data,* while APPEND leaves the existing dimension values alone.

INTEGER and SHORTINTEGER Data Types

The IMPORT command translates between the INTEGER and SHORTINTEGER data types. When you are importing a variable with one of these data types from an EIF file and it already exists in your workspace as the other type, Oracle OLAP converts the data automatically. The maximum SHORTINTEGER value is `32,767` and the minimum is `-32,767`. When you import an INTEGER value that exceeds these limits into a SHORTINTEGER variable, the result is `NA`.

TEXT and ID Data Types

When the EIF file you are importing contains ID data that you want to import into TEXT dimensions, variables, relations, or valuesets, Oracle OLAP automatically converts the ID data to text during the import process.

Existing Programs and Models

When you are importing a program or model that already exists in your workspace, you must specify DFNS. A program or a model is a definition only; it does not have any data. The default option DATA does not import the source code when it already exists.

When you define a program, you may specify a data type or a dimension name, which is used when the program is called as a function. When you specify a data type, it determines the data type of the return value. When you specify a dimension name, the return value is a single value of that dimension. When you import an existing program, the data type or the dimension in the imported program definition and the existing program definition must match. Otherwise, Oracle OLAP produces an error message.

PERMIT Commands

The PERMIT commands associated with an object are imported with the object definition. You can see them when you describe the object. However, permission conditions are not evaluated when the object is imported.

When an object with the same name already exists in the target workspace and you specify the DFNS keyword, the PERMIT commands for the object are updated. However, you must execute a PERMITRESET to put the new permission into effect. When an object with the same name already exists in the target workspace and you do *not* specify the DFNS keyword, the PERMIT commands for the object are not updated. When there is no pre-existing object in the target workspace, and you import with or without the DFNS keyword, the PERMIT commands for the object are updated, but you must execute a PERMITRESET to put the new permission into effect. (See the PERMIT command.)

When you export and import an entire workspace, then update, detach, and reattach the workspace, Oracle OLAP will ensure that all the permissions that were in effect before exporting are in place in the target workspace.

Permission Programs: Copying to and from Analytic Workspaces When you export PERMIT_READ or PERMIT_WRITE programs which are hidden, they are empty when imported. Additionally, when you outfilePERMIT_READ or PERMIT_WRITE programs which are hidden, then they are empty when infiled.

> **Tip:** Rename PERMIT_READ and PERMIT_WRITE programs before using EXPORT (to EIF) or OUTFILE After copying the programs to an analytic workspace using IMPORT (to EIF) or INFILE.

Reducing Workspace Size

You can use IMPORT in conjunction with the EXPORT command to compact an entire workspace at once. To do this, first export the workspace and then import it under a different name. You can then delete the old workspace and rename the new one with the original name.

Preserving Conjoint Type

When you export a HASH, BTREE, or NOHASH conjoint dimension to an EIF file, the conjoint type is exported along with the definition in the EIF file. When you then import the conjoint dimension into a workspace, Oracle OLAP preserves the conjoint type when you import into a new dimension or a dimension already using that conjoint type. When you import the dimension into an existing dimension that does not use the same conjoint type, Oracle OLAP does not preserve the original conjoint type that was saved in the EIF file.

EIFBYTES, EIFNAMES, and EIFTYPES

You can use the EIFBYTES option to learn the number of bytes read or written by the most recent IMPORT (EIF File) command. You can use the EIFNAMES option to get a list of the names of all the objects imported by the most recent IMPORT command and use the EIFTYPES option to learn the types of objects in that list.

The following format causes IMPORT to store information about the specified objects into the EIFNAMES and EIFTYPES options without actually importing the objects. IMPORT places a list of the object names specified by the IMPORT command in the EIFNAMES option. IMPORT also places a list of the type of each object listed in EIFNAMES into the EIFTYPES option. You may use the LIST keyword to send to the current outfile a full listing of the object definitions.

IMPORT *name* FROM EIF FILE *file-id* [LIST] ONLY

For more information, see the entries for EIFBYTES, EIFNAMES, and EIFTYPES.

Unnamed Composites

When you define variables or other objects with the SPARSE keyword in the dimension list, Oracle OLAP creates an unnamed composite that corresponds to the SPARSE dimension list. When you export or import an object with the unnamed composite in its definition, the composite is automatically exported or imported with the object. Since the unnamed composite is not a regular workspace object, you cannot import or export it independently.

Variable Segments Specified with SEGWIDTH

When you use the SEGWIDTH keyword of the CHGDFN command to specify the length of variable segments, segment information cannot be exported and imported automatically. You can save your SEGWIDTH settings by exporting the entire workspace, creating a new workspace, importing only the workspace objects into the new workspace, specifying segmentation, and then importing the variable data into the new workspace.

TEXT and NTEXT

You can export and import TEXT and NTEXT values. Both data types can be exported to a single EIF file.

- Exported TEXT values are stored in the EIF file using the character set specified for the file in the EXPORT (to EIF) command.

- Exported NTEXT values are stored in the EIF file as NTEXT (UTF8 Unicode).
- NTEXT values imported into TEXT objects are converted into the database character set. This can result in data loss when the NTEXT values cannot be represented in the database character set.
- TEXT values imported into NTEXT objects are converted into the NTEXT (UTF8 Unicode) character set.

Examples

Example 14–15 Importing Dimensions from an EIF File

This example shows how to import the contents and dimensions of two variables into the current Oracle OLAP workspace from a disk file called `finance.eif` in the current directory object.

```
IMPORT actual budget FROM EIF FILE 'finance.eif'
```

Example 14–16 IIMPORTING a Concat Dimension

This example shows the result of importing a concat dimension that has a definition that is different than a concat dimension that already exists in the current analytic workspace. Suppose that the DESCRIBE command returns the following definitions for dimensions and variables in the current analytic workspace.

```
DEFINE city TEXT DIMENSION
DEFINE state TEXT DIMENSION
DEFINE country TEXT DIMENSION
DEFINE locality DIMENSION CONCAT (city, state)
DEFINE geog DIMENSION CONCAT (locality, country)
DEFINE sales INTEGER VARIABLE <geog>
```

The following statement reports the `sales` data.

```
REPORT sales
```

The preceding statement produces the following results.

```
GEOG                SALES
------------------- -----
<city: Boston>       1000
<city: Springfield>  2000
<state: Ma>          3000
<country: Usa>       4000
```

The DESCRIBE command returns the following definitions for dimensions and variables in the `diffconcat.eif` file.

```
DEFINE CITY TEXT DIMENSION
DEFINE REGION TEXT DIMENSION
DEFINE COUNTRY TEXT DIMENSION
DEFINE GEOG DIMENSION CONCAT (CITY, REGION, COUNTRY)
DEFINE SALES INTEGER VARIABLE <GEOG>
```

The following statement reports the `sales` data for the dimension values in the analytic workspace from which you exported the concat dimension that is in the `diffconcat.eif` file.

```
REPORT sales
```

The preceding statement produces the following results.

```
GEOG               SALES
------------------ -----
<city: Boston>      1111
<city: Worcester>   2222
<region: East>      3333
<country: Usa>      4444
```

The following statement imports the `sales` variable from the `diffconcat.eif` file and implicitly imports the concat dimension `geog`. The APPEND keyword causes Oracle OLAP to add the value `Worcester` to the `city` dimension. After that, it imports new values for `sales` that correspond to `<city: Boston>`, `<city: Worcester>`, and `<country: Usa>`.

```
IMPORT sales FROM EIF FILE diffconcat.eif APPEND
```

After the import operation, reporting the SALES values produces the following results.

```
GEOG                SALES
------------------- -----
<city: Boston>       1111
<city: Springfield>  2000
<city: Worcester>    2222
<state: Ma>          3000
<country: Usa>       4444
```

IMPORT (from text)

You can use the IMPORT (from text) command to copy data from a text file into an Oracle OLAP worksheet object. A worksheet's rows are similar to the lines of a text file.

IMPORT is commonly used to copy text files into an analytic workspace from other software products.

Normally, you should use the FILEREAD command for text files instead of IMPORT. FILEREAD is more efficient and does not require a worksheet object or separate handling of each column of data.

Syntax

IMPORT *worksheetname* FROM [<u>TEXT</u>|STRUCTURED|RULED [RULER *ruler-exp*] -

PRN FILE *file-id* [STOPAFTER *n*] [TEXTSTART *schar*] [TEXTEND *echar*] -

[DELIMITER *dchar*] [NLS_CHARSET *charset-exp*]

Arguments

worksheetname
A text expression that specifies the name of an Oracle OLAP worksheet object. When you have not previously defined *worksheetname* in your workspace, IMPORT will define it for you automatically, using the default dimensions `WKSCOL` and `WKSROW`. Any previous contents of *worksheetname* will be overwritten. In any one IMPORT command, you can import only one *worksheetname* from one text file.

FROM . . . PRN
Indicates that you want to import your Oracle OLAP worksheet from a text file.

TEXT
Imports a whole source file as-is into an Oracle OLAP worksheet on a line-by-line basis. The source file is copied into a single wide worksheet column with a data type of TEXT. The column is always column 1 of the worksheet. Each line in the source file is imported into a separate cell on a separate row in the first column, using as many rows as there are lines in the source file. A blank line in the source file produces a TEXT value with zero characters (a null) in the corresponding row of the worksheet's first column. (TEXT is the default.)

STRUCTURED

Imports a source file into a target worksheet on a cell-by-cell basis, automatically performing three functions:

1. Each line of characters in the source file is copied into a single row of the target worksheet.
2. Each group of characters on a line in the source file is copied into a separate TEXT cell on the target worksheet row. A group of characters is defined by two conditions: an uninterrupted (except by a decimal point) sequence of *numbers*, or enclosure in *double quotes*. This means that numbers containing commas to mark off thousands will be split up into different cells unless the commas are first removed.
3. Any non-numeric characters not enclosed in double quotes are ignored, except minus signs that immediately precede numbers and so are copied into the same TEXT cell along with the numbers. (Be sure there are no spaces between a minus sign and its number in the source file.)

A blank line in the source file results in an `NA` in the first cell of the corresponding worksheet row.

When your file format does not conform to the pattern described here, you can use the TEXTSTART, TEXTEND, and DELIMITER keywords. These arguments let you customize the delimiters IMPORT uses to identify the start and end of each field.

RULED

Indicates import of a file on a column-by-column basis into worksheet cells of various data types. Every line in the source file must follow the same pattern of data along its length as every other line in the file. You describe this data pattern to Oracle OLAP in the one-line *ruler-exp* using the RULER keyword. IMPORT loops over each line in the source file and copies its contents into a corresponding pattern of cells on a row of the target worksheet, one row for each line. As *ruler-exp* loops over successive lines in the source file, it adds row after row to the target worksheet, building vertical *columns* of similar cells as it goes along. Status messages are sent to the current outfile every 20 rows, starting with the message `20 rows processed`.

When the source file contains records that follow several different patterns of character groups, you will have to use the less exacting options, STRUCTURED or TEXT, to import the data.

RULER *ruler-exp*

Used only with the RULED keyword to specify the data type, length, and repeat count of each character group in the record pattern of the source file. *Ruler-exp* consists of a list of character-group specifications. Each character-group specification must be separated by a comma (`,`), by backslash N (`\n`), or by a space( ). You do not have to include enough specifications to account for all the characters in the basic record pattern (or line pattern) of the source file; RULER will step to the next record as soon as it runs out of specifications on each line, regardless of how far it is from the end of the current record. Remember to enclose literal text in single quotes.

The specifications for groups of characters are of three types: `T` for TEXT, `A` for numeric (INTEGER or DECIMAL), and `S` for skip or ignore. The formats for these types are shown in Table 14–1, " Character-Group Specifications for IMPORT from Text" on page 14-59.

Table 14–1 Character-Group Specifications for IMPORT from Text

Format	Description
[*mm*]**`T`***nn*	Specifies that Oracle OLAP should copy *mm* groups (default = 1) of *nn* characters (bytes) apiece as TEXT. Specifying a group (or groups) of 0 characters leaves an empty cell(s) in the corresponding position in the worksheet. Each group may consist of up to 400098 characters. Trailing blanks are stripped.
[*mm*]**`A`***nn*	Specifies that Oracle OLAP should copy *mm* groups (default = `1`) of *nn* characters (bytes) apiece and try to convert each group to a number. When a character group cannot be converted to a number, it is copied into a TEXT cell and any trailing blanks are stripped. A valid *number* includes anything you can type for a GET(DECIMAL) function except `NA`. Commas embedded in a number before a period (decimal point) are ignored. This means that multiple numbers separated only by commas or two numbers separated only by a single period are treated as parts of a single number (when you want the numbers treated separately, insert *spaces* between them in the source file). Leading dollar signs (`$`) and trailing percent signs (`%`) are ignored, and leading and trailing spaces are stripped. Multiple periods are treated as excess decimal points and ignored (to undo the effects of dotfill). For example,`...17...` is treated as though the field is `17`. Numbers preceded by a hyphen, or a hyphen and spaces, and numbers enclosed in parentheses, are treated as negative. Specifying a group (or groups) of `0` (zero) characters leaves an empty cell (or cells) in the corresponding position in the worksheet. Each group may consist of up to 4000 characters.

Table 14–1 (Cont.) Character-Group Specifications for IMPORT from Text

Format	Description
[*mm*]S*nn*	Specifies that Oracle OLAP should skip or ignore *mm* groups of *nn* characters (bytes). The limit for *nn* is 32,767. (You would probably only use *mm* to expand this limit to handle a very long record.)

FILE *file-id*
Identifies the file you want to import. *File-id* is a text expression that represents the name of the file. The name must be in a standard format for a file identifier.

STOPAFTER *n*
Specifies that no more than *n* records should be read from the file. When STOPAFTER is omitted, Oracle OLAP will read the whole file.

TEXTSTART *schar*
The *schar* argument is a text expression that specifies a single character that you want Oracle OLAP to interpret as the start of a text field in a structured file. The default character is a double quote (`"`).

TEXTEND *echar*
The *echar* argument is a text expression that specifies a single character that you want Oracle OLAP to interpret as the end of a text field in a structured file. The default character is a double quote (`"`).

DELIMITER *dchar*
The *dchar* argument is a text expression that specifies a single character that you want Oracle OLAP to interpret as the general field delimiter in a structured file. Oracle OLAP uses the general field delimiter to identify both numeric and text fields. The default character is a comma (`,`).

NLS_CHARSET *charset-exp*
Specifies the character setthat Oracle OLAP will use when importing text data from the file specified by *file-id*. This allows Oracle OLAP to convert the data accurately from that character set. This argument must be the last one specified. When this argument is omitted, and Oracle OLAP is unable to determine the character set from the file itself, then Oracle OLAP imports the data in the database character set, which is recorded in the NLS_LANG option.

Notes

WKSROW and WKSCOL Dimensions

The WKSROW (the default worksheet row) dimension of an Oracle OLAP worksheet corresponds to the lines of a text file. The WKSCOL (the default worksheet column) dimension of a worksheet divides each row into cells that can be used to separate data types when there are potentially several types on each line of the source file. WKSROW and WKSCOL are INTEGER dimensions with values of 1, 2, 3, and so on.

Minimum Worksheet Size

Oracle OLAP sets up a minimum-size worksheet that is 63 cells square, regardless of whether or not all the cells are used. When the source text file requires an Oracle OLAP worksheet larger than the minimum, IMPORT automatically increases the dimension values of WKSCOL and WKSROW as needed.

Importing Numbers

When importing a number from a text file, IMPORT gives it an INTEGER data type.

File Transfer to Another Computer

When the file you are importing originated on another computer, ensure that its character set is appropriate. When you transfer a text file to another computer, the communications program handling the transfer makes any necessary character translations; for example, from ASCII to EBCDIC. You should set the parameters of the transfer program so that the resulting file is in the correct character set for the receiving computer.

Examples

Example 14–17 Importing Columns Without the RULER Keyword

Suppose you have a file named abctxt in your current directory. It has 10 five-digit groups of integers, followed by a group of 20 characters of text. To import this file into an Oracle OLAP worksheet called sheet1, use the following statement.

```
IMPORT sheet1 FROM RULED PRN FILE 'abctxt' ruler '10a5, t20'
```

The actual format for the file name must follow the conventions for your operating system.

Example 14–18 Importing Columns with the RULER Keyword

Suppose a file called `mix` has no line delimiters, with records containing 100 characters apiece. Each record has the character distribution illustrated in the following table.

Character	Content
1 - 10	To be ignored
11 - 17	Decimal number
18 - 28	To be ignored
29 - 30	Two single-character code
31 - 35	Integer
36 - 100	To be ignored

To import this file into an Oracle OLAP worksheet called `sheet2`, use the following statement.

```
DEFINE sheet2 WORKSHEET temp
IMPORT sheet2 FROM RULED PRN FILE 'mix' RULER -
   's10, a7, s11, 2t1, a5'
```

IMPORT (from spreadsheet)

You can use the IMPORT (from spreadsheet) command to copy data (not formulas) from a spreadsheet file into an Oracle OLAP worksheet object. A worksheet's dimensions are similar to the columns and rows of a spreadsheet. IMPORT always copies an entire spreadsheet file at a time.

IMPORT is commonly used to copy data from other software products (for example, a Lotus spreadsheet) into an Oracle OLAP workspace.

Syntax

IMPORT *worksheetname* FROM *source* [INTO *workspace*]

where:

source is one of the following:

WKS FILE *file-id* [NLS_CHARSET *charset-exp*]

WK1 FILE *file-id* [NLS_CHARSET *charset-exp*]

WRK FILE *file-id* [NLS_CHARSET *charset-exp*]

WR1 FILE *file-id* [NLS_CHARSET *charset-exp*]

DIF FILE *file-id* [NLS_CHARSET *charset-exp*]

CSV FILE *file-id* [STOPAFTER *n*|DELIMITER *dchar*|NLS_CHARSET *charset-exp*]

Arguments

worksheetname
An Oracle OLAP worksheet object. In any one IMPORT command, you can import only one *worksheetname* from one spreadsheet file. You can specify a qualified object name for the worksheet; however, when you specify the INTO *worksheet* argument, the target workspace specified must be identical. See the INTO *workspace* argument for information about where the worksheet object will be imported.

FROM WKS
FROM WK1
FROM WRK
FROM WR1
FROM DIF
Indicates that you want to import your Oracle OLAP worksheet from a 1-2-3 file, Version 1 (`WKS`) or Version 2 (`WK1`); a Symphony file, Version 1.0 (`WRK`) or Version 1.1 (`WR1`); or a data interchange format file (`DIF`).

Oracle OLAP does not recognize numbers in E format (exponential notation) in `DIF` files.

INTO *workspace*
A workspace name that identifies the attached workspace into which data will be imported. When *worksheetname* exists in the specified workspace, then its data will be overwritten by the imported data. When *worksheetname* does not already exist, IMPORT creates it in the specified workspace. IMPORT ignores an identically named worksheet when it exists in another attached workspace.

When you do not specify this argument, then Oracle OLAP does the following:

- When you have not previously defined *worksheetname* in an attached workspace, IMPORT defines it automatically in the current workspace using the default dimensions `WKSCOL` and WKSROW.
- When *worksheetname* already exists in any attached workspace, IMPORT overwrites the data it contains with the imported data.

FILE *file-id*
Identifies the file you want to import. The *file-id* argument is a text expression that represents the name of the file. The name must be in a standard format for a file identifier.

NLS_CHARSET *charset-exp*
Specifies the character set that Oracle OLAP will use when importing text data from the file specified by *file-id*. This allows Oracle OLAP to convert the data accurately from that character set. This argument must be the last one specified. When this argument is omitted, and Oracle OLAP is unable to determine the character set from the worksheet itself, then Oracle OLAP imports the data in the database character set, which is recorded in the NLS_LANG option.

FROM CSV FILE *file-id* [STOPAFTER *n*] [DELIMITER *dchar*]
Indicates that you want to import from a source file on a cell-by-cell basis. See "CSV Import" on page 14-65.

STOPAFTER *n* specifies that no more than *n* records should be read from the file. When STOPAFTER is omitted, Oracle OLAP will read the whole file.

DELIMITER *dchar* identifies the single character (*dchar*) that you want Oracle OLAP to interpret as the general field delimiter. The default value is comma.

Notes

WKSCOL and WKSROW Dimensions

The default dimensions of an Oracle OLAP worksheet are `WKSCOL` and WKSROW, which correspond to the columns and rows of a spreadsheet. `WKSCOL` and WKSROW are INTEGER dimensions with values of 1, 2, 3, and so on. When these dimensions already exist in an attached workspace but not in the current workspace, the IMPORT command will fail when it tries to create these dimensions. You can prevent this problem by first defining the worksheet with different dimensions. (See "Worksheet Dimensions" on page 10-85 for more information.)

Addition of Cells when Needed

When the source spreadsheet contains more cells than are defined by the dimensions of the worksheet, IMPORT automatically adds dimension values to provide the required number of cells.

Empty and NA Cells

IMPORT merges the source file with the worksheet on a cell-by-cell basis. Cells from the source file that are not empty, even when they just contain `NA`, overwrite the contents of the corresponding cells in the worksheet; empty cells in the source file do not overwrite the worksheet; source-file cells beyond the end of the current worksheet are appended to it so that no data is discarded.

Numbers in DIF Files

When importing any number from DIF files, IMPORT gives it a DECIMAL data type.

CSV Import

The CSV import option automatically performs the following functions when importing from a source file into the cells of a worksheet:

- Each line of characters in the source file is copied into a single row in the target worksheet.

- Each group of characters on a line in the source file is copied into a separate TEXT cell in the target worksheet row, and groups are separated by the delimiter character.

When a group of characters is inside double quotation marks:

- A delimiter character found in this group is treated as a literal.
- When a double quotation mark occurs *within* this group, it must be followed by another double quotation mark.
- A linefeed (`\n`) found within the group is ignored.
- Spaces or tabs found before a starting quotation mark and after an end quotation mark are ignored.

TEXT, not NTEXT

All imported text is rendered in the database character set in the worksheet object. The NTEXT data type is not supported in worksheets.

Examples

Example 14–19   Importing a DIF File

This example shows how to import a spreadsheet in DIF format in a file called `mortgage.dif`. We define the worksheet first as a temporary object, which saves memory and storage space. IMPORT would define the worksheet *sheet1* automatically when it did not already exist. When it had already been used in a previous IMPORT command, any data in it would be overwritten with new data.

```
DEFINE sheet1 WORKSHEET TEMP
IMPORT sheet1 FROM DIF FILE 'mortgage.dif'
```

15 INF_STOP_ON_ERROR to LIKEESCAPE

This chapter contains the following OLAP DML statements:

INF_STOP_ON_ERROR

The INF_STOP_ON_ERROR option specifies the behavior of Oracle OLAP when an error is reached when reading from a file using the INFILE command.

Syntax

INF_STOP_ON_ERROR = {YES|<u>NO</u>}

Arguments

YES
When an error occurs, report the error and stop reading from the file.

NO
When an error occurs, report the error and continue reading from the file.

Example

Example 15–1 Using INF_STOP_ON_ERROR with DBMS_EXECUTE

Assume that you have an file named `attachmyaw.inf` that includes the following OLAP DML statement that detaches an analytic workspace named `myaw`

```
AW DETACH myaw
```

Assume that the `myaw` workspace is *not* attached when a SQL application issues the `DBMS_AW.EXECUTE` statement with the OLAP DML INFILE command to read the `attachmyaw.infinfile` file.

When the INF_STOP_ON_ERR option is set to `NO` then the error `Analytic workspace MYAW is not attached` is reported, Oracle OLAP continues to read the file, and the `DBMS_AW.EXECUTE` procedure completes successfully.

```
DBMS_AW.EXECUTE('INF_STOP_ON_ERR = NO ');
DBMS_AW.EXECUTE('INFILE attachmyaw.inf');

The current directory is MYDIR.
ERROR: (ORA-34344) Analytic workspace MYAW is not attached.
ERROR: (ORA-34344) Analytic workspace MYAW is not attached.

PL/SQL procedure successfully completed.
```

When the INF_STOP_ON_ERR option is set to `YES` then the error `Analytic workspace MYAW is not attached` is reported, Oracle OLAP stops reading the file, and the `DBMS_AW.EXECUTE` procedure aborts.

```
DBMS_AW.EXECUTE('INF_STOP_ON_ERR = YES ');
DBMS_AW.EXECUTE('INFILE attachmyaw.inf');


The current directory is MYSPL.
DECLARE
  *
ERROR at line 1:
ORA-35166: (ORA-34344) Analytic workspace MYAW is not attached.
ORA-06512: at "SYS.DBMS_AW", line 27
ORA-06512: at "SYS.DBMS_AW", line 115
ORA-06512: at line 8
```

INFILE

The INFILE command causes Oracle OLAP to read statement input from a specified file.

Syntax

INFILE {*file-id*|EOF} [NOW] [NLS_CHARSET *charset-exp*]

Arguments

file-id

The name of a file from which to read input. *File-id* is a text expression that represents the name of the file. The name must be in a standard format for a file identifier.

The input file must contain only OLAP DML statements, along with appropriate responses to any prompts generated by the statements. Each statement or response must appear on a separate line in the file.

EOF

Terminates the reading of input from the current file and causes Oracle OLAP to resume reading input from the location from which the INFILE command was executed. Use of INFILE EOF is optional. See "About the Input File" on page 15-6 and "INFILE with Both NOW and EOF" on page 15-7.

NOW

Indicates that Oracle OLAP should open the input file specified in the INFILE and read its statements immediately upon encountering the INFILE instead of waiting until the program containing the INFILE is finished. This has the effect of nesting the input file's statements within the program. See "INFILE with Both NOW and EOF" on page 15-7. This argument must be specified after *file-id.*

NLS_CHARSET *charset-exp*

Specifies the character set that Oracle OLAP will use when reading data from the file specified by *file-id*. This allows Oracle OLAP to convert the data accurately into the current character set, as identified by the NLS_LANG option. This argument must be specified after *file-id*. When this argument is omitted, then Oracle OLAP handles the data in the file as having the database character set, which is recorded in the NLS_LANG option.

Notes

File Reading and Writing Options

A number of options are important during file read and write operations. These options are listed in Table 15–1, " File Reading and Writing Options" on page 15-6.

Table 15–1 File Reading and Writing Options

Statement	Description
ECHOPROMPT	An option that determines whether or not input lines and error messages should be echoed to the current outfile.
INF_STOP_ON_ERROR	An option that specifies the behavior of Oracle OLAP when an error is reached when reading from a file using the INFILE command
ESCAPEBASE	An option that contains the type of escape that is produced by the INFILE keyword of the CONVERT function.
OUTFILEUNIT	(Read-only) An option that contains the file unit number of the current OUTFILE destination, set by the last OUTFILE command.

About the Input File

When the end of the input file is reached, Oracle OLAP resumes reading input from the location from which the INFILE command was executed. This could be another input file. You do not need to end the input file with the statement INFILE EOF.

INFILE ignores trailing blanks at the end of a line, or between the last text on a line and a continuation mark. INFILE also ignores blank lines.

When you use the NOW keyword and the input file ends with a continued statement, the statement is ignored. For example, if the file ends with "show - ," Oracle OLAP ignores the SHOW command.

Using INFILE in a Program

When you include an INFILE command without the NOW keyword in a program, the INFILE command is not executed until the program has finished executing. In a nested program, it is not executed until all the programs involved have finished executing. Also, when several INFILE commands have been executed by a program, the input files are read in the opposite order from which they were specified.

For example, assume that `program.a` calls `program.b` which calls `program.c`, and each program contains two INFILE commands, one before and one after the call to the next program (as illustrated in the following code). In this case, the order of execution is: `a2,b2, c2, c1, b1, a1`.

```
program.a
   INFILE a1
   "
        program.b
           INFILE b1
           "
                program.c
                   INFILE c1
                   INFILE c2
           "
           INFILE b2
   "
   INFILE a2
```

When you include an INFILE command in a program with the NOW keyword, the INFILE command executes immediately. However, INFILE with the NOW keyword requires more space than usual on the program stack. To conserve stack space, you should use the NOW keyword only when it is necessary.

INFILE with NOW Outside of Programs

The NOW keyword is intended for use within programs, but you can use it at any time. When you use it when the input file would not ordinarily be deferred, the NOW keyword has no visible effect. However, since it requires extra stack space, you should not use it in these situations.

INFILE with Both NOW and EOF

When you use both the NOW and EOF keywords, the NOW keyword is ignored.

Displaying Infiled Statements and Responses

When you want the statements from a disk file to be copied to a debugging file as they are executed, see DBGOUTFILE.

Permission Programs: Copying to and from Analytic Workspaces When you export PERMIT_READ or PERMIT_WRITE programs which are hidden, they are empty when imported. Additionally, when you outfile PERMIT_READ or PERMIT_WRITE programs which are hidden, then they are empty when infiled.

> **Tip:** Rename PERMIT_READ and PERMIT_WRITE programs before using EXPORT (to EIF) or OUTFILE After copying the programs to an analytic workspace using IMPORT (from EIF) or INFILE.

Examples

Example 15–2 Reading the Input File Immediately

The following line of code in a program causes the file called `newdefs` to be read in immediately.

```
INFILE newdefs NOW
```

INFO

The INFO function obtains information that has been produced by the FORECAST, PARSE, or REGRESS command or that has been produced for a model in your analytic workspace.

Because the syntax of the INFO function is different depending on the type of information being obtained, four separate entries are provided:

- INFO (FORECAST)
- INFO (MODEL)
- INFO (PARSE)
- INFO (REGRESS)

INFO (FORECAST)

The INFO (FORECAST) function obtains information produced by the FORECAST command and stored internally by Oracle OLAP. Through the use of keywords, INFO lets you extract specific pieces of information about the forecast you have calculated.

Note: Before using INFO, familiarize yourself with FORECAST.REPORTthat is a standard report of its results, which may give you all the information you need. INFO is useful primarily for creating customized reports or for performing further analysis on the results.

Return Value

The return value depends on the keyword you use, as described in the tables in this entry. INFO returns `NA` when you use an index that is out of range or for any choice that does not apply to the forecasting method last used. For example, when your forecast formula has two coefficients and you request the twelfth one, INFO returns `NA`.

Syntax

INFO(FORECAST *choice* [*index*])

Arguments

FORECAST
Indicates that you want to obtain information produced by the FORECAST command.

choice
The specific information you want. The choices available for FORECAST are listed in Table 15–2, " Choices for All Methods", Table 15–3, " Choices for TREND and EXPONENTIAL Forecasts", and Table 15–4, " Choices for WINTERS Forecasts". Choices marked as indexed require the *index* argument.

index
An INTEGER expression that specifies which result you want for a choice that can have several different results. For example, a trend equation might have several

coefficients. You would use *index* to specify which coefficient you want information about. When you omit *index* for a choice that requires it, an error occurs.

Table 15–2 Choices for All Methods

Keyword	Type	Indexed?	Meaning
AVAILABLE	BOOL	No	Is there a computed forecast for which to obtain information?
DEPENDENT	TEXT	No	The variable or expression being forecast.
METHOD	TEXT	No	The forecast method.
MAPE	DEC	No	The mean absolute percent error (a measure of goodness of fit).
LENGTH	INT	No	The number of forecast periods calculated.
TIME	TEXT	No	The dimension along which forecasting is performed.
FCNAME	TEXT	No	The name of the variable that contains the fitted and forecasted values (`NA` when no forecasts were saved).

Table 15–3 Choices for TREND and EXPONENTIAL Forecasts

Keyword	Type	Indexed?	Meaning
FORMULA	TEXT	No	The text of the forecasting equation.
NUMCOEFS	INT	No	The number of coefficients.
COEFFICIENT	DEC	Yes	The specified coefficient in the forecasting equation; *index* specifies which one you want.

Table 15–4 Choices for WINTERS Forecasts

Keyword	Type	Indexed?	Meaning
PERIODICITY	INT	No	The number of periods in a seasonal cycle.
ALPHA	DEC	No	The smoothing constant for the smoothed data series.
BETA	DEC	No	The smoothing constant for the seasonal index series.

Table 15–4 (Cont.) Choices for WINTERS Forecasts

Keyword	Type	Indexed?	Meaning
GAMMA	DEC	No	The smoothing constant for the trend series.
STSMOOTHED	DEC	No	The starting value of the smoothed data series.
STSEASONAL	DEC	Yes	The starting values for the seasonal index series; *index* specifies which one you want.
STTREND	DEC	No	The starting value for the trend series.
FCSMOOTHED	TEXT	No	The variable that holds the smoothed data series.
FCSEASONAL	TEXT	No	The variable that holds the seasonal index series.
FCTREND	TEXT	No	The variable that holds the trend series.

Notes

Determining Results Availability

When you try to extract information without having calculated a forecast, INFO produces an error. You can use the keyword AVAILABLE to determine whether any results are currently available.

Examples

Example 15–3 Getting Forecast Information

In this example, suppose you forecasted sales.

The following statements limit the dimensions of the `sales` variable, then obtain the formula for your forecast.

```
LIMIT product TO 'Sportswear'
LIMIT district TO 'Chicago'
LIMIT month TO 'Jan95' TO 'Dec96'
FORECAST LENGTH 12 METHOD EXPONENTIAL FCNAME fcst time -
month sales
SHOW INFO(FORECAST FORMULA)
```

These statements produce the following output.

```
87718.0009541865 * (1.005533834579 ** MONTH)
```

The next statement obtains the mean absolute percent error for your forecast.

```
SHOW INFO(FORECAST MAPE)
```

This statement produces the following output.

```
.17
```

INFO (MODEL)

The INFO (MODEL) function obtains information that is produced for the models in your analytic workspace and stored internally by Oracle OLAP. Through the use of keywords, INFO lets you extract specific pieces of information about the structure of a compiled model or the status of a model that you have run in your current session.

Note: Before using INFO, familiarize yourself with the reports created by MODEL.COMPRPT, MODEL.DEPRT, and MODEL.XEQRPT that might give you all the information you need.

Return Value

The return value depends on the keyword you use, as described in the tables in this entry. INFO returns `NA` when you use an index that is out of range or when you request information that is not relevant. For example, if the model contains 5 statements and you request information about statement 6, INFO returns `NA`; or if you specify the DIMENSION REFERENCE choice when the assignment target is actually a variable, INFO returns `NA`.

Syntax

INFO(MODEL *choice* [*index1* [*index2* [*index3*]]])

where:

index is an argument specifies the result you want for a choice that can have several different results. Depending on the keyword choice, you can supply one or more of the following index arguments:

block-num
dimension-num
element-num
model-num
qualifier-num
source-num
stmnt-num

Arguments

MODEL
Indicates that you want to obtain information about a model in your analytic workspace. INFO returns information about the model that you have most recently defined or considered in the current session (see the DEFINE MODEL and CONSIDER commands).

choice
A keyword that specifies the information you want. The choices available for models are listed in the following tables that represent different informational categories:

- Table 15–5, " INFO (MODEL) Choices to Retrieve General Information About the Model"
- Table 15–6, "INFO (MODEL) Choices to Retrieve Information about the Structure of the Model"
- Table 15–7, " INFO (MODEL) Choices to Retrieve Information about Target, Sources, and Dependencies". These choices provide information about statements that are equations. Equations have the form *assignment target = expression.* The expression can refer to one or more data sources. Assignment targets and data sources can be either variables or dimension values, and they can have qualifiers that affect their dimensionality.
- Table 15–8, "I NFO (MODEL) Choices to Retreive Information About Execution Status". All of these choices (*except* XEQSTATUS) are relevant only after running a model with a simultaneous block. When the current model has not been compiled, Oracle OLAP returns an error when you use any choice except AVAILABLE or NAME.

Each table consists of three columns that provide the following information: keyword, data type of returned value; index argument associated with the keyword.

Table 15–5 INFO (MODEL) Choices to Retrieve General Information About the Model

Keyword(s)	Data Type	Index Argument(s); Meaning
AVAILABLE	BOOL	(No arguments) Is there a model for which information is available?
NAME	TEXT	[MODEL *model-num*] Without *model-num* (or with *model-num* equal to 0), the name of the current model. With *model-num* greater than 0, the name of the included model that is the specified *model-num* within the current model.
COUNT STATEMENTS	INT	(No arguments) The number of statements in the current model. The count includes comments, equations, and DIMENSION and INCLUDE commands (if any), it but does not include the statements in an included model.
STATEMENT	TEXT	*stmnt-num* The text of statement *stmnt-num.*
SIMULTANEOUS	BOOL	(No arguments) Does the current model contain a simultaneous block?

Table 15–6 INFO (MODEL) Choices to Retrieve Information about the Structure of the Model

Keyword(s)	**Data Type**	**Index Argument(s); Meaning**
COUNT ELEMENTS	INT	[BLOCK *block-num*] Without *block-num,* the number of blocks in the current model. With *block-num,* the total number of statements and nested blocks within block *block-num* in the current model. When you request further information about a particular element (for example, with the TYPE ELEMENT choice), you always specify the block number to which the element belongs as well as the number of the element within that block.
TYPE ELEMENT	TEXT	*element-num* BLOCK *block-num* Returns BLOCK or STATEMENT, depending on whether element *element-num* of block *block-num* is a nested block or a statement.
NUMBER BLOCK	INT	*element-num* BLOCK *block-num* The block number of the nested block that is element *element-num* of block *block-num.*
TYPE BLOCK	TEXT	*block-num* Returns SIMPLE, STEP-FORWARD, STEP-BACKWARD, or SIMULTANEOUS, depending on the execution type of block *block-num.*
COUNT DIMS	INT	[BLOCK *block-num*] Without *block-num,* the number of model dimensions of the current model. With *block-num,* the number of step-forward, step-backward, or simultaneous dimensions of block *block-num* within the current model.
DIMENSION	TEXT	*dimension-num* [BLOCK *block-num*] Without *block-num,* the name of model dimension *dimension-num* of the current model. With *block-num,* the name of the specified step-forward, step-backward, or simultaneous dimension of block *block-num.*

Table 15–6 INFO (MODEL) Choices to Retrieve Information about the Structure of the Model

Keyword(s)	Data Type	Index Argument(s); Meaning
NUMBER STATEMENT	INT	*element-num* BLOCK *block-num* The statement number of the statement that is element *element-num* of block *block-num.* The statement number refers to the position of the statement within its own model. To request further information about the statement (for example, with the HIDDEN choice), its model must be the model that you are currently considering.
HIDDEN	BOOL	*stmnt-num* Has statement *stmnt-num* been masked by a subsequent statement?
NUMBER MODEL	INT	*element-num* BLOCK *block-num* The number of the included model from which the statement that is element *element-num* of block *block-num* is taken.

Table 15–7 INFO (MODEL) Choices to Retrieve Information about Target, Sources, and Dependencies

Keyword(s)	Data Type	Index Argument(s); Meaning
COUNT SOURCES	INT	STATEMENT *stmnt-num* The number of data sources in statement *stmnt-num* within the current model.
TYPE REFERENCE	TEXT	STATEMENT *stmnt-num* [SOURCE *source-num*] Without *source-num,* the object type of the assignment target of statement *stmnt-num.* With *source-num,* the object type of data source *source-num* in statement *stmnt-num.* The object type is VARIABLE when the reference is to a variable. The type is DIMENSION when the reference is to the value of a dimension.

Table 15–7 (Cont.) INFO (MODEL) Choices to Retrieve Information about Target, Sources, and Dependencies

Keyword(s)	**Data Type**	**Index Argument(s); Meaning**
VARIABLE REFERENCE	TEXT	STATEMENT *stmnt-num* [SOURCE *source-num*] Without *source-num,* the name of the variable that is the assignment target of statement *stmnt-num.* With *source-num,* the name of the variable that is data source *source-num* in statement *stmnt-num.*
VALUE REFERENCE	TEXT	STATEMENT *stmnt-num* [SOURCE *source-num*] Without *source-num,* the dimension value that is the assignment target of statement *stmnt-num.* With *source-num,* the dimension value that is data source *source-num* in statement *stmnt-num.*
DIMENSION REFERENCE	TEXT	STATEMENT *stmnt-num* [SOURCE *source-num*] Without *source-num,* the model dimension of the target dimension value in statement *stmnt-num.* With *source-num,* the model dimension of source dimension value *source-num* in statement *stmnt-num.*
COUNT QUALIFIERS	INT	STATEMENT *stmnt-num* [SOURCE *source-num*] Without *source-num,* the number of qualifiers of the assignment target in statement *stmnt-num.* With *source-num,* the number of qualifiers of data source *source-num* in statement *stmnt-num.*
TYPE QUALIFIER	TEXT	*qualifier-num* STATEMENT *stmnt-num* [SOURCE *source-num*] Without *source-num,* the qualifier type of qualifier *qualifier-num* of the target of statement *stmnt-num.* With *source-num,* the qualifier type of qualifier *qualifier-num* of data source *source-num* in statement *stmnt-num.* The qualifier type can indicate dimensional dependence: LAG (previous dimension values only), LEAD (later values only), BOTH (both previous and later values), and VARIABLE (either previous or later values, depending on the value of a variable when the model is run). The qualifier type can also be QDR (qualified data reference).

Table 15–7 (Cont.) INFO (MODEL) Choices to Retrieve Information about Target, Sources, and Dependencies

Keyword(s)	**Data Type**	**Index Argument(s); Meaning**
DIMENSION QUALIFIER	TEXT	*qualifier-num* STATEMENT *stmnt-num* [SOURCE *source-num*] Without *source-num,* the dimension of qualifier *qualifier-num* of the assignment target in statement *stmnt-num.* With *source-num,* the dimension of qualifier *qualifier-num* of data source *source-num* in statement *stmnt-num.*

Table 15–8 I NFO (MODEL) Choices to Retreive Information About Execution Status

Keyword(s)	**Data Type**	**Index Argument(s); Meaning**
XEQSTATUS	TEXT	[BLOCK *block-num*] Without *block-num,* the execution status of the model as a whole; when the model has not been run, the status is NOT EXECUTED. With *block-num,* the execution status of block *block-num*; when the model has not been run, an error is returned. When the model has been run, the status for the model as a whole or for a block can be SOLVED, DIVERGED, or FAILED TO CONVERGE. The status of an outer-level block can be EXECUTION INCOMPLETE when a nested block within it diverged or failed to converge.
COUNT ITERATIONS	INT	BLOCK *block-num* The number of iterations that were performed for block *block-num* before it was solved or it diverged or failed to converge.
DAMP	DEC	(No arguments) The value of the MODDAMP option when the model was run. (Relevant only when the solution method is GAUSS.)
DIVERGSTMT	INT	BLOCK *block-num* The element number of the statement that diverged during the calculations for block *block-num.*

Table 15–8 I(Cont.) NFO (MODEL) Choices to Retreive Information About Execution

Keyword(s)	Data Type	Index Argument(s); Meaning
GAMMA	INT	(No arguments) The value of the MODGAMMA option when the model was run.
MAXITERS	INT	(No arguments) The value of the MODMAXITERS option when the model was run.
OVERFLOW	INT	(No arguments) The value of the MODOVERFLOW option when the model was run.
SIMULTYPE	TEXT	(No arguments) The value of the MODSIMULTYPE option when the model was run: AITKENS or GAUSS.
TOLERANCE	INT	(No arguments) The value of the MODTOLERANCE option when the model was run.

block-num
An INTEGER expression that specifies the block for which you want information. *Block-num* corresponds to the block numbers that are identified in the report produced by the MODEL.COMPRPT program.

dimension-num
An INTEGER expression that specifies the model dimension or block dimension for which you want information. For the model as a whole, the first dimension listed for the model is *dimension-num* 1, and so on. For example, assume that the MODEL.COMPRPT specifies the model dimensions as `<line month>`. In this case, `line` is *dimension-num* `1` and `month` is *dimension-num* `2`. For a simultaneous block in the current model, the first dimension of the block is *dimension-num* `1`, and so on. A step-forward or step-backward block has a single dimension, so the dimension of the block is always *dimension-num* `1`. To see a list of the dimensions for the model as a whole and for each block of the model, you can run the MODEL.COMPRPT program.

element-num
An INTEGER expression that specifies the element for which you want information. When you request information about an element, you always specify the block

number to which the element belongs. An element is either a statement in the specified block, or it is a nested block within the specified block. The element numbers correspond to the order of the statements and blocks in the compiled model. You can run the MODEL.COMPRPT program to see the list of elements in the compiled model.

For example, suppose the current model has the following compiled structure.

```
block 1
statement a
  block 2
  statement b
  statement c
  END block 2
statement d
END block 1
```

When you request information about `block 1` in the preceding model, `statement a` is *element-num* `1`; `block 2` is *element-num* `2`; and `statement d` is *element-num* `3`. When you request information about `block 2`, `statement b` is *element-num* `1` and `statement c` is *element-num* `2`.

model-num
For a hierarchy of included models, an INTEGER expression that specifies the model for which you want information. The model you are currently considering is *model-num* `0` (zero), the model it includes is *model-num* `1`, and so on. The root model has the highest model number in the hierarchy.

qualifier-num
An INTEGER expression that specifies the qualifier for which you want information. Qualifiers change the dimensionality of a variable or dimension value reference. The reference can be qualified by a function, such as LAG, LEAD, or TOTAL or by a qualified data reference (QDR). To see the qualifiers for a statement, you can run the MODEL.DEPRT program for the model that contains the statement.

For each equation in the model, the MODEL.DEPRT report lists the assignment target and its qualifiers on one line, followed by the data sources. Each data source is listed on a separate line, together with its qualifiers. The MODEL.DEPRTreport also specifies the type of each qualifier: LAG, LEAD, BOTH, VARIABLE, or QDR (see the TYPE QUALIFIER choice in the third group of INFO keyword choices).

For the target and each source, *qualifier-num* corresponds to the order in which the qualifiers are listed in the MODEL.DEPRT report.

source-num
An INTEGER expression that specifies the data source for which you want information. In a calculation, each reference to a variable or a dimension value is counted as a source of data for the assignment target. A constant value is not counted as a source.

To see the data sources in a statement, you can run the MODEL.DEPRT program for the model that contains the statement. For each equation in the model, the MODEL.DEPRT report lists the assignment target on one line, followed by its data sources. Each data source is listed on a separate line.

stmnt-num
An INTEGER expression that specifies the statement for which you want information. *Stmnt-num* always refers to a statement from the model you are currently considering. It does not refer to a statement taken from an included model.

To see the statement numbers in the current model, you can run the MODEL.COMPRPT program. To the left of each statement, the report lists the model from which the statement is taken and the statement number within that model.

Notes

Determining Results Availability
You can use the keyword AVAILABLE to determine whether any model results are currently available. When you try to extract any other information without having considered or defined a model in your current session, INFO produces an error.

Examples

Example 15–4 Getting Qualifier Information

Assume that the following statement is statement 3 of a model called `income.plan`.

```
budget(line revenue) = LAG(actual(line revenue), 1, month) -
   + plan.factor
```

You can run the `MODEL.DEPRPT` program to see the qualifiers of the target and sources in this statement.

```
MODEL.DEPRPT income.plan
```

This statement produces the following output.

```
MODEL INCOME.PLAN
...
3    BUDGET(QDR <LINE>):
       ACTUAL(LAG <MONTH>)(QDR <LINE>)
       PLAN.FACTOR
...
```

This report shows that the assignment target, `budget`, has two data sources, `actual` and `plan.factor`.

Example 15–5   Checking Qualifier Information

The following statements make INCOME.PLAN the current model and check the number and type of the qualifiers of the assignment target of statement 3.

```
CONSIDER income.plan
SHOW INFO(MODEL COUNT QUALIFIERS STATEMENT 3)
```

These statements produce the following output.

```
1
```

The OLAP DML statement

```
SHOW INFO(MODEL TYPE QUALIFIER 1 STATEMENT 3)
```

produces the following output.

```
QDR
```

Example 15–6 Checking Different Data Sources

The following statements check the number and type of the qualifiers of the two data sources in statement 3.

The OLAP DML statement

```
SHOW INFO(MODEL COUNT QUALIFIERS STATEMENT 3 SOURCE 1)
```

produces the following output.

```
2
```

The OLAP DML statement

```
SHOW INFO(MODEL TYPE QUALIFIER 1 STATEMENT 3 SOURCE 1)
```

produces the following output.

```
LAG
```

The OLAP DML statement

```
SHOW INFO(MODEL TYPE QUALIFIER 2 STATEMENT 3 SOURCE 1)
```

produces the following output.

```
QDR
```

The OLAP DML statement

```
SHOW INFO(MODEL COUNT QUALIFIERS STATEMENT 3 SOURCE 2)
```

produces the following output.

```
0
```

INFO (PARSE)

The INFO (PARSE) function obtains information produced by the PARSE command and stored internally by Oracle OLAP. Through the use of keywords, INFO lets you extract specific pieces of information about the expression that you have parsed.

Return Value

The return value depends on the keyword you use, as described in Table 15–9. When you try to extract unavailable information or use an an index that is out of range, INFO returns `NA`. For example, if you parse a phrase that contains four expressions and then ask for the twelfth FORMULA, INFO will return `NA`.

Syntax

INFO(PARSE *choice* [*index*])

Arguments

PARSE
Indicates that you want to obtain information produced by the PARSE command.

choice
The specific information you want. The choices available for PARSE are listed in Table 15–9, " INFO PARSE Keywords". Choices marked as indexed can take the optional *index* argument.

index
An INTEGER expression that specifies which result you want for a choice that can have several different results. For example, when you parse text that contains three expressions, each expression has its own formula and data type. You would use *index* to specify which expression you are interested in.

When you omit *index,* INFO returns all the information as a multiline value.

Table 15–9 INFO PARSE Keywords

Keyword	Type	Indexed?	Meaning
PARSEABLE	BOOL	No	Was Oracle OLAP able to parse the text?
ERRORTEXT	TEXT	No	The text of an error message when the expressions were not parsed.
NUMFORMULAS	INT	No	The number of expressions (formulas) that were parsed.
NUMDIMS	INT	No	The number of dimensions in the union of all the expressions that were parsed.
FORMULA	TEXT	Yes	The text (formula) of the specified expression; *index* specifies which one you want.
DATA	TEXT	Yes	The data type of the specified expression.
TYPE	TEXT	Yes	The type of object of the specified expression; when the expression is the name of an object, it returns the type; when the expression is a qualified data reference, it returns QDR; when the expression is anything else, it returns EXP.
DIMENSION	TEXT	Yes	The name of the specified dimension in the union of all dimensions of the expressions.

Examples

Example 15–7 Getting Parsed Information

In a simple report program, you want to allow the user to specify the data to be reported as an argument to the program. You want to allow the user to specify an expression, as well as the name of a data variable. You cannot process expression arguments with the ARGS command, so you use PARSE and INFO to parse the program arguments and produce the report.

The following statements create a simple report program.

```
DEFINE report1 PROGRAM
PROGRAM
PUSH month product district DECIMALS
DECIMALS = 0
LIMIT month TO FIRST 2
LIMIT product TO ALL
LIMIT district TO 'Chicago'
PARSE ARGS
REPORT ACROSS month: WIDTH 8 <&INFO(PARSE FORMULA 1) -
        WIDTH 13 &INFO(PARSE FORMULA 2)>
POP month product district DECIMALS
END
```

When users run the program, they can supply either the name of a variable (`sales`) or an expression (`sales-expense`) or both as arguments.

The following statement

```
REPORT1 sales sales-expense
```

produces the following output.

```
DISTRICT: CHICAGO
             --------------------MONTH--------------------
             --------Jan95--------- --------Feb95---------
PRODUCT       SALES   SALES-EXPENSE  SALES   SALES-EXPENSE
------------ -------- ------------- -------- -------------
Tents          29,099         1,595   29,010         1,505
Canoes         45,278           292   50,596           477
Racquets       54,270         1,400   58,158         1,863
Sportswear     72,123         7,719   80,072         9,333
Footwear       90,288         8,117   96,539        13,847
```

INFO (REGRESS)

The INFO (REGRESS) function obtains information produced by the REGRESS command and stored internally by Oracle OLAP. Through the use of keywords, INFO lets you extract specific pieces of information about the regression you have calculated.

Note: Before using INFO, familiarize yourself with REGRESS.REPORT that produces a standard report of its results, which might give you all the information you need. INFO is useful primarily for creating customized reports or for performing further analysis on the results

Return Value

The return value depends on the keyword you use, as described in Table 15–10, " INFO REGRESS Keywords".

Syntax

INFO(REGRESS *choice* [*index*])

Arguments

REGRESS
Indicates that you want to obtain information produced by the REGRESS command.

choice
The specific information you want. The choices available for REGRESS are listed in Table 15–10, " INFO REGRESS Keywords". Choices marked as indexed require the *index* argument.

index
An INTEGER expression that specifies which result you want for a choice that can have several different results. For example, in a regression there may be more than one independent variable. You would use *index* to specify which independent variable you want information about. When you omit *index* for a choice that requires it, an error occurs.

Table 15–10 INFO REGRESS Keywords

Keyword	**Type**	**Indexed?**	**Meaning**
AVAILABLE	BOOL	No	Is there a computed regression from which to extract information?
DEPENDENT	TEXT	No	The name of the dependent variable in the regression.
NOINTERCEPT	BOOL	No	Was the regression calculated with the intercept suppressed?
WEIGHTED	BOOL	No	Was the last regression weighted?
WEIGHT	TEXT	No	The expression used to weight the last regression.
NUMCOEFS	INT	No	The number of coefficients.
INDEPENDENT	TEXT	Yes	An independent variable; *index* specifies which one you want (*Intercept* will be first unless it was suppressed).
COEFFICIENT	DEC	Yes	An estimated coefficient; *index* specifies which one you want.
STDERROR	DEC	Yes	The standard error of an estimated coefficient; *index* specifies which one you want.
TRATIO	DEC	Yes	The t-ratio for an estimated coefficient; *index* specifies which one you want.
NUMOBS	INT	No	The number of observations that were used.
FRATIO	DEC	No	The F-ratio for the regression.
RBSQ	DEC	No	The corrected R-squared for the regression.
FORMULA	TEXT	No	The regression formula.
STDERROREST	DEC	No	The standard error of estimate for the regression
RESET	BOOL		Use when you want to reset the original state of AVAILABLE back to `NO`

Notes

Determining Results Availability

When you try to extract information without having performed a regression, INFO produces an error. You can use the keyword AVAILABLE to determine whether any results are currently available. Once a successful regression has run, AVAILABLE remains true even when one or more unsuccessful regressions follow, because the results of the previous successful regression are still available. AVAILABLE will remain true until you use RESET to change the AVAILABLE state back to its original value of `NO`.

NA Results Due to Index

INFO returns `NA` when you use an index that is out of range. For example, when your regression has five independent variables and you request the coefficient of the twelfth one, INFO returns `NA`.

Examples

Example 15–8 Getting Regression Information

The following statement sends the third coefficient from your most recently calculated regression to the current outfile.

```
SHOW INFO(REGRESS COEFFICIENT 3)
```

This statement produces the following result.

```
7.55
```

INITCAP

The INITCAP function returns a specified text expression, with the first letter of each word in uppercase and all other letters in lowercase. Words are delimited by white space or characters that are not alphanumeric.

Return Value

The same data type as the expression.

Syntax

INITCAP (*text-exp)*

Arguments

text-exp
A text expression.

Examples

The following example capitalizes each word in the string.

```
SHOW INITCAP('the soap')
The Soap
```

INLIST

The INLIST function determines whether every line of a text value is a line in a second text value. Normally, INLIST is used to determine whether all the lines of a list (in the form of a multiline text value) can be found in a master list (in the form of a second multiline text value).

INLIST accepts TEXT values and NTEXT values as arguments. When only one argument is NTEXT, then INLIST automatically converts the other argument to NTEXT before performing the function operation.

Return Value

BOOLEAN

Syntax

INLIST(*masterlist list*)

Arguments

masterlist
A multiline text expression to which the lines of *list* are compared.

list
A multiline text expression whose lines are compared with the lines of *masterlist.* When every line of *list* can be found as a line of *masterlist,* INLIST returns the value `YES`. When one or more lines of *list* are not found in *masterlist,* INLIST returns the value `NO`.

Examples

Example 15–9 Comparing a List to a Master List

This example shows how to use INLIST to determine whether the lines of one list can be found in a master list. The master list in this case is a multiline text value in a variable called `depts`. The `depts` variable has the following values.

```
Marketing
Purchasing
Accounting
Engineering
Personnel
```

The first function call compares a list, which is specified as a text literal, with the master list.

```
INLIST(depts, 'Accounting\nPersonnel')
```

The return value is

```
YES
```

The second function call compares a variable `newlist` that has the following values,

```
Development
Accounting
```

with the master list in `depts`.

```
INLIST(depts, newlist)
```

The return value is

```
NO
```

INSBYTES

The INSBYTES function inserts one or more bytes into a text expression.

When you are using a single-byte character set, you can use INSCHARS.

Return Value

TEXT

Syntax

INSBYTES(*text-expression bytes* [*after*])

Arguments

text-expression
A `TEXT` expression into which the bytes are to be inserted. When *text-expression* is a multiline TEXT value, INSBYTES preserves the line breaks in the returned value.

bytes
One or more bytes that you insert into *text-expression.*

after
An integer that represents the byte position after which the specified *bytes* are to be inserted. The position of the first byte in *text-expression* is 1. To insert bytes at the beginning of the text, specify 0 for *after.* When you omit this argument, INSBYTES inserts the bytes after the last byte in *text-expression.*

When you specify a value for *after* that is greater than the length of *text-expression*, INSBYTES adds blanks to the last line of *text-expression*. The number of inserted blanks is the difference between the value of *after* and the length of *text-expression*. For example, `insbytes('abc' 'def' 4)` inserts one blank space before adding `def` to `abc`, resulting in:

```
abc def
```

Examples

Example 15–10 Inserting Bytes in Text

This example shows how to insert the bytes `there` in the TEXT value `hellojoe`.

The function

```
INSBYTES('hellojoe', 'there', 5)
```

returns the following value.

```
hellotherejoe
```

INSCHARS

The INSCHARS function inserts one or more characters into a text expression.

When you are using a multibyte character set, you can use the INSBYTES function instead of the INSCHARS function.

Return Value

TEXT or NTEXT

This function accepts TEXT values and NTEXT values as arguments. The data type of the return value depends on the data type of the values specified for the arguments:

- When all arguments are TEXT values, the return value is TEXT.
- When all arguments are NTEXT values, the return value is NTEXT.
- When the arguments include both TEXT and NTEXT values, the function converts all TEXT values to NTEXT before performing the function operation, and the return value is NTEXT.

Syntax

INSCHARS(*text-expression characters* [*after*])

Arguments

text-expression
The expression into which the characters are to be inserted. When *text-expression* is a multiline TEXT value, INSCHARS preserves the line breaks in the returned value.

characters
One or more characters that you insert into *text-expression.*

after
An integer that represents the character position after which the specified *characters* are to be inserted. The position of the first character in *text-expression* is 1. To insert characters at the beginning of the text, specify 0 for *after.* When you omit this argument, INSCHARS inserts the characters after the last character in *text-expression.*

When you specify a value for *after* that is greater than the length of *text-expression*, INSCHARS adds blanks to the last line of *text-expression*. The number of inserted blanks is the difference between the value of *after* and the length of *text-expression*. For example, `INSCHARS('abc' 'def' 4)` inserts one blank before adding `'def'` to 'abc', resulting in:

```
abc def
```

Examples

Example 15–11 Inserting Characters in Text

This example shows how to insert the characters `there` in the TEXT value `hellojoe`.

The function

```
INSCHARS('hellojoe', 'there', 5)
```

returns the following value.

```
hellotherejoe
```

INSCOLS

The INSCOLS function inserts into the columns of a multiline TEXT value all the columns of another TEXT value. The inserted columns are placed after the column position you specify, and the original columns in each line are moved to the right. The function returns a multiline TEXT value composed of the resulting columns.

Return Value

TEXT or NTEXT

This function accepts TEXT values and NTEXT values as arguments. The data type of the return value depends on the data type of the values specified for the arguments:

- When all arguments are TEXT values, the return value is TEXT.
- When all arguments are NTEXT values, the return value is NTEXT.
- When the arguments include both TEXT and NTEXT values, the function converts all TEXT values to NTEXT before performing the function operation, and the return value is NTEXT.

Syntax

INSCOLS(*text-expression columns* [*after*])

Arguments

text-expression
The expression into which you want to insert columns.

columns
The expression containing one or more columns in each line. All the columns of this expression will be inserted into the corresponding lines of *text-expression.*

after
An integer between 0 and 4,000 representing the column position after which columns should be inserted. The column position of the first character in each line is 1. When you do not specify *after,* insertion begins at the end of each line. The total length of a line cannot exceed 4,000 columns of single-byte characters or 2,000 columns of double-byte characters.

Notes

Number of Lines Returned

The number of lines in the return value is always the same as the number of lines in *text-expression.* When the *columns* TEXT expression has fewer lines, INSCOLS repeats its last line in each subsequent line of the return value.

After Column Beyond the End of a Line

When you specify an *after* column that is to the right of the last character in a given line in *text-expression,* the corresponding line in the return value will have spaces filling in the intervening columns.

Examples

Example 15–12 Inserting Text Columns

In the following example, a color code (stored in the multiline TEXT value `itemcolor`) is inserted into item identifiers that are stored in the `itemid` text value. The columns are inserted after Column 3.

`itemcolor` has the following value.

```
Blu
Red
Gre
Ora
```

`itemid` has the following value.

```
542-Fra
379-Eng
968-USA
369-Can
```

The INSCOLS function call

```
INSCOLS(itemid itemcolor 3)
```

returns the following.

```
542Blu-Fra
379Red-Eng
968Gre-USA
369Ora-Can
```

INSLINES

The INSLINES function inserts one or more lines into a multiline text expression.

Return Value

TEXT or NTEXT

This function accepts TEXT values and NTEXT values as arguments. The data type of the return value depends on the data type of the values specified for the arguments:

- When all arguments are TEXT values, the return value is TEXT.
- When all arguments are NTEXT values, the return value is NTEXT.
- When the arguments include both TEXT and NTEXT values, the function converts all TEXT values to NTEXT before performing the function operation, and the return value is NTEXT.

Syntax

INSLINES(*text-expression lines* [*after*])

Arguments

text-expression
A multiline expression into whose values one or more lines are to be inserted.

lines
An expression that represents one or more lines of text that you insert into *text-expression.*

after
An integer that represents the line number after which the specified *lines* are to be inserted. The position of the first line in *text-expression* is `1` (one). To insert lines at the very beginning, specify `0` (zero) for *after.* When you omit this argument, INSLINES inserts the new lines after the last line of *text-expression.*

Examples

Example 15–13 Inserting Text Lines

This example shows how to insert a new line into a multiline text value in a variable called `mktglist` with the following value.

```
Salespeople
Products
Services
```

The INSLINES function

```
INSLINES(mktglist, 'Advertising', 2)
```

returns the following.

```
Salespeople
Products
Advertising
Services
```

INSTAT

The INSTAT function checks whether a dimension or dimension surrogate value is in the current status list or whether a dimension value is in a valueset.

Return Value

BOOLEAN

YES if the value is in the current status list or in a valueset and NO if it is not.

Syntax

INSTAT(*dimension*, *value*)

Arguments

dimension
The name of the dimension, dimension surrogate, or valueset.

value
The dimension or dimension surrogate value you want to test, either a text literal (enclosed in single quotes) or an expression that specifies the value. To specify the value of a conjoint dimension or a concat dimension, enclose the value in angle brackets. For a conjoint dimension, separate the base dimension values with a comma and space. For a concat dimension, separate the base dimension and its value with a colon and a space.

Notes

Checking an Invalid Value

When you specify a dimension name and value in an INSTAT command, Oracle OLAP tells you whether that value is in the current status list for that dimension. Conversely, theISVALUE function tells you whether an item is a value of a dimension, regardless of whether it is in the status. INSTAT produces an error when *value* is not a dimension value, but ISVALUE simply returns a value of `FALSE`.

Examples

Example 15–14 Checking Current Status

In the following example, a program accepts a value of the `month` dimension as an argument. The first lines of the program use INSTAT to check whether the dimension value that was passed as an argument is in the current status for `month`. When it is, the program calls a report program. When it is not, the program branches to its error-handling section.

```
ARGUMENT onemonth month

IF INSTAT(month onemonth)
   THEN sales_report
   ELSE GOTO error
...
```

Example 15–15 Using INSTAT When the Dimension is a Conjoint Dimension

When the dimension that you specify is a conjoint dimension, then the entire value must be enclosed in single quotes. For example, suppose the analytic workspace already has a `region` dimension and a `product` dimension. The `region` dimension values include `East`, `Central`, and `West`. The `product` dimension values include `Tents`, `Canoes`, and `Racquets`.

The following statements define a conjoint dimension, and add values to it.

```
DEFINE reg.prod DIMENSION <geography product>
MAINTAIN reg.prod ADD <'East', 'Tents'> <'West', 'Canoes'>
```

To specify base positions, use a statement such as the following.

```
SHOW INSTAT(reg.prod '<1, 1>')
YES
```

To specify base text values, use a statement such as the following.

```
SHOW INSTAT(reg.prod '<\'East\', \'Tents\'>')
YES
```

Example 15–16 Using INSTAT When the Dimension is a Concat Dimension

When the dimension that you specify is a concat dimension, then you must enclose the entire `<component dimension: dimension value>` pair in single quotes.

The following statement defines a concat dimension that has as its base dimensions `region` and `product`.

```
DEFINE reg.prod.ccdim DIMENSION CONCAT(region product)
```

A report of `reg.prod.ccdim` returns the following.

```
REG.PROD.CCDIM
----------------------
<region: East>
<region: Central>
<region: West>
<product: Tents>
<product: Canoes>
<product: Racquets>
```

To specify a base dimension position, use a statement such as the following.

```
SHOW INSTAT(reg.prod.ccdim '<product: 3>')
yes
```

To specify base dimension text values, use a statement such as the following.

```
SHOW INSTAT(reg.prod.ccdim '<product: Tents>')
YES
```

INSTR

The INSTR function searches a string for a substring using characters and returns the position in the string that is the first character of a specified occurrence of the substring. INSTR calculates strings using characters as defined by the input character set.

To search a string for a substring using bytes, use INSTR.

Return Value

A nonzero INTEGER when the search is successful or `0` (zero) when it is not.

Syntax

INSTR (*string* , *substring* [, *position* [, *occurrence*]])

Arguments

string
The text expression to search.

substring
The string to search for.

position
A nonzero INTEGER indicating the character of *string* where the function begins the search. When *position* is negative, then INSTR counts and searches backward from the end of *string*. The default value of position is `1`, which means that the function begins searching at the first character of *string*.

occurrence
An INTEGER indicating which occurrence of *string* the function should search for. The value of *occurrence* must be positive. The default values of *occurrence* is `1`, meaning the function searches for the first occurrence of *substring*.

Examples

Example 15–17 Searching Forward for a String

The following example searches the string "Corporate Floor", beginning with the third character, for the string "or". It returns the position in "Corporate Floor" at which the second occurrence of "or" begins.

```
SHOW INSTR('Corporate Floor','or', 3, 2)
14
```

Example 15–18 Searching Backward for a String

In this next example, the function counts backward from the last character to the third character from the end, which is the first "o" in "Floor". The function then searches backward for the second occurrence of "or", and finds that this second occurrence begins with the second character in the search string.

```
SHOW INSTR('Corporate Floor','or', -3, 2)
2
```

INSTRB

The INSTRB function searches a string for a substring using bytes and returns the position in the string that is the first byte of a specified occurrence of the substring.

To search a string for a substring using characters, use INSTR.

Return Value

A nonzero INTEGER when the search is successful or `0` (zero) when it is not.

Syntax

INSTRB (*string* , *substring* [, *position* [, *occurrence*]])

Arguments

string
The text expression to search.

substring
The string to search for.

position
A nonzero INTEGER indicating the byte of *string* where the function begins the search. When *position* is negative, then INSTRB counts and searches backward from the end of *string*. The default value of *position* is `1`, which means that the function begins searching at the first byte of *string*.

occurrence
An INTEGER indicating which occurrence of *string* the function should search for. The value of *occurrence* must be positive. The default values of *occurrence* is `1`, meaning the function searches for the first occurrence of *substring*.

Examples

This example assumes a double-byte database character set.

```
SHOW INSTRB('Corporate Floor','or',5,2)
27
```

INTPART

The INTPART function calculates the integer part of a decimal number by truncating its decimal fraction.

Return Value

INTEGER

Syntax

INTPART(*expression*)

Arguments

expression
The decimal expression whose integer part is to be returned.

Notes

Large Values

When *expression* has a value larger than is allowed for an integer (a value between `-2,147,483,647` and `2,147,483,647`), INTPART returns an `NA` value.

Examples

Example 15–19 Calculating the Integer Part of a Decimal Number

The following example shows the integer part of the number `3.14`. The statement

```
show intpart(3.14)
```

produces the following result.

```
3
```

IRR

The IRR function computes the internal rate of return associated with a series of cash flow values. Each value of the result is calculated to be the discount rate for each period that makes the net present value of the corresponding cash flows equal to zero.

Return Value

DECIMAL

Syntax

IRR(*cashflows*, [*time-dimension*])

Arguments

cashflows
A numeric expression dimensioned by *time-dimension,* that specifies the series of cash flow values.

time-dimension
A name that specifies the time dimension. When *cashflows* has a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR, the *time-dimension* argument is optional. IRR automatically uses the DAY, WEEK, MONTH, QUARTER, or YEAR dimension of *cashflows* when you do not specify a value for *time-dimension*.

Notes

The Dimensions of the Result

The result returned by the IRR function is dimensioned by all the dimensions of *cashflows* except its time dimension. When *cashflows* is dimensioned only by the time dimension, IRR returns a single value.

The Result Value

The internal rate of return calculated by the IRR function is expressed as a decimal, so an 8.25 percent internal rate of return produces a result value of `.0825`.

Cash Flow Occurrences

All the cash flows used to compute a result value are assumed to occur at the same relative point within the period with which they are associated.

Ignored Cash Flows

Cash flows that corresponds to out-of-status dimension positions are ignored.

NASKIP Option Settings

IRR is affected by the NASKIP option. When NASKIP is set to `YES` (the default), IRR ignores `NA` values and computes the internal rate of return using the cash flows that are available. When NASKIP is set to `NO`, IRR returns `NA` when any cash flow has a value of `NA`. When all the cash flows are `NA`, IRR returns `NA` for either setting of NASKIP.

Multiple Discount Rates

Some series of cash flows have multiple discount rates, which make the net present value equal to zero. In such cases, IRR will find and return only one of these discount rates as the internal rate of return. When there is only a single solution and it is between -99.9 percent and 10,000 percent, the IRR function will find it. When IRR cannot calculate an internal rate of return, the corresponding value in the result is `NA`.

Examples

Example 15–20 Calculating the Internal Rate of Return

The following statements create a dimension called `project`, add values to it, and create a variable called `cflow`, which is dimensioned by `year` and `project`.

```
DEFINE project DIMENSION TEXT
MAINTAIN project ADD 'a' 'b' 'c' 'd' 'e'
DEFINE cflow VARIABLE DECIMAL <project year>
```

Once you have assigned the following values to CFLOW,

```
               -----------------------CFLOW----------------------
               ----------------------PROJECT---------------------
YEAR                 a          b          c          d          e
-------------- ---------- ---------- ---------- ---------- -------
Yr95            -200.00      -200.00    -300.00   -100.00  -200.00
Yr96             100.00       150.00     200.00     25.00    25.00
Yr97             100.00       400.00     200.00    100.00   200.00
```

then the following statement

```
REPORT IRR(cflow, year)
```

produces the following report of the internal rate of return.

```
               IRR(CFLOW,
PROJECT          YEAR)
-------------- ----------
a                    0.00
b                    0.84
c                    0.22
d                    0.13
E                    0.06
```

ISDATE

The ISDATE program determines whether a text expression represents a valid date. ISDATE acts as a BOOLEAN function, returning `YES` when the text expression does represent a valid date and `NO` when it does not. ISDATE does not convert the text expression to a DATE formula. ISDATE only tests a text expression to see if it can be converted to a DATE value. You must use CONVERT to make the conversion.

Return Value

BOOLEAN

Syntax

ISDATE(*test-date*)

Arguments

test-date
A single-line ID or TEXT expression to be examined to see if it represents a valid date, as defined by the DATE data type.

Notes

Valid Date Styles
For a description of the valid styles for entering dates, see DATEORDER.

Examples

Example 15–21 Testing a Text Expression

In the following statement, the ISDATE program tests a literal text expression to see if it is a valid date, and the output is sent to the current outfile.

```
SHOW ISDATE('3 5 1995')
```

This statement produces the following output.

```
YES
```

ISVALUE

The ISVALUE function tests whether a dimension or a composite has a specified value.

Return Value

BOOLEAN

Syntax

ISVALUE(*name*, *value*)

Arguments

name
The name of the dimension or the composite to be checked.

When the composite is unnamed, use the SPARSE keyword to refer to the composite (for example, `SPARSE <market product>`).

value
The value you want to test, either a text literal or text expression for an ID or TEXT dimension, an integer for an INTEGER dimension, or a combination of values enclosed by angle brackets for composites and conjoint dimensions.

Notes

Compared to INSTAT

ISVALUE tells you whether an item is a value of a dimension, but not whether a dimension value is in the current status. The INSTAT function, on the other hand, tells you whether a value of a dimension is in the current status of the dimension.

Examples

Example 15–22 Testing Valid Values

Suppose you want to find out if `Packs` is a value of the `product` dimension. The following statement produces the answer `YES` or `NO`.

```
SHOW ISVALUE(product, 'Packs')
```

Example 15–23 Embedding Quoted Strings

You can embed quoted strings within a quoted string, which is necessary when there are special characters in a base dimension value of a composite or conjoint dimension, such as `Joe's Deli`. Suppose you want to find out if `New York` and `Apple Sauce` are a valid combination of base dimension values in the `markprod` conjoint dimension. The following statement produces the answer YES or `NO`.

```
SHOW ISVALUE(markprod, '<\'New York\' \'Apple Sauce\'>')
```

When embedded quoted strings have a further level of embedding, you must use backslashes before each special character, such as the apostrophe and the backslash that must precede it in "Joe's Deli," as shown in the following statement.

```
SHOW ISVALUE(markprod, '<\'Joe\\\'s Deli\' \'Apple Sauce\'>')
```

Example 15–24 Testing Logical Position Numbers

You can test for the logical position numbers of base dimension values in a conjoint dimension. For example, suppose `market` and `product` are the base dimensions of the conjoint dimension `markprod`. The following statement tests whether or not there is a value assigned to the combination of the fourth `market` dimension value and the third `product` dimension value.

```
SHOW ISVALUE(markprod, '<4 3>')
```

JOINBYTES

The JOINBYTES function joins two or more text values as a single line.

Return Value

TEXT

Syntax

JOINBYTES(*first-expression*, *next-expression*...)

Arguments

first-expression
An expression to which JOINBYTES joins *next-expression*. When the *first-expression* has a data type other than TEXT, JOINBYTES converts it to TEXT. See "Converting Expressions to TEXT" on page 15-56.

next-expression...
One or more expressions to join with *first-expression*. When an expression you want to concatenate has a data type other than TEXT, JOINBYTES converts it to TEXT.

Notes

Converting Expressions to TEXT

When the data type of an expression is *not* TEXT, JOINBYTES automatically converts its value to TEXT before concatenating it with the other values. For example, when you put a number in a JOINBYTES function, the number is automatically converted to TEXT and no extra step is needed to accomplish this. The format of the result depends on the settings of the COMMAS, DECIMALS, and PARENS options. (When PARENS is set to `YES`, a space is inserted wherever a parenthesis would appear between joined expressions.)

NA Values

JOINBYTES ignores any arguments that have a value of `NA`.

Maximum Length of Joined Line

The maximum length of a joined line is 4,000 bytes. When the length of the joined line exceeds 4,000, JOINBYTES automatically breaks the line and puts the remaining bytes on the next line. The line break could occur between the two bytes of a double-byte character. JOINBYTES would then end one line with the first byte of the double-byte character and start the next line with the second byte of the character.

Line Breaks

JOINBYTES removes line breaks from the text it joins. To preserve the breaks in a multiline text expression, use the INSCHARS function.

Single-Byte Characters

When you are using a single-byte character set, you can use the JOINCHARS function instead of the JOINBYTES function.

NTEXT Data Type

This function does not accept NTEXT arguments, because it is oriented toward byte-manipulation instead of character manipulation. It always returns values of type TEXT. When you must use this function on NTEXT values, use the CONVERT or TO_CHAR function to convert the NTEXT value to TEXT.

Examples

Example 15–25 Using JOINBYTES to Concatenate Values

This example shows how you can use JOINBYTES to combine text with the current values of the two variables `name.product` and `price`. The variable `price` has a data type of DECIMAL; however, JOINBYTES automatically converts its value to TEXT in order to join it with the other text values.

```
LIMIT product TO 'Canoes'
LIMIT month TO 'Dec96'
```

The JOINBYTES function

```
JOINBYTES('Current Price for ' name.product ' is:  $' price)
```

returns the following value.

```
Current Price for Aluminum Canoes is:  $200.03
```

JOINCHARS

The JOINCHARS function joins two or more text values as a single line.

Return Value

TEXT or NTEXT

This function accepts TEXT values and NTEXT values as arguments. The data type of the return value depends on the data type of the values specified for the arguments:

- When all arguments are TEXT values, the return value is TEXT.
- When all arguments are NTEXT values, the return value is NTEXT.
- When the arguments include both TEXT and NTEXT values, the function converts all TEXT values to NTEXT before performing the function operation, and the return value is NTEXT.

Syntax

JOINCHARS(*first-expression*, *next-expression*...)

Arguments

first-expression
An expression to which JOINCHARS joins *next-expression*. When the *first-expression* has a data type other than TEXT or NTEXT, JOINCHARS converts it to TEXT. See "Converting Expressions That Are not TEXT or NTEXT" on page 15-58.

next-expression...
One or more expressions to join with *first-expression*. When an expression you want to concatenate has a data type other than TEXT or NTEXT, JOINCHARS converts it to TEXT. See "Converting Expressions That Are not TEXT or NTEXT" on page 15-58.

Notes

Converting Expressions That Are not TEXT or NTEXT

When the data type of an expression is *not* TEXT or NTEXT, JOINCHARS automatically converts its value to TEXT before concatenating it with the other values. For example, when you put a number in a JOINCHARS function, the number is automatically converted to TEXT and no extra step is needed to accomplish this. The format of the result depends on the settings of the COMMAS, DECIMALS, and PARENS options. (When PARENS is set to `YES`, a space is inserted wherever a parenthesis would appear between joined expressions.)

NA Values

JOINCHARS ignores any arguments that have a value of `NA`.

Maximum Length of Joined Line

The maximum length of a joined line is 4,000 bytes. When the length of the joined line exceeds 4,000 bytes, JOINCHARS automatically breaks the line and puts the remaining characters on the next line. When the line break would occur between the two bytes of a double-byte character, JOINCHARS does not split the double-byte character. It puts both bytes of the double-byte character on the next line.

Line Breaks

JOINCHARS removes line breaks from the text it joins. To preserve the breaks in a multiline text expression, use the INSCHARS function.

multibyte Characters

When you are using a multibyte character set, you can use the JOINBYTES function instead of the JOINCHARS function.

Examples

Example 15–26   Using JOINCHARS to Concatonate Values

This example shows how you can use JOINCHARS to combine text with the current values of the two variables `name.product` and `price`. The variable `price` has a data type of DECIMAL; however, JOINCHARS automatically converts its value to TEXT in order to join it with the other text values.

```
LIMIT product TO 'Canoes'
LIMIT month TO 'Dec96'
```

The JOINCHARS function

```
JOINCHARS('Current Price for ' name.product ' is:  $' price)
```

returns the following value.

```
Current Price for Aluminum Canoes is:  $200.03
```

JOINCOLS

The JOINCOLS function joins the corresponding lines of two or more multiline text values. The function returns a multiline text value composed of the concatenated lines.

Return Value

TEXT or NTEXT

This function accepts TEXT values and NTEXT values as arguments. The data type of the return value depends on the data type of the values specified for the arguments:

- When all arguments are TEXT values, the return value is TEXT.
- When all arguments are NTEXT values, the return value is NTEXT.
- When the arguments include both TEXT and NTEXT values, the function converts all TEXT values to NTEXT before performing the function operation, and the return value is NTEXT.

Syntax

JOINCOLS(*first-expression*, *next-expression*...)

Arguments

first-expression

An expression whose lines JOINCOLS joins with those of *next-expression*. When the expression has a data type other than TEXT or NTEXT, JOINCOLS converts it to TEXT. See "Converting Expressions That Are not TEXT or NTEXT" on page 15-58.

next-expression...

One or more expressions to join with *first-expression*. When an expression you want to concatenate has a data type other than TEXT or NTEXT, JOINCOLS converts it to TEXT. See "Converting Expressions That Are not TEXT or NTEXT" on page 15-58.

Notes

Converting Expressions That Are Not TEXT or N TEXT

When the data type of an expression is *not* TEXT or NTEXT, JOINCOLS automatically converts its value to TEXT before concatenating it with the other values. So when you put a number in a JOINCOLS function, the number is automatically converted to TEXT. The format of the result depends on the settings of the COMMAS and DECIMALS options.

NA Values

JOINCOLS ignores any arguments that have a value of `NA`.

Number of Lines Returned

The number of lines in the return value is always the same as that in the argument expression that has the most lines. When a given argument expression has fewer lines, JOINCOLS repeats its last line in each subsequent line of the return value. This repeating feature is useful when an argument expression is a single-line separator, such as a space or hyphen. See Example 15–27, "Joining the Columns of Two Text Expressions" on page 15-62.

Maximum Length of Joined Line

A single concatenated line cannot exceed 498 bytes.

Examples

Example 15–27 Joining the Columns of Two Text Expressions

In the following example, each line in `citylist` is joined with a quoted text value, and the corresponding line from `cityreps`.

`citylist` has the following values.

```
Boston
Houston
Chicago
Denver
```

`cityrep` has the following values.

```
Brady
Lopez
Alfonso
Cody
```

The JOINCOLS function

```
JOINCOLS(citylist ' -- ' cityreps)
```

returns the following.

```
Boston -- Brady
Houston -- Lopez
Chicago -- Alfonso
Denver -- Cody
```

JOINLINES

The JOINLINES function joins the values of two or more text expressions into a single multiline value. When multiline text values are joined, all the lines of the first expression appear first, followed by all the lines of the second expression, and so forth. Normally the arguments for JOINLINES are text values, but they can have other data types.

Return Value

TEXT or NTEXT

This function accepts TEXT values and NTEXT values as arguments. The data type of the return value depends on the data type of the values specified for the arguments:

- When all arguments are TEXT values, the return value is TEXT.
- When all arguments are NTEXT values, the return value is NTEXT.
- When the arguments include both TEXT and NTEXT values, the function converts all TEXT values to NTEXT before performing the function operation, and the return value is NTEXT.

Syntax

JOINLINES(*first-expression next-expression*...)

Arguments

first-expression
An expression to which JOINLINES adds *next-expression*. When the expression has a data type other than TEXT or NTEXT, JOINLINES converts it to TEXT. See "Converting Expressions That Are not TEXT or NTEXT" on page 15-58.

next-expression...
One or more expressions to join with *first-expression*. When an expression you want to concatenate has a data type other than TEXT, JOINLINES converts it to TEXT. See "Converting Expressions That Are not TEXT or NTEXT" on page 15-58.

Notes

Converting Expressions That Are Not TEXT or NTEXT

When the data type of an expression is *not* TEXT or NTEXT, JOINLINES automatically converts its value to TEXT before concatenating it with the other values. So when you put a number in a JOINLINES function, the number is automatically converted to TEXT and no extra step is need to accomplish this. The format of the result depends on the settings of the COMMAS and DECIMALS options.

NA Values

JOINLINES ignores any arguments that have a value of `NA`.

Examples

Example 15–28   Joining the Lines of Two Text Expressions

This example shows how to make a new list by adding the value `Regions` to the end of a variable called `mktglist`.

`mktglist` has the following initial values.

```
Salespeople
Products
Services
```

The statement

```
newlist = JOINLINES(mktglist 'Regions')
```

assigns the following to `newlist`.

```
Salespeople
Products
Services
Regions
```

KEY

The KEY function returns the value of the specified base dimension for a value of a conjoint dimension or a composite.

Return Value

The return value depends on the data type of the specified base dimension.

Syntax

KEY(*dimension-exp*, *base-dimension-exp*)

Arguments

dimension-exp
An expression that specifies a value of a conjoint dimension or a composite. When you specify the conjoint dimension itself, KEY uses the first value in status. When you specify the composite itself, KEY uses the first value in status for every base dimension in the composite.

base-dimension-exp
An expression that specifies the name of a base dimension of the previously specified conjoint dimension or composite for which you want to know the dimension value.

Examples

Example 15–29 Reporting with a Conjoint

Suppose you want to produce a report of data dimensioned by a conjoint dimension. You can label each row with the base values of each conjoint dimension value with the KEY function. Each base value occupies its own column and you have more control over the layout.

The following program excerpt loops over the conjoint dimension `proddist`, whose values are a combination of `product` and `district`. Assume also that there is a variable named `dsales` which is dimensioned by `proddist`.

```
DEFINE proddist DIMENSION <product district>
LD Conjoint dimension made up of combinations of product and district values
DEFINE dsales VARIABLE DECIMAL <month proddist>
LD Sparse sales data made dense by dimensioning by conjoint dimension proddist
```

The program excerpt shows `dsales` for three months. The base values of the conjoint dimension value each occupy their own column. For contrast, the second loop uses the conjoint dimension directly, without the KEY function. The conjoint dimension values are displayed in one column, with angle brackets.

```
LIMIT month TO FIRST 3
FOR proddist
  ROW KEY(proddist district) KEY( proddist product) ACROSS month: dsales
BLANK 2
FOR proddist
  ROW W 25 proddist ACROSS month: dsales
```

The program produces the following report.

```
Boston         Tents       32,153.52  32,536.30  43,062.75
Denver         Canoes      45,467.80  51,737.01  58,437.11
Atlanta        Sportswear 114,446.26 123,164.92 138,601.64
<Tents, Boston>            32,153.52  32,536.30  43,062.75
<Canoes, Denver>           45,467.80  51,737.01  58,437.11
<Sportswear, Atlanta>     114,446.26 123,164.92 138,601.64
```

LAG

The LAG function returns the values of a dimensioned variable or expression at a specified offset of a dimension prior to the current value of that dimension. Typically, you use the LAG function to retrieve values for a previous time period.

See also LAGABSPCT, LAGDIF, LAGPCT, and LEAD.

Return Value

The data type of the *variable* argument or `NA` when you try to lag prior to the first period of a time dimension.

Syntax

LAG(*tariable n*, *dimension*, [STATUS|NOSTATUS|*limit-clause*])

Arguments

variable

A variable or expression that is dimensioned by *dimension.*

n

The offset (that is, the number of dimension values) to lag. LAG uses this value to determine the number of values that LAG should go back in dimension to retrieve the value of variable. (See "Negative n Value" on page 15-69.) To count the values, LAG uses the default status, unless you use the STATUS keyword or the *limit-clause* argument to specify a different dimension status list.

dimension

The dimension along which the lag occurs. While this can be any dimension, it is typically a hierarchical time dimension of type TEXT that is limited to a single level (for example, the month or year level) or a dimension with a type of DAY, WEEK, MONTH, QUARTER, or YEAR.

When *variable* has a dimension with a type of DAY, WEEK, MONTH, QUARTER, or YEAR and you want LAG to use that dimension, you can omit the *dimension* argument.

STATUS
Specifies that LAG should use the current status list (that is, only the dimension values currently in status in their current status order) when computing the lag.

NOSTATUS
Specifies that LAG should use the default status (that is, a list all the dimension values in their original order) when computing the lag.

limit-clause
Specifies that LAG should use the default status limited by *limit-clause* when computing the lag. You can use any valid LIMIT clause (see the entry for the LIMIT command for further information). To specify that LAG should use the current status limited by *limit-clause* when computing the lag, specify a LIMIT function for *limit-clause.*

Notes

Negative *n* Value
Normally, *n* is a positive integer that indicates the number of time periods (or dimension values) before the current one. When you specify a negative value for *n*, it indicates the number of time periods after the current one. In effect, using a negative value for *n* turns LAG into a LEAD function.

Assigning Results to *time-series*
Use care when assigning the results of LAG back into the *time-series* variable. Results are assigned one cell at a time, so you can overwrite the whole array with the first value returned, instead of moving all the values over *n* positions. You can, however, use LAG to calculate a series of values based on the initial value.

Examples

Example 15–30 Using LAG

Assume that you have a `sales` variable that is dimensioned by three dimensions of the TEXT type (named `product`, `district`, and `time`). The `time` dimension is a hierarchical dimension with the following values.

```
1999
2000
Jan1999
Feb1999
...
Dec1999
Jan2000
Feb2000
...
Dec2000
```

Also, assume that there is a dimension named `timelevels` that has as values the names of the levels of the `time` dimension (that is, `Month` and `Year`) and a relation named `timelevelrel` that is dimensioned by `time` and that has values from `timelevels` (that is, the related dimension of `timelevelrel` is `timelevels`). A report of `timelevelrel` shows these relationships.

```
TIME           TIMELEVELREL
-------------- ------------
1999           Year
2000           Year
Jan1999        Month
Feb1999        Month
...            ...
Dec1999        Month
Jan2000        Month
Feb2000        Month
...            ...
Dec2000        Month
```

Suppose you want to compare racquet sales in Dallas for the first two months of 2000 with sales for the corresponding months of 1999. You can use the LAG function to produce the values from 1999 in the same report with the 2000 values. The following statements

```
LIMIT product TO 'racquets'
LIMIT district TO 'Dallas'
LIMIT time TO 'Jan2000' 'Feb2000'-
REPORT DOWN time sales HEADING 'Last Year' LAG(sales, 12, time, -
     LEVELREL timelevelrel)
```

produce this report.

```
DISTRICT: DALLAS
               -------PRODUCT-------
               ------RACQUETS-------
TIME           SALES      Last Year
-------------- ---------- ----------
Jan2000        125,879.86 118,686.75
Feb2000        150,833.64 142,305.99
```

LAGABSPCT

The LAGABSPCT function returns the percentage difference between the value of a dimensioned variable or expression at a specified offset of a dimension prior to the current value of that dimension and the current value of the dimensioned variable or expression.

Return Value

DECIMAL or `NA` when you try to lag prior to the first period of a time dimension.

Syntax

LAGABSPCT(*variable*, *n*, *dimension*, [STATUS|<u>NOSTATUS</u>|*limit-clause*])

Arguments

time-series
A variable or expression that is dimensioned by *dimension.*

n
The offset (that is, the number of dimension values) to lag. LAGABSPCT uses this value to determine the number of values that LAGABSPCT should go back in dimension to retrieve the value of variable. (See "Negative n Value" on page 15-73.) To count the values, LAGABSPCT uses the default status, unless you use the STATUS keyword or the *limit-clause* argument to specify a different dimension status list.

dimension
The dimension along which the lag occurs. While this can be any dimension, it is typically a hierarchical time dimension of type TEXT that is limited to a single level (for example, the month or year level) or a dimension with a type of DAY, WEEK, MONTH, QUARTER, or YEAR.

When *variable* has a dimension with a type of DAY, WEEK, MONTH, QUARTER, or YEAR and you want LAGABSPCT to use that dimension, you can omit the *dimension* argument.

STATUS
Specifies that LAGABSPCT should use the current status list (that is, only the dimension values currently in status in their current status order) when computing the lag.

NOSTATUS
Specifies that LAGABSPCT should use the default status (that is, a list all the dimension values in their original order) when computing the lag.

limit-clause
Specifies that LAGABSPCT should use the default status limited by *limit-clause* when computing the lag. You can use any valid LIMIT clause (see the entry for the LIMIT command for further information). To specify that LAGABSPCT should use the current status limited by limit-clause when computing the lag, specify a LIMIT function for *limit-clause*.

Notes

Formula Used

To obtain its results, LAGABSPCT uses the following formula.

```
(currentvalue - previousvalue) / ABS(previousvalue)
```

DIVIDEBYZERO Option

When the previous value of the time series used by LAGABSPCT is zero, the result LAGABSPCT returns is determined by the DIVIDEBYZERO option. When DIVIDEBYZERO is set to `NO`, an error occurs. When DIVIDEBYZERO is set to `YES`, LAGABSPCT returns `NA`.

Percentage Points

LAGABSPCT returns a decimal value that corresponds to a percent difference. To represent this value as percentage points, you can multiply it by 100. See "Using LAGDIF and LAGABSPCT" on page 15-74.

Negative *n* Value

Normally, *n* is a positive integer that indicates the number of time periods (or dimension values) before the current one. When you specify a negative value for *n*, it indicates the number of time periods after the current one. In this case, LAGABSPCT compares the current value of the time series with a subsequent value.

NASKIP2 Is Ignored

LAGABSPCT ignores NASKIP2. NASKIP2 does not control how `NA` values are treated in OLAP DML functions. It only controls arithmetic operations (involving the `+` (plus) and `-` (minus) operators) that are executed at the command line and in programs, models, and formulas.

Absolute Value

Unlike the LAGPCT function, which always uses the sign of the previous period value in calculating the result, LAGABSPCT uses the absolute value of the previous period value and therefore provides the direction of the percentage difference.

Related Functions

See also LAG, LAGDIF, and LAGPCT.

Examples

Example 15–31 Using LAGDIF and LAGABSPCT

Suppose you have a variable called `sales` that is dimensioned by a hierarchical dimension named `time`, and dimensions called `district` and `products`. Assume also that there is a dimension named `timelevels` that contains the names of the levels of the `time` dimension (that is, `Month` and `Year`) and a relation named `timelevelrel` that is dimensioned by `time` and that has values from `timelevels` (that is, the related dimension of `timelevelrel` is `timelevels`).

You want to compare sales for racquets in Dallas for the January, 2000 and the previous year. You can use theLAG function to display sales from the previous years. You can use the LAGABSPCT function to calculate the percentage difference between the two months and indicate the direction of the change. For example, when sales increase, the percentage difference LAGABSPCT returns is positive. When sales decrease, the percentage difference LAGABSPCT returns is negative.

You can also use the LAGPCT function to calculate the percentage difference between two years. You can multiply the values returned by LAGABSPCT by 100 to display them as percentage points.

The following statements

```
ALLSTAT
LIMIT product TO 'Racquets'
LIMIT district TO 'Dallas'
LIMIT time TO 'Jan2000'
REPORT DOWN time sales -
HEADING 'Last Jan' LAG(sales, 12, time, time LEVELREL timelevelrel)-
HEADING 'Lagdif' LAGDIF(sales, 12, time, time LEVELREL timelevelrel)-
HEADING 'Lagabspct' rset '%' d 0 LAGABSPCT(sales, 12, time, -
                    time LEVELREL timelevelrel) * 100
```

produce this report.

```
DISTRICT: Dallas
               ------------------PRODUCT------------------
               -----------------Racquets------------------
TIME           SALES      Last Jan   Lagdif     Lagabspct
-------------- ---------- ---------- ---------- ----------
Jan2000        125,879.86 118,686.75 7,193.11   6%
```

LAGDIF

The LAGDIF function returns the difference between the value of a dimensioned variable or expression at a specified offset of a dimension prior to the current value of that dimension and the current value of the dimensioned variable or expression.

Return Value

DECIMAL or `NA` when you try to lag prior to the first period of a time dimension.

Syntax

LAGDIF(*variable*, *n*, *dimension*, [STATUS|NOSTATUS|*limit-clause*])

Arguments

variable
A variable or expression that is dimensioned by *dimension.*

n
The offset (that is, the number of dimension values) to lag. LAGDIF uses this value to determine the number of values that LAGDIF should go back in dimension to retrieve the value of variable. (See "Negative n Value" on page 15-77.) To count the values, LAGDIF uses the default status, unless you use the STATUS keyword or the *limit-clause* argument to specify a different dimension status list.

dimension
The dimension along which the lag occurs. While this can be any dimension, it is typically a hierarchical time dimension of type TEXT that is limited to a single level (for example, the month or year level) or a dimension with a type of DAY, WEEK, MONTH, QUARTER, or YEAR.

When variable has a dimension with a type of DAY, WEEK, MONTH, QUARTER, or YEAR and you want LAGDIF to use that dimension, you can omit the dimension argument.

STATUS
Specifies that LAGDIF should use the current status list (that is, only the dimension values currently in status in their current status order) when computing the lag.

NOSTATUS
Specifies that LAGDIF should use the default status (that is, a list all the dimension values in their original order) when computing the lag.

limit-clause
Specifies that LAGDIF should use the default status limited by *limit-clause* when computing the lag. You can use any valid LIMIT clause (see the entry for the LIMIT command for further information). To specify that LAGDIF should use the current status limited by *limit-clause* when computing the lag, specify a LIMIT function for *limit-clause*.

Notes

NASKIP2 Is Ignored
LAGDIF ignores NASKIP2. NASKIP2 does not control how `NA` values are treated in OLAP DML functions. It only controls arithmetic operations (involving the `+` (plus) and `-` (minus) operators) that are executed at the command line and in programs, models, and formulas.

Negative *n* Value
Normally, *n* is a positive integer that indicates the number of time periods (or dimension values) before the current one. When you specify a negative value for *n*, it indicates the number of time periods after the current one. In this case, LAGDIF compares the current value of the time series with a subsequent value.

Examples

For an example of using LAGDIF, see Example 15–31, "Using LAGDIF and LAGABSPCT" on page 15-74.

LAGPCT

The LAGPCT function returns the percentage difference between the value of a dimensioned variable or expression at a specified offset of a dimension prior to the current value of that dimension and the current value of the dimensioned variable or expression.

Return Value

DECIMAL or `NA` when you try to lag prior to the first period of a dimension of a time dimension.

Syntax

LAGPCT(*variable*, *n*, [*dimension*], [STATUS|<u>NOSTATUS</u>|*limit-clause*])

Arguments

variable

A variable or expression that is dimensioned by *dimension.*

n

The offset (that is, the number of dimension values) to lag. LAGPCT uses this value to determine the number of values that LAGPCT should go back in dimension to retrieve the value of variable. (See "Negative n Value" on page 15-79.) To count the values, LAGPCT uses the default status, unless you use the STATUS keyword or the limit-clause argument to specify a different dimension status.

dimension

The dimension along which the lag occurs. While this can be any dimension, it is typically a hierarchical time dimension of type TEXT that is limited to a single level (for example, the month or year level) or a dimension with a type of DAY, WEEK, MONTH, QUARTER or YEAR.

When *variable* has a dimension with a type of DAY, WEEK, MONTH, QUARTER, or YEAR and you want LAGPCT to use that dimension, you can omit the dimension argument.

STATUS

Specifies that LAGPCT should use the current status list (that is, only the dimension values currently in status in their current status order) when computing the lag.

NOSTATUS
Specifies that LAGPCT should use the default status (that is, a list all the dimension values in their original order) when computing the lag.

limit-clause
Specifies that LAGPCT should use the default status limited by *limit-clause* when computing the lag. You can use any valid LIMIT clause (see the entry for the LIMIT command for further information). To specify that LAGPCT should use the current status limited by limit-clause when computing the lag, specify a LIMIT function for *limit-clause*.

Notes

Formula Used

To obtain its results, LAGPCT uses the following formula.

```
(currentvalue - previousvalue) / previousvalue
```

DIVIDEBYZERO Option

When the previous value of the time series used by LAGPCT is zero, the result LAGPCT returns is determined by the DIVIDEBYZERO option. When DIVIDEBYZERO is set to `NO`, an error occurs. When DIVIDEBYZERO is set to `YES`, LAGPCT returns `NA`.

Percentage Points

LAGPCT returns a decimal value that corresponds to a percent difference. To represent this value as percentage points, you can multiply it by `100`. See "Using LAGPCT" on page 15-80.

Negative *n* Value

Normally, *n* is a positive integer that indicates the number of time periods (or dimension values) before the current one. When you specify a negative value for *n*, it indicates the number of time periods after the current one. In this case, LAGPCT compares the current value of the time series with a subsequent value.

NASKIP2 Is Ignored

LAGPCT ignores NASKIP2. NASKIP2 does not control how `NA` values are treated in OLAP DML functions. It only controls arithmetic operations involving the + (plus) and - (minus) operators that are executed at the command line and in programs, models, and formulas.

Examples

Example 15–32 Using LAGPCT

Suppose you have a variable called `sales` that is dimensioned by a hierarchical dimension named `time`, and dimensions called `district` and `products`. Assume also that there is a dimension named `timelevels` that contains the names of the levels of the `time` dimension (that is, `Month` and `Year`) and a relation named `timelevelrel` that is dimensioned by `time` and that has values from `timelevels` (that is, the related dimension of `timelevelrel` is `timelevels`).

You can compare racquet sales in Dallas for 2000 with sales for 1999. You can use the LAG function to show values from 1999 in the same report with the 2000 values. You can use the LAGPCT function to calculate the percentage difference between the two. You can multiply the value LAGPCT returns by 100 and include a percent sign to display the difference as percentage points.

The following statements

```
ALLSTAT
LIMIT product TO 'Racquets'
LIMIT district TO 'Dallas'
LIMIT TIME TO '2000'
REPORT DOWN time sales HEADING 'Last Year' -
LAG(sales, 1, time, time LEVELREL timelevelrel)-
HEADING 'LAGPCT (Decimal Format)' -
LAGPCT(sales, 1, time LEVELREL timelevelrel) -
HEADING 'LAGPCT (Percent Format)' rset '%' -
LAGPCT(sales, 1, time LEVELREL timelevelrel) * 100
```

produce this report.

```
DISTRICT: Dallas
               ------------------PRODUCT------------------
               -----------------racquets------------------
                                      LAGPCT     LAGPCT
                                      (Decimal   (Percent
TIME           SALES      Last Year   Format)    Format)
-------------- ---------- ---------- ---------- ----------
2000           93,000,003 89,000,891 0.04       4.49%
```

LARGEST

The LARGEST function returns the largest value of an expression. You can use this function to compare numeric values or date values.

Return Value

The data type of the expression. It can be INTEGER, LONGINT, DECIMAL, or DATE.

Syntax

LARGEST(*expression* [[STATUS] *dimensions*])

Arguments

expression
The expression whose largest value is to be returned.

STATUS
Can be specified when one or more of the dimensions of the result of the function are not dimensions of the expression. (See the description of the *dimensions* argument.) When you specify the STATUS keyword when this is not the case, Oracle OLAP produces an error.

In cases where one or more of the dimensions of the result of the function are not dimensions of the expression, the STATUS keyword might be *required* in order for Oracle OLAP to process the function successfully, or the STATUS keyword might provide a performance enhancement. See "The STATUS Keyword" on page 15-82.

dimensions
The dimensions of the result. By default, LARGEST returns a single value. When you indicate one or more dimensions for the results, LARGEST calculates the largest value along the dimensions that are specified and returns an array of values. Each dimension must be either a dimension of *expression* or related to one of its dimensions. When it is a related dimension, you can specify the name of the relation instead of the dimension name. This enables you to choose which relation is used when there is more than one.

Notes

NA Values

LARGEST is affected by the NASKIP option. When NASKIP is set to `YES` (the default), LARGEST ignores `NA` values and returns the largest value or values that are not `NA`. When NASKIP is set to `NO`, LARGEST returns `NA` when any value of the expression is `NA`. When all the values of the expression are `NA`, LARGEST returns `NA` for either setting of NASKIP.

Calculating Over a Time Dimension

When *expression* is dimensioned by a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR, you can specify any other DAY, WEEK, MONTH, QUARTER, or YEAR dimension as a related *dimension.* Oracle OLAP uses the implicit relation between the dimensions. To control the mapping of one DAY, WEEK, MONTH, QUARTER, or YEAR dimension to another (for example, from weeks to months), you can define an explicit relation between the two dimensions and specify the name of the relation as the *dimension* argument to the LARGEST function.

For each time period in the related dimension, Oracle OLAP finds the largest data value in any source time period that ends in the target time period. This method is used regardless of which dimension has the more aggregate periods.

The STATUS Keyword

When one or more of the dimensions of the result of the function are not dimensions of the expression, Oracle OLAP creates a temporary variable to use while processing the function. When you specify the STATUS keyword, Oracle OLAP uses the current status instead of the default status of the related dimensions for calculating the size of this temporary variable.

When the size of the temporary variable for the results of the function would exceed 2 gigabytes, you *must* specify the STATUS keyword in order for Oracle OLAP to execute the function successfully. When the dimensions of the expression are limited to a few values and are physically fragmented, you can specify the STATUS keyword to improve the performance of the function.

When you use the LARGEST function with the STATUS keyword for an expression that requires going outside of the status for results (for example, with the LEAD or LAG functions or with a qualified data reference), the results outside of the status will be returned as `NA`.

Examples

Example 15–33 Finding the Largest Monthly Sales

This example uses the LARGEST function to find the largest monthly sportswear sales for each district during the first half of 1996. To see the largest sales figure for each district, specify `district` as the dimension of the results.

```
LIMIT product TO 'Sportswear'
LIMIT month TO 'Jan96' TO 'Jun96'
REPORT HEADING 'Largest Sales' LARGEST(sales district)
```

The preceding statements produce the following output.

```
                 Largest
DISTRICT          Sales
-------------- ----------
Boston          79,630.20
Atlanta        177,967.49
Chicago        112,792.78
Dallas         175,716.31
Denver          97,236.88
Seattle         60,322.88
```

LAST_DAY

The LAST_DAY function returns the last day of the month in which a particular date falls.

Return Value

DATETIME

Syntax

LAST_DAY(*datetime-expression*)

Arguments

datetime-expression
An expression that has the DATETIME data type.

Examples

Example 15–34 Calculating Remaining Days in a Month

The following statement calculates how many days remain between today's date and the end of the month.

```
SHOW JOINCHARS('Days left: ' LAST_DAY(SYSDATE) - SYSDATE)
```

When today's date is September 8, 2000, then this statement returns the following.

```
Days left: 22
```

LCOLWIDTH

The LCOLWIDTH option controls the default width of the label column in reports. For output from ROW command and HEADING, LCOLWIDTH affects the first column. For output from REPORT, LCOLWIDTH affects the first column except when the first column is a data column or part of a set of columns that represent the base dimensions of a composite or a conjoint dimension.

Data type

INTEGER

Syntax

LCOLWIDTH = *n*

Arguments

n

An integer expression that specifies the desired column width in number of characters. You can use an integer literal or an expression that returns an integer value. The default is `14`.

Notes

Label Columns in REPORT

By default, the REPORT command produces a column of dimension values that label the rows down the left side of the report. The default width of this label column is controlled by the LCOLWIDTH option. However, when the values of a composite or a conjoint dimension are shown down the report, Oracle OLAP creates a separate column for each base dimension. The default width of these base dimension columns is controlled by the COLWIDTH option, which has a default value of 10 characters.

Maximum Column Width

You can set LCOLWIDTH to any value from 1 to 4000.

Important: The maximum width of a line in a report is 4000 characters. Therefore, the combined width of all the columns of a report cannot be greater than 4000 characters

---

Overriding LCOLWIDTH

You can override the LCOLWIDTH value for the label column by using the WIDTH attribute in a HEADING, REPORT, or ROW command.

Examples

Example 15–35 Setting Default Column Widths

Suppose you want to look at unit sales for six months. Since the longest product name is 10 characters, you do not need the default width of 14 for your label column. Also, since the sales figures are not large, you do not need a width of 10 characters for your data columns. You can set LCOLWIDTH and COLWIDTH to give smaller default column widths.

```
LIMIT district TO 'Atlanta'
LIMIT month TO 'Oct95' TO 'Mar96'
LCOLWIDTH = 10
COLWIDTH = 6
REPORT ACROSS month: units
```

These statements produce the following output.

```
DISTRICT: ATLANTA
            ------------------UNITS------------------
            ------------------MONTH------------------
PRODUCT     Oct95  Nov95  Dec95  Jan96  Feb96  Mar96
---------- ------ ------ ------ ------ ------ ------
Tents         503    345    259    279    305    356
Canoes        317    282    267    281    309    386
Racquets    1,365  1,270  1,357  1,125  1,304  1,263
Sportswear  3,065  2,327  1,955  2,591  2,829  3,137
Footwear    3,445  3,247  2,831  3,089  3,282  3,475
```

LD

The LD command adds the description to the current object definition. The description consists of text that you specify to describe the object. You can assign a description to any type of definition.

> **Tip:** The current object definition is the definition of the object that has been most recently defined or considered during the current session. To make an object definition the current definition, use a CONSIDER command.

Syntax

LD [*text*]

Arguments

text
The text of the description you want to assign to the definition. When *text* is omitted, any existing description for the current definition is deleted.

Notes

Specifying the LD

You can create a multiline description by using a hyphen as a continuation character. However, you cannot create a description with an initial blank line with the LD command.

Examples

Example 15–36 Adding a Description to the Definition of a Variable

This example changes the description associated with the variable `units`. First, execute the CONSIDER command to make `units` the current definition. Then use the LD command to assign a new description. The `units` variable has the following definition.

```
DEFINE units VARIABLE INTEGER <month product district>
LD Actual Unit Shipments
```

The statements

```
CONSIDER units
ld Actual Unit Shipments for Each Division
DESCRIBE units
```

produce the following definition for `units`.

```
DEFINE units VARIABLE INTEGER <month product district>
LD Actual Unit Shipments for Each Division
```

LEAD

The LEAD function returns the values of a dimensioned variable or expression at a specified offset of a dimension subsequent to the current value of that dimension. Typically, you use the LEAD function to retrieve values for a future time period.

Return Value

The data type of the *variable* argument or `NA` when you try to retrieve a value from beyond the last period defined for the time dimension.

Syntax

LEAD(*variable*, *n*, [*time-dimension*], [[STATUS|NOSTATUS|*limit-clause*])

Arguments

variable
A variable or expression that is dimensioned by *dimension.*

n
The offset (that is, the number of dimension values) to lead. LEAD uses this value to determine the number of values that LEAD should go ahead in dimension to retrieve the value of variable. (See "Negative n Value" on page 15-90.) To count the values, LEAD uses the default status, unless you use the STATUS keyword or the limit-clause argument to specify a different dimension status.

dimension
The dimension along which the lead occurs. While this can be any dimension, it is typically a hierarchical time dimension of type TEXT that is limited to a single level (for example, the month or year level) or a dimension with a type of DAY, WEEK, MONTH, QUARTER, or YEAR.

When variable has a dimension with a type of DAY, WEEK, MONTH, QUARTER, or YEAR and you want LEAD to use that dimension, you can omit the *dimension* argument.

STATUS
Specifies that LEAD should use the current status list (that is, only the dimension values currently in status in their current status order) when computing the lead.

NOSTATUS

Specifies that LEAD should use the default status (that is, a list all the dimension values in their original order) when computing the lead.

limit-clause

Specifies that LEAD should use the default status limited by *limit-clause* when computing the lead. You can use any valid LIMIT clause (see the entry for the LIMIT command for further information). To specify that LEAD should use the current status limited by *limit-clause* when computing the lead, specify a LIMIT function for *limit-clause*.

Notes

Negative *n* Value

Normally, *n* is a positive integer that indicates the number of time periods (or dimension values) after the current one. When you specify a negative value for *n*, it indicates the number of time periods before the current one. In effect, using a negative value for *n* turns LEAD into a LAG function.

Examples

Example 15–37 Using LEAD

Assume that you have a `sales` variable that is dimensioned by three dimensions of the TEXT type (named `product`, `district`, and `time`). The `time` dimension is a hierarchical dimension with the following values.

```
1999
2000
Jan1999
Feb1999
...
Dec1999
Jan2000
Feb2000
...
Dec2000
```

Also, assume that there is a dimension named `timelevels` that contains the names of the levels of the `time` dimension (that is, `Month` and `Year`) and a relation named `timelevelrel` that is dimensioned by `time` and that has values from `timelevels` (that is, the related dimension of `timelevelrel` is `timelevels`). A report of `timelevelrel` shows these relationships.

```
TIME           TIMELEVELREL
-------------- ------------
1999           Year
2000           Year
Jan1999        Month
Feb1999        Month
...            ...
Dec1999        Month
Jan2000        Month
Feb2000        Month
...            ...
Dec2000        Month
```

Suppose you want to compare racquet sales in Dallas for the first two months of 1999 with sales for the corresponding months of 2000. You can use the LEAD function to produce the values from 2000 in the same report with the 1999 values. The following statements

```
LIMIT product TO 'Racquets'
LIMIT district TO 'Dallas'
LIMIT time TO 'JAN1999' 'FEB1999'
REPORT DOWN time sales HEADING 'Following Year' LEAD(sales, 12, time, time
LEVELREL timelevelrel)
```

produce this report.

```
DISTRICT: DALLAS
               -------PRODUCT-------
               ------RACQUETS-------
TIME           SALES      Following Year
-------------- ---------- ---------------------
Jan2000        118,686.75 125,879.86
Feb2000        142,305.99 150,833.64
```

LEAST

The LEAST function returns the smallest expression in a list of expressions. All expressions after the first are implicitly converted to the data type of the first expression before the comparison.

To retrieve the largest expression in a list of expressions, use GREATEST.

Return Value

The data type of the first expression.

Syntax

LEAST (*expr* [, *expr*]...)

Arguments

expr
An expression.

Examples

Example 15–38 Finding the Shortest Text Expressions

The following statement returns the shortest string.

```
SHOW LEAST('Harry','Harriot','Harold')
Harry
```

Example 15–39 Finding the Smallest Numerical Expressions

The following statement selects the number with the smallest value.

```
SHOW LEAST (5, 3, 18)
3
```

LIKECASE

The LIKECASE option controls whether the LIKE operator is case sensitive.

Data type

BOOLEAN

Syntax

LIKECASE = {YES|NO}

Arguments

YES
Specifies that the LIKE operator is case sensitive. (Default)

NO
Specifies that the LIKE operator is not case sensitive.

Notes

Newline Characters
The LIKENL option controls whether the LIKE operator recognizes newline characters.

Examples

Example 15–40 The Effect of LIKECASE

The following statements show the use of the LIKECASE option.

```
LIKECASE = YES
SHOW 'oracle' LIKE 'Oracle%'
```

The output of this SHOW command is

```
NO
```

The SHOW command

```
SHOW 'ORACLE' LIKE '%orc%'
```

produces the following output.

```
NO
```

The statements

```
LIKECASE = NO
SHOW 'ORACLE' like 'orc%'
```

produce the following output.

```
YES
```

LIKEESCAPE

The LIKEESCAPE option lets you specify an escape character for the LIKE operator.

Data type

ID

Syntax

LIKEESCAPE = *char*

Arguments

char
A text expression that specifies the character to use as an escape character in a LIKE text comparison. The default is no escape character.

The LIKE escape character affects the LISTNAMES program, which accepts a LIKE argument that it uses in a LIKE text comparison.

Notes

Using the Escape Character

The LIKE escape character lets you find text expressions that contain the LIKE operator wildcard characters, which are an underscore (_), which matches any single character, and a percent character (`%`), which matches any string of zero or more characters.

To include an underscore or percent character in a text comparison, first specify an escape character with the LIKEESCAPE option. Then, in your LIKE expression, precede the underscore or percent character with the LIKEESCAPE character you specified.

You might want to avoid using a backslash (\) as the LIKE escape character, because the backslash is the standard OLAP DML escape character. You would therefore need to have two backslashes to indicate that LIKEESCAPE should treat the second backslash as a literal character.

Examples

Example 15–41 Using an Escape Character with the LIKE Operator

This example demonstrates how to specify an escape character and how to use it with the LIKE operator.

Suppose you have a variable named `prodstat` that contains the following text values.

```
DEFINE prodstat TEXT <product>
prodstat(product 'Tents') = -
'What are the results of the fabric testing?'
prodstat(product 'Canoes') = -
'How has the flooding affected distribution?'
prodstat(product 'Racquets') = -
'The best-selling model is Whack_it!'
prodstat(product 'Sportswear') = -
'90% of the stock is ready to ship.'
prodstat(product 'Footwear') = -
'When are the new styles going to be ready?'
```

Suppose you have the following program, named `findeschar`, to find certain characters in the text contained in the cells of the `prodstat` variable. The program uses the LIKE operator.

```
ARGUMENT findstring TEXT
FOR product
   IF prodstat LIKE findstring
   THEN SHOW JOINCHARS(product ' - ' prodstat)
```

Before the program can find a text value that contains a percent character (`%`) or an underscore (_), you must specify an escape character by using the LIKEESCAPE option. Suppose you want to use a question mark (`?`) as the escape character. Before you set the escape character to a question mark, the following statement finds text that contains a question mark.

```
CALL findeschar('%?%') "Find any text that contains a question mark.
```

The preceding statement produces the following output.

```
Tents - What are the results of the fabric testing?
Canoes - How has the flooding affected distribution?
Footwear - When are the new styles going to be ready?
```

The following statements specify the question mark (?) as the escape character and then call the FINDESCHAR program.

```
LIKEESCAPE = '?'
CALL findeschar('%?%') "Find any text that ends with a percent character.
```

The preceding statement does not find any text because none of the text values in `prodstat` ends in a percent character. To find any text that contains a percent character, the following statement adds another wildcard character. LIKEESCAPE interprets the first percent character as the wildcard that matches zero or more characters, the second percent character as the literal percent character (%) because it is preceded by the question mark escape character, and the third percent character as another wildcard character. The result is that LIKEESCAPE looks for a percent character preceded by and followed by zero or more characters.

```
CALL findeschar('%?%%') "Find any text that contains a percent character.
```

The preceding statement produces the following output.

```
Sportswear - 90% of the stock is ready to ship.
```

The following statement finds text that contains an underscore.

```
CALL findeschar('%?%') "Find any text that contains an underscore.
```

The preceding statement produces the following output.

```
Racquets - The best-selling model is Whack_it!
```

The following statement doubles the escape character to find text that contains the escape character.

```
CALL findeschar('%??%') "Find any text that contains a question mark.
```

The preceding statement produces the following output.

```
Tents - What are the results of the fabric testing?
Canoes - How has the flooding affected distribution?
Footwear - When are the new styles going to be ready?
```

Example 15–42 Using an Escape Character with the LISTNAMES Program

This example demonstrates how to find the name of an object that contains a LIKE argument wildcard character. These following statements use the LIKEESCAPE option to specify an escape character, define a couple of object names that contain an underscore, and then list the dimensions whose names include an underscore.

```
LIKEESCAPE = '?'
DEFINE my_textdim DIMENSION TEXT
DEFINE my_intdim DIMENSION INTEGER
LISTNAMES DIMENSION LIKE '%?%'
```

The preceding statement produces the following output.

```
3 DIMENSIONs
----------------
MY_INTDIM
MY_TEXTDIM
_DE_LANGDIM
```

16

LIKENL to MAX

This chapter contains the following OLAP DML statements:

LIKENL

The LIKENL option controls whether the LIKE operator recognizes newline characters between lines of a text expression, when deciding whether a text value is like a text pattern. The LIKENL option applies to the text expressions on either side of the LIKE operator.

Data type

BOOLEAN

Syntax

```
LIKENL = {YES|NO}
```

Arguments

YES
Specifies that the LIKE operator recognizes newline characters between lines of a text expression. (Default)

NO
Specifies that the LIKE operator ignores newline characters between lines of a text expression. Newline characters are ignored in both of the expressions being compared.

Notes

Newline Characters

In the OLAP DML, the representation of a newline character is "`\n`".

Case Sensitivity

The setting of LIKECASE controls whether the LIKE operator is case sensitive.

Examples

Example 16–1 The Effect of LIKENL

The following statements show the use of the LIKENL option:

- The statement

  ```
  SHOW textvar
  ```

  produces the following output.

  ```
  Hello
  world
  ```

- The statements

  ```
  LIKENL = YES
  SHOW textvar LIKE '%low%'
  ```

  produce the following output.

  ```
  NO
  ```

- The statement

  ```
  SHOW 'Hello\nworld' LIKE '%\n%'
  ```

  produces the following output.

  ```
  YES
  ```

- The statement

  ```
  SHOW 'Hello\nworld' LIKE '%low%'
  ```

  produces the following output.

  ```
  NO
  ```

- The statements

  ```
  LIKENL = NO
  SHOW textvar LIKE '%low%'
  ```

  produce the following output.

  ```
  YES
  ```

- The statement

  ```
  SHOW 'Hello\nworld' LIKE '%\n%'
  ```

  produces the following output.

  ```
  YES
  ```

- The statement

  ```
  SHOW 'Hello\nworld' LIKE '%low%'
  ```

  produces the following output.

  ```
  YES
  ```

LIMIT command

The LIMIT command sets the current status list of a dimension and its dimension surrogates, or assigns values to a valueset. You use LIMIT to restrict the data values you are working on by temporarily limiting the range of the dimensions of the data. Using LIMIT, you create a current status list for a dimension. The current status list of a dimension is an ordered list of currently accessible values for the dimension. Values that are in the current status list of a dimension are said to be "in status." For more information on dimension status and its importance when working with analytic workspace data, see "Working with Subsets of Data" on page 3-30.

Syntax

LIMIT {*dimension*|*valueset*} [*concat-component*] *limit-type* [*limit-clause*] [IFNONE *label*]

where:

limit-type is one of the following keywords that specify how Oracle OLAP should modify the current status list:

TO
ADD
INSERT [FIRST|LAST|BEFORE *position*|AFTER *position*]
KEEP
REMOVE
COMPLEMENT
SORT

limit-clause specifies the values to use for the limit. There are several types of *limit-clause*:

{*inclusive_val_args....*| *exclusive_val_args*}

LEVELREL *level-relation* [*valueset2*]

related-dimension [*related-dimension-valuelist*]

[*family-keyword*] USING *parent-relation* [*inclusive_val_args*]

NOCONVERT [{*unrelated-dimension*|*numeric-valueset*}]

POSLIST *poslist-exp*

Arguments

dimension
The name of the dimension or dimension surrogate for which you are setting status.

valueset
The name of the valueset for which you are assigning values.

concat-component
Specifies the name of the component of the concat dimension whose values are used to determine the limit. When you specify a value for *concat-component*, the limit sets the status of the specified concat dimension using the values of *dimension* which is a component of the concat dimension. This *limit-clause* applies only when *dimension* is a concat dimension. The status of a concat dimension and of its component dimensions are not shared. Changing the status of a component dimension after you have used that dimension as the *limit-clause* in setting the status of a concat dimension does not change the status of the concat dimension.

TO
Replaces the status of a dimension or valueset with the values specified by the *limit-clause* arguments. The TO keyword selects values from the default status of a dimension in the same order as they appear in the LIMIT command or in the order implied by the valuelist argument. When you use arguments that imply ordering, the ordering of the values is based on their positions in the default status.

ADD
Expands the status of a dimension or valueset by adding the values specified by the *limit-clause* arguments that are not already in status. The ADD keywords selects values from the default status of a dimension in the same order as they appear in the LIMIT command or in the order implied by the *valuelist* argument. When you use arguments that imply ordering, the ordering of the values is based on their positions in the default status. ADD adds unique dimension values in the specified order at the end of the current status list or valueset list.

INSERT
Expands the status of a dimension or valueset by inserting the values specified by the *limit-clause* arguments in a specified position in the current status. The INSERT keyword selects values from the default status of a dimension in the same order as they appear in the LIMIT command or in the order implied by the *valuelist argument.*

When you use arguments that imply ordering (for example,`value1 TO value2`), the ordering of the values is based on their positions in the default status. INSERT adds values to a specified position in the current status. When an added value is already in status, it is removed from its position in the current status and added in the order in which it appears in the valuelist, preserving the order of the added values.

FIRST
Inserts the new values before the first value in status.

LAST
Inserts new values after the last value in status.

BEFORE
AFTER
Specifies whether new values Oracle OLAP inserts new values before or after position in the current status.

position
A dimension value in the current status, a character expression whose value is a dimension value in the current status, or an integer expression whose value represents the position of a dimension value in the default status.

KEEP
Reduces the status of a dimension or valueset by keeping only the values specified by the *limit-clause* arguments. Oracle OLAP performs the selection based on the current dimension status. KEEP preserves the current order of values among the values that remain in the status.

REMOVE
Reduces the status of a dimension or a valueset by removing the values specified by the *limit-clause* arguments. Oracle OLAP performs the selection based on the current dimension status. KEEP preserves the current order of values among the values that remain in the status.

COMPLEMENT
Replaces the status of a dimension or valueset with the values that are not specified by the *limit-clause* arguments. When you do not specify any arguments after COMPLEMENT, status is replaced by all values not now in status. Oracle OLAP performs the selection based on the current dimension status. COMPLEMENT leaves dimension values that remain in their default order. (Abbreviated COMP)

SORT
Sorts the values of a dimension or valueset according to the *limit-clause* arguments. LIMIT creates a temporary list of values based on the *limit-clause* arguments, and uses this list to sort the current status list. Any values not present in the temporary list are moved to the end of the current status list.

limit-clause
Specifies the values to use for the limit. There are several types of limit clauses. Because the complete syntax for each type of limit clause is complex, subsequent entries discuss the LIMIT command with each type of clause:

LIMIT command (using values)
LIMIT command (using LEVELREL)
LIMIT command (using related dimension)
LIMIT command (using parent relation)
LIMIT command (NOCONVERT)
LIMIT command (using POSLIST)

IFNONE
(For use only within an OLAP DML program) Specifies that program execution should branch to *label* when the requested status has null status or is based on a related dimension that turns out to have null status (that is, to have no values). In either case, the null status is not put into effect when program execution branches. Instead, the original status, before the LIMIT command was executed, is retained. This is true even when OKNULLSTATUS is `YES`. Within an OLAP DML program, you cannot use both IFNONE and NULL in the same command.

label
The name of a label elsewhere in the program constructed following the "Guidelines for Constructing a Label" on page 14-7. Execution of the program branches to the line directly following the specified label.

Note that *label,* as specified in IFNONE, must *not* be followed by a colon. However, the actual label elsewhere in the program must end with a colon.

Notes

Specifying a Value of a Concat Dimension
To specify a value of a nonunique concat dimension, use the following syntax.

<*base-dimension*: *value*>.

Default Status List

When you first attach an analytic workspace, the current status list of each dimension consists of all of the values of the dimension that have read permission, in the order in which the values are stored. This list of values is called the default status list for the dimension.

Unique Values

LIMIT selects only unique values of a dimension. When a value appears more than once in a LIMIT command, it is placed in status in the order of its first appearance. For example, the following lines.

```
LIMIT time TO 'Jan97', 'Feb97', 'Jan97'
STATUS time
```

produce this output.

```
The current status of TIME is:
JAN97, FEB97
```

Nonexistent Values

Oracle OLAP does not signal an error when you try to set the status of a dimension or valueset that has no values, unless you explicitly list values that do not exist. For example, assume that you have not added any values to a newly defined dimension WEEK. In this case, the statement `LIMIT week TO FIRST 10` does not cause an error. However, `LIMIT week TO 'Pete'` causes an error because `Pete` is not a value. Similarly, `LIMIT week TO 20` causes an error because `week` does not have a value at position `20`.

Empty Status

Oracle OLAP allows the status of a dimension or valueset to be set to null (empty status) only when you have explicitly specified that you want null status to be permitted. You can give this permission in either of two ways:

- Set the OKNULLSTATUS option to `YES`. This specification indicates that null status should be allowed whenever it occurs (except when the IFNONE argument is present in a LIMIT command).
- Use the NULL keyword in a LIMIT command to set the status of a particular dimension or valueset to null. You can do this by specifying TO NULL or KEEP NULL. This specification indicates that null status should be allowed for this LIMIT command only.

When you have not used either of these two methods to give permission for null status and you execute a LIMIT command that would result in null status, Oracle OLAP does not change the status to null when it executes the statement. Instead, Oracle OLAP leaves the status as it was before the statement was issued and either signals an error (when IFNONE is not present) or branches to the IFNONE label (when IFNONE is present).

An IFNONE argument indicates that you do not want program execution to take its normal course when a dimension's status were to be set to null. Therefore, when IFNONE is present, Oracle OLAP branches to the IFNONE label and does not set the status to null, even when OKNULLSTATUS is `YES`. When the NULL keyword is present together with IFNONE, Oracle OLAP signals the inconsistency with an error.

IFNONE requires the use of unstructured programming techniques. Oracle OLAP now provides alternative structured techniques, so the use of IFNONE is discouraged. IFNONE has been retained for compatibility with previous versions of Oracle OLAP.

Limiting a Conjoint

To limit a conjoint dimension to a value list, you can use the following constructions:

- Specify the actual values, surrounding each combination with angle brackets

  ```
  LIMIT proddist TO '<Tents, Boston>' -
     '<Footwear, Denver>'
  ```

- Use a variable name for the values, surrounding the combination with angle brackets.

  ```
  prodname = 'Canoes'
  distname = 'Seattle'
  LIMIT proddist To <prodname, distname>
  ```

- Create a multiline list, where each line is a combination surrounded by angle brackets.

  ```
  namelist = '<Tents Boston>\n<Footwear, -
     Denver>\n <Canoes, Seattle>'
  LIMIT proddist TO namelist
  ```

- Use the implicit relation between a conjoint dimension and its base dimension to limit the conjoint dimension. For example, use the following statement to limit PRODDIST to all conjoint values having "Canoes" as one of its base values.

  ```
  LIMIT proddist TO product 'Canoes'
  ```

  > **Note:** You can use logical position numbers for base dimension values in a conjoint dimension. "Using INSTAT When the Dimension is a Conjoint Dimension" on page 15-44 illustrates using logical position numbers

For an example of how you can limit a conjoint dimension that has a concat base dimension, see Example 16–13, "Limiting a Conjoint Dimension with a Concat Base Dimension" on page 16-28.

Limiting a Concat

You can define a concat dimension using simple dimensions, conjoint dimensions, and other concat dimensions as the base dimensions of the concat. The syntax for limiting a concat dimension to one of its values is the following.

LIMIT *concatdim* TO <*base-dim*: *value*>

For example, the concat dimension `reg.dist.ccdim` has the simple dimensions `region` and `district` as its base dimensions. The following statement sets the status of `reg.dist.ccdim` to two of its values, `region: East` and `district: Atlanta`.

```
LIMIT reg.dist.ccdim TO <region: 'East'> <district: 'Atlanta'>
```

For other methods of setting the status of a concat dimension, see Example 16–4, "Limiting a Concat Dimension" on page 16-14.

Alternative to Branching Using an IFNONE Label

As an alternative to branching to an `IFNONE` label, you can also handle null status for a dimension with the OKNULLSTATUS option. When you set OKNULLSTATUS to `YES`, then you are allowed to set the status of a dimension to null. You can then check for null status and execute appropriate commands with an IF...THEN...ELSE command, or you can handle null status as one of the cases in a SWITCH command.

```
OKNULLSTATUS = YES
LIMIT month TO sales GT salesnum
IF STATLEN(month) LT 1
   THEN GOTO showerr
```

Examples

Example 16–2 Adding and Removing Values

These lines add values to the status for the `month` dimension.

```
LIMIT month TO 'Jan96' TO 'Jun96'
LIMIT month ADD 'Jul96' 'Sep96'
```

Issuing a `STATUS month` statement produces this output.

```
The current status of MONTH is:
Jan96 TO Jul96, Sep96
```

This line removes values from the status for the `month` dimension.

```
LIMIT month REMOVE 'Jan96' TO 'Mar96'
```

Now, issuing a `STATUS month` statement produces this output

```
The current status of MONTH is:
Apr96 TO Jul96, Sep96
```

Example 16–3 Limiting with a Dimension Surrogate

A dimension and any dimension surrogates for it share the same status.

For example, assume that there is a NUMBER dimension named `store_id` that has the values `25`, `410`, `150`, `205`, `310`, and `10`. It also uses `storepos`, an INTEGER dimension surrogate for `store_id`. The dimension surrogate `storepos` has the values `1`, `2`, `3`, `4`, `5`, and `6`. A TEXT dimension surrogate for `store_id` is `storename`. It has the text values `Raoul's - Boston`, `Poldy's Potpourri`, `Molly's Emporium`, `Raoul's - Atlanta`, `Kinch's Kitchen Supplies`, and `Raoul's - Chicago`. The following statements are equivalent.

```
LIMIT store_id TO 25 410 150
LIMIT store_id TO storepos 1 2 3
LIMIT storepos TO 1 TO 3
LIMIT storepos TO first 3
LIMIT storename TO first 3
LIMIT storename TO 'Raoul\'s - Boston' TO 'Molly\'s Emporium'
LIMIT store_id TO storename storepos 1 2 3
LIMIT storename TO store_id 25 TO 150
```

The following statements set the status of the NUMBER type `store_id` dimension by limiting `storename`, which is a TEXT dimension surrogate for `store_id`, and report the values of `store_id`.

```
LIMIT storename TO 'Raoul\'s Sweets' TO 'Henry\'s Flowers'
REPORT store_id
```

The preceding statement produces the following output.

```
STORE_ID
--------------
10
20
30
```

Example 16–4 Limiting a Concat Dimension

In the following examples, the concat dimension `reg.dist.ccdim` has the simple dimensions `region` and `district` as its base dimensions. A concat dimension has an implicit relation to each of its component dimensions.

- The following statement sets the status of the concat dimension using the related dimension syntax and specifying the positions of the component (related) dimension.

  ```
  LIMIT reg.dist.ccdim TO district 1, 4, 5
  ```

  Issuing a `STATUS reg.dist.ccdim` statement produces the following output.

  ```
  The current status of REG.DIST.CCDIM is:
  <DISTRICT: BOSTON>, <DISTRICT: DALLAS>, <DISTRICT: DENVER>
  ```

- The following statement limits the concat dimension directly to the values specified by positions of the concat dimension.

  ```
  LIMIT reg.dist.ccdim TO 1, 4, 5
  ```

  Issuing a `STATUS reg.dist.ccdim` statement produces the following output.

  ```
  The current status of REG.DIST.CCDIM is:
  <REGION: EAST>, <DISTRICT: BOSTON>, <DISTRICT: ATLANTA>
  ```

- The following statements set the status of `district` and then limit `reg.dist.ccdim` to the status of `district`.

  ```
  LIMIT district TO LAST 3
  LIMIT reg.dist.ccdim TO district
  ```

  Issuing a `REPORT reg.dist.ccdim` statement produces the following output.

  ```
  REG.DIST.CCDIM
  ----------------------
  <district: Dallas>
  <district: Denver>
  <district: Seattle>
  ```

- In the following statement, the *limit-clause* argument is a list of values of the concat dimension.

  ```
  LIMIT reg.dist.ccdim TO <region: 'East'> <district:
  'Boston'> <district: 'Atlanta'>
  ```

- The following statements define a valueset for `reg.dist.ccdim`, store the current status of the concat dimension in the valueset, reset the status of the concat to `ALL`, and then limit the concat to the valueset and report the values of the concat in status.

  ```
  DEFINE regdist.vset VALUESET reg.dist.ccdim
  LIMIT regdist.vset TO reg.dist.ccdim
  LIMIT reg.dist.ccdim TO ALL
  LIMIT reg.dist.ccdim TO regdist.vset
  RPR W 22 reg.dist.ccdim
  ```

The preceding statements produce the following result.

```
REG.DIST.CCDIM
----------------------
<region: East>
<district: Boston>
<district: Atlanta>
```

You can also limit a concat dimension using a valueset of one of its component dimensions.

- When the component dimensions contain identical values, you can limit the concat dimension to those values by using a Boolean expression. When the `district` and `region` dimensions both have `New York` as a value, then the following statement limits the `reg.dist.ccdim` to those values.

  ```
  LIMIT reg.dist.ccdim TO BASEVAL(reg.dist.ccdim) EQ 'New York'
  ```

- In the following example, the concat dimension `geog` has the simple dimension `region` and the conjoint dimension `cityandstate` as its base dimensions. The following statement sets the status of the concat dimension by limiting the conjoint base dimension.

  ```
  LIMIT geog TO cityandstate <'Princeton' 'New Jersey'> -
     <'Patterson' 'New Jersey'>
  ```

  Issuing a `STATUS  geog` statement produces the following output.

  ```
  The current status of GEOG is:
  <CITYANDSTATE: <PRINCETON, NEW JERSEY>, <CITYANDSTATE: <PATTERSON, NEW
  JERSEY>>
  ```

- The following statements sets the status of the concat dimension by limiting the conjoint base dimension by specifying a value of a base dimension of the conjoint dimension.

  ```
  LIMIT geog TO cityandstate city 'Princeton'
  RPR W 30 geog
  ```

  The preceding statement produces the following output.

  ```
  GEOG
  ------------------------------
  <cityandstate: <Princeton, New Jersey>>
  <cityandstate: <Princeton, Indiana>>
  ```

Example 16–5 Limiting with a Worksheet

This example shows how to limit a dimension to the values that are contained in a column of a worksheet. Here the dimension `month` is limited to the values that are contained in the first column of the worksheet `workitem`. The following statements produce a `workitem` report, which is shown following the statements.

```
LIMIT month TO ALL
LIMIT wkscol TO 1
LIMIT wksrow TO workitem NE NA
REPORT workitem
               -WORKITEM-
               --WKSCOL--
WKSROW             1
-------------- ----------
             1 Jan96
             2 Feb96
             3 Mar96
             4 Apr96
             5 May96
             6 Jun96
             7 Jul96
             8 Aug96
             9 Sep96
            10 Oct96
            11 Nov96
            12 Dec96
```

The following statement limits the `month` dimension to the values that are listed in the first column of `workitem`.

```
LIMIT month TO CHARLIST(workitem)
```

Issuing a `STATUS month` statement produces the following output.

```
The current status of MONTH is:
Jan96 TO Dec96
```

Example 16–6 Using Ampersand Substitution with LIMIT

Assume that you want specify exactly two products for a program named `product.rpt`. In this cae, you could declare two dimension-value arguments to handle them. But when you want to be able to specify any number of products using LIMIT commands, then you can use a single argument with ampersand substitution.

Suppose you use the following commands in your program.

```
ARGUMENT natext TEXT
ARGUMENT widthamt INTEGER
ARGUMENT rptprod TEXT
    ...
LIMIT product TO &rptprod
```

You can run the program and specify that you want the first three products in the report.

```
CALL product.rpt ('Missing' 8 'first 3')
```

The single quotation marks are necessary to indicate that "first 3" should be taken as a single argument, rather than two separate arguments separated by a space. The ampersand causes the LIMIT command to interpret `'first 3'` as a keyword expression rather than as a dimension value.

Example 16–7   Branching on Null Status

Your program might try to set or refine the status of the `product` dimension to include only the products for which unit sales are greater than 500. When no products have unit sales of more than 500, then you can use the `IFNONE` keyword to specify that execution branch to the `novals` label.

```
LIMIT product KEEP units GT 500 IFNONE novals
```

In the commands following the `novals` label, you can handle the special situation in which no products have units sales greater than 500.

LIMIT command (using values)

A LIMIT command with a using values limit clause assigns valules to a valueset or sets the current status list of a dimension or dimension surrogates to:

- Specified value or values. The values can be any of the following:
  - Dimension values, expressed as literal values separated by commas, or as a multiline text expression, each line of which is a value of the dimension.
  - Ranges of dimension values, expressed as *value1* TO *value2*.
  - Integer values that represent the logical positions of dimension values, expressed as comma-delimited integers.
  - Ranges of INTEGER values that represent the logical positions of dimension values, expressed as *value1* TO *value2*.
  - Valuesets.
- Values for which a Boolean expression is `TRUE`.
- The top or bottom performers of a dimension based on a criterion
- The top or bottom performers of a dimension, by percentage, based on a criterion represented as an expression

Syntax

LIMIT {*dimension*|*valueset*} [*concat-component*] *limit-type* -

{*inclusive_val_args*....| *exclusive_val_args*} [IFNONE *label*]

where:

inclusive_val_args is one or more of the following constructs:

intvaluelist

text-expression

value1 TO *value2*

valuelist

valueset

exclusive_val_args is one of the following constructs:

ALL

boolean-expression

{BOTTOM|TOP} *n* BASEDON *expression*

{BOTTOM|TOP} *n-percent* PERCENTOF *expression*

{FIRST|LAST|NTH} *n*

LONGLIST

NULL

Arguments

dimension
The name of the dimension or dimension surrogate for which you are setting status.

valueset
The name of the valueset for which you are assigning values.

concat-component
The name of the component of the concat dimension whose values are used to determine the limit. (See the main entry for LIMIT command for complete description of this argument.)

limit-type
A keyword that specifies how Oracle OLAP should modify the current status list. (See the main entry for LIMIT command for a list and descriptions of these keywords.)

intvaluelist
A list of one or more integers, or the name of a single-cell variable that holds a numeric value. Separate the values with commas (,). Numeric values with decimal places (SHORTDECIMAL or DECIMAL values) are automatically truncated to integers before being used as dimension values. An integer specifies a dimension value by its logical position in the full set of dimension values. You cannot specify a NUMBER dimension value by an integer position. When the values of the NUMBER dimension are integers, then you can set the status of the dimension by specifying dimension values, as in *intvalue1*, *intvalue2* and so on.

text-expression
A multiline text expression, each line of which is a value of dimension.

value1* TO *value2
Specifies a range of dimension values where *value1* and *value2* can be either dimension values or integers. Such a range can be increasing (for example, 1 to 10) or decreasing (for example, 10 to 1). The status of the dimension or valueset is assigned accordingly. You cannot specify the values of a NUMBER dimension by using INTEGER positions. Instead, you can define an INTEGER dimension surrogate for the NUMBER dimension and limit the dimension by the positions of the surrogate.

valuelist
A list of one or more values of dimension. A dimension value can be specified as a text expression whose value is a valid dimension value. For a NUMBER dimension, dimension values are numbers. For dimensions with a type of DAY, WEEK, MONTH, QUARTER, or YEAR, dimension values can also be specified as DATE expressions.

valueset
An analytic workspace valueset object that is a saved list that holds the values for the dimension whose status is being set. You cannot define a valueset for a dimension surrogate, therefore you cannot specify a valueset when setting the status of a dimension surrogate. However, when you limit a dimension with a valueset, then you automatically limit to the same set any dimension surrogates of that dimension.

ALL
Specifies that all dimension values in the default status are to be included in the status. The default status is made up of all dimension values for which read permission is granted, in the same order as when the dimension was last maintained. When you start up an analytic workspace, the status for each dimension in your analytic workspace is the default status. Changing the read permission for a dimension with PERMIT or PERMITRESET commands changes the default status for the dimension.

boolean-expression
An expression whose TRUE values are used by Oracle OLAP when limiting the dimension or status. The *boolean-expression* must be dimensioned by the dimension whose status is being set. For a dimension surrogate, the Boolean expression is evaluated over the dimension for which it is a surrogate. The data types of the expressions you are comparing in the Boolean expression must be similar. See the CONVERT function for information on converting data types. To correctly use LIMIT with a Boolean expression you need to understand how it works with a Boolean expression that has with more than one dimension, see "How LIMIT

handles Boolean expressions with more than one dimension" on page 16-24 for details.

BOTTOM *n* BASEDON *expression*
TOP *n* BASEDON *expression*
Specifies that the status of a dimension or valueset is set based on a criterion, where *n* is the number of values to select and *expression* is the criterion on which to base the selection. All dimensions of *expression* other than the one whose status is being set must be limited to a single value. TOP results in the status sorted in descending order, BOTTOM results in the status sorted in ascending order. You cannot use a composite after the BASEDON keyword. When you attempt to do so, an error message will be displayed.

BOTTOM *n-percent* PERCENTOF *expression*
TOP *n-percent* PERCENTOF *expression*
Specifies that the status of a dimension or valueset is set by finding the top or bottom performers based on a criterion represented as an expression. This construction sorts values and adds them to the status that is based on their contribution, by percentage, to an expression.

For example, the following statement sorts products in descending order by each product's contribution to `TOTAL(sales)` and then add values to the status, starting from the top, until the cumulative total of `sales` by `product` reaches or exceeds 30 percent of all sales.

```
LIMIT product TO TOP 30 PERCENTOF TOTAL(sales, product)
```

Important: Do not use a criterion expression that causes a side effect or changes its own value.

FIRST *n*
LAST *n*
NTH *n*
Specifies the first *n*, last *n* or *n*th values in the dimension's full set of values when used with TO, ADD, COMPLEMENT, or INSERT. When used with KEEP or REMOVE, specifies the first *n*, last *n* or *n*th values in the current status.

Important: It can happen that the last item in status, based on a PERCENTOF criterion, is one of a number of dimension values having the same associated criterion value. In this case, LIMIT includes all dimension values with that criterion value in the resulting status, even when that causes the total of the criterion value to far exceed the specified percentage.

LONGLIST
Indicates that there can be up to 2,000 arguments in the LIMIT command. When there are less than 300 arguments, LONGLIST is not needed.

NULL
Indicates an empty dimension or valueset list. Using this keyword with the TO or KEEP arguments removes all values from the current status, leaving an empty dimension or valueset list, even when OKNULLSTATUS is `NO`. You cannot use IFNONE and NULL in the same LIMIT command. ADD, INSERT, and REMOVE NULL leave status unchanged. COMPLEMENT NULL places all values in status.

IFNONE *label*
Specifies that program execution should branch to *label* when the requested status has null status or is based on a related dimension that turns out to have null status (that is, to have no values). (See the main entry for LIMIT command for complete description of this phrase.)

Notes

Considerations When Specifying Values

Keep the following points in mind when specifying values in *limit-clause:*

- You can embed quoted strings within a quoted string, which is necessary when there are special characters in a base dimension value of a composite or conjoint dimension, such as `Joe's Deli`. See the Example 15–23, "Embedding Quoted Strings" on page 15-55 for an example of embedded quoted strings.
- When the dimension has the NTEXT data type and an argument that represents a dimension value has the TEXT data type, LIMIT converts the argument value to NTEXT. Similarly, when the dimension has the TEXT data type and an argument that represents a dimension value has the NTEXT data type, LIMIT converts the argument value to TEXT; however, in this case, the conversion can

result in data loss when the NTEXT value cannot be represented in the database character set.

- When you specify a value of a dimension with a type of DAY, WEEK, MONTH, QUARTER, or YEAR, the value can be in the format specified by the VNF (value name format) for the dimension (or in the default VNF for the type of dimension you are limiting when the dimension does not have a VNF) or in a valid input style for DATE values.

  You only need to provide the date components that are relevant for the type of dimension you are limiting. For a DAY or WEEK dimension, you must supply the day, month, and year components. For a MONTH or QUARTER dimension, you only need to supply the month and year (for example, `Jun95` or `0695` for June 1995). For a YEAR dimension, you only need to specify the year (for example, `95` for 1995). The valid input styles for dates are discussed in DATEORDER.

  When you specify a DATE expression or a text value that represents a complete date, you can specify *any* date that falls within the time period that is represented by the desired dimension value. Oracle OLAP uses the DATEORDERR option to resolve any ambiguities.

How LIMIT handles Boolean expressions with more than one dimension

In the following LIMIT command, the `sales` variable is dimensioned by three dimensions: `product`, `district`, and `month`.

```
LIMIT product TO sales GT 90000
```

The result of the previous LIMIT command is evident when the `district` and `month` dimensions are limited to a single value, as they are when you execute these statements.

```
LIMIT month TO 'Jan95'
LIMIT district TO 'Boston'
STATUS product
```

The STATUS command produces the following output.

```
The current status of PRODUCT is:
Footwear
```

In this case, the resulting status is all of the products whose sales exceed $90,000 for the month of January 1995 in the Boston district, which is only Footwear.

Consider the following example in which the MONTH dimension is not limited to a single value.

```
LIMIT product TO ALL
LIMIT month TO 'Jan95' 'Feb95' 'Mar95'
LIMIT district TO 'Boston'
```

When you execute a REPORT `sales` statement, you can see the BOSTON sales figures for three months.

```
DISTRICT: BOSTON
               -------------SALES--------------
               -------------MONTH--------------
PRODUCT          Jan95      Feb95      Mar95
-------------- ---------- ---------- ----------
Tents           32,153.52  32,536.30  43,062.75
Canoes          66,013.92  76,083.84  91,748.16
Racquets        52,420.86  56,837.88  58,838.04
Sportswear      53,194.70  58,913.40  62,797.80
Footwear        91,406.82  86,827.32 100,199.46
```

However, the following LIMIT and STATUS commands produce the output shown following them. Again, only Footwear is in the status for `month`.

```
LIMIT product TO sales GT 90000
STATUS product

The current status of PRODUCT is:
Footwear
```

In this case, each product has three sales figures, one for each month. For each product, LIMIT evaluates the sales data for *only* the first month in status. A product is added to the status when its sales data exceeds $90,000 in that month.

When you would like all months evaluated for each product, you can use the EVERY, ANY, or NONE functions. For example, the following LIMIT command adds a product to the status when *any* of its months has a sales figure that exceeds $90,000.

```
LIMIT product TO ANY(sales GT 90000, product)
```

In this case a `STATUS product` statement produces the following output.

```
The current status of product is:
Canoes, Footwear
```

Limiting Using Implicit Relations

Every dimension with a type of DAY, WEEK, MONTH, QUARTER, or YEAR is related to all other dimensions of this type through an implicit relation. When you limit the values of one DAY, WEEK, MONTH, QUARTER, or YEAR dimension by specifying another DAY, WEEK, MONTH, QUARTER, or YEAR dimension as the *related-dimension,* Oracle OLAP uses the implicit relation by default. However, when an explicit relation is defined between the two of these types of dimensions, you can override the default by specifying the name of the explicit relation as the *related-dimension.* For example, you can issue the following statement.

```
LIMIT month TO quarter year
```

This statement temporarily limits `quarter` to `year`, then limits `month` to `quarter`, and finally, restores `quarter` to its original status.

Examples

Example 16–8 Limiting with a Literal Value

This example shows how to limit the status of a dimension to one or more values (the *value1*, *value2* construction of v*aluelist*).

```
LIMIT month TO 'Jan96'
```

Example 16–9 Limiting with a Boolean Expression

You can limit a dimension or valueset according to the values of a Boolean expression. In this example, the values of the TOTALL function are broken out by `product` and compared to a constant. The LIMIT command sets the status to all the products whose sales, totaled for all months and districts, are greater than 12 million.

```
LIMIT product TO TOTAL(sales product) GT 12000000
```

Example 16–10 Limiting with a Formula

When you use the same criterion frequently to limit a dimension, you can save the expression as a formula and use the name of the formula as the limit expression.

```
DEFINE criterion FORMULA TOTAL(sales product) GT 12000000
LIMIT product TO criterion
```

Example 16–11 Limiting with a Valueset

You can save a status list in a valueset and use those values later to limit the status. When it takes several LIMIT commands to produce the status list you want, the valueset keeps you from having to repeat those LIMIT commands each time you need the same list. The following statements limit `district` to the districts in which sportswear sales exceeded $1,000,000 in 1996. The status is saved in the valueset `sports.district`, and you can limit `district` to the same list with one LIMIT command.

```
DEFINE sports.district VALUESET district
LIMIT product TO 'Sportswear'
LIMIT month TO year 'Yr96'
LIMIT sports.district TO TOTAL(sales district) GT 1000000
LIMIT district TO sports.district
```

Issuing a `STATUS district` statement produces this output.

```
The current status of DISTRICT is:
ATLANTA TO DENVER
```

Example 16–12 Limiting with a Variable

Here the TOP and BASEDON keywords are used to limit the status of a dimension, using the values of a variable as a criterion. The status list is sorted in descending order according to the values of `sales`.

```
LIMIT product TO 'Sportswear'
LIMIT month TO 'Jul96'
LIMIT district TO TOP 2 BASEDON sales
```

The following REPORT command

```
REPORT DOWN district sales
```

produces this output, which shows the results of the LIMIT commands.

```
PRODUCT: SPORTSWEAR
               --SALES---
               --MONTH---
DISTRICT         Jul96
-------------- ----------
Dallas         220,416.81
Atlanta        211,666.14
```

Example 16–13 Limiting a Conjoint Dimension with a Concat Base Dimension

In the following examples, `prod.regdist` is a conjoint dimension that has the `product` simple dimension and the `reg.dist.ccdim` concat dimension as its base dimensions. The conjoint dimension `prod.regdist` has the following values.

```
Tents   <region: East>
Tents   <region: West>
Canoes  <region: East>
Canoes  <region: West>
Tents   <district: Boston>
Tents   <district: Atlanta>
Tents   <district: Denver>
Canoes  <district: Atlanta>
Canoes  <district: Seattle>
```

You can set the status of a conjoint that has a concat dimension as a base dimension by specifying the concat dimension, one of its component dimensions, and a value of the component dimension as the LIMIT *limit-clause* as in the following statement.

```
LIMIT prod.regdist TO reg.proddist.ccdim district 'Atlanta'
RPR W 20 prod.regdist
```

The preceding statement produces the following output.

```
--------------PROD.REGDIST---------------
      PRODUCT           REG.DIST.CCDIM
-------------------- --------------------
Tents                <district: Atlanta>
Canoes               <district: Atlanta>
```

You can also set the status of the conjoint by specifying its values as in the following statement.

```
LIMIT prod.regdist TO <'Tents' '<region: East>'> <'Tents' '<district: Boston>'>
RPR W 20 prod.regdist
```

The preceding statement produces the following output.

```
--------------PROD.REGDIST---------------
      PRODUCT           REG.DIST.CCDIM
-------------------- --------------------
Tents                <region: East>
Tents                <district: Boston>
```

LIMIT command (using LEVELREL)

A LIMIT command with that uses only dimension values that are at the same level as the current level of the hierarchical dimension or dimension surrogate when setting status or assigning values to a valueset.

Syntax

LIMIT {*dimension*|*valueset*} [*concat-component*] *limit-type-*

LEVELREL *level-relation* [*valueset2*] -

[IFNONE *label*]

Arguments

dimension
The name of the dimension or dimension surrogate for which you are setting status.

valueset
The name of the valueset for which you are assigning values.

concat-component
The name of the component of the concat dimension whose values are used to determine the limit. (See the main entry for LIMIT command for complete description of this argument.)

limit-type
A keyword that specifies how Oracle OLAP should modify the current status list. (See the main entry for LIMIT command for a list and descriptions of these keywords.)

LEVELREL
Sets the status of a hierarchical dimension to all of the values of the hierarchical dimension that are at the same level as the current value of the dimension; or, that limits a hierarchical dimension to those values of the hierarchical dimension that are at the same level as the current value of the dimension and that are also in a specified valueset.

level-relation
Specifies the name of a level relation for the hierarchical dimension you want to limit. A level relation is a relation between a hierarchical dimension and another

dimension (sometimes called the level dimension) that has the names of the levels of the hierarchy as values. A level relation is dimensioned by the hierarchical dimension and has the values of the level dimension. For example, assume that there is hierarchical TEXT dimension named `time`, a level dimension for `time` named `tlevelS`, and a level relation named `time.tlevels` that is dimensioned by `time`. Assume also that the `time` dimension has a unique value for months and years and the `tlevels` dimension has two values `Month` and `Year.` In this case, for each month value (for example, `Feb 97`), the `time.tlevels` relation has a value of `Month`; and, for each year value (for example, `1997`), the `time.tlevels` relation has a value of `Year`.

valueset2
Specifies the name of a valueset object is dimensioned by the level dimension for the hierarchical dimension that you want to limit. Assume that there are the objects described in the description of the *level-relation* parameter. Additionally, assume that you have defined a valueset named `bestsalesyear` that is dimensioned by `tlevels` and, for each value, contains only the values of `time` that pertain to the year with the best sales year (for example, `1998`). In this case, for `Month`, `bestsalesyear` would have a list of all of the months in 1998 (that is, `Jan98` through `Dec98` and for `Year` would have only one value (`1998`).

IFNONE *label*
Specifies that program execution should branch to *label* when the requested status has null status or is based on a related dimension that turns out to have null status (that is, to have no values). (See the main entry for LIMIT command for complete description of this phrase.)

Examples

Example 16–14 Limiting to a Single Time Period of a Hierarchical Time Dimension

Assume that you have defined a hierarchical text dimension named `time`, a level dimension named `timelevels` that has `Month` and `Year` as values, and a relation named `timelevelsrel` that is dimensioned by `time` and that has `timelevels` as a related dimension (that is, for each value of the `time` dimension, `timelevelsre` contains a value of either `Month` or `Year`) When you wanted to limit the values of `time` that are already in status to only those values that are at the same level as `Jan99`, you can issue the following statement:

```
LIMIT time TO LEVELREL timelevelsrel
```

LIMIT command (using related dimension)

A LIMIT command with a related-dimension limit clause that uses the values of a different related dimension to assign values to a valueset or to set the status of one dimension or a dimension surrogate.

Syntax

LIMIT {*dimension*|*valueset*} [*concat-component*] *limit-type-*

related-dimension [*related-dimension-valuelist*] -

[IFNONE *label*]

Arguments

dimension
The name of the dimension or dimension surrogate for which you are setting status.

valueset
The name of the valueset for which you are assigning values.

concat-component
The name of the component of the concat dimension whose values are used to determine the limit. (See the main entry for LIMIT command for complete description of this argument.)

limit-type
A keyword that specifies how Oracle OLAP should modify the current status list. (See the main entry for LIMIT command for a list and descriptions of these keywords.)

related-dimension
Specifies the name of a relation or a dimension that is related to the dimension being limited. Using a relation name enables you to choose which relation is used when there is more than one. You can also specify as related-dimension a dimension surrogate for the dimension you are limiting or a dimension surrogate of the related dimension. For example, `dimsurr` is a dimension surrogate of `dim2` and `dim2` is related to `dim1`. The dimension surrogate `dimsurr` has the values `Dsv1`, `Dsv2`, `Dsv3` and `Dsv4`. The following statement limits `dim1` by specifying values of `dimsurr`.

```
LIMIT dim1 TO dimsurr dsv1 dsv3
```

related-dimension-valuelist
The values of the related dimension or a dimension surrogate for the related dimension or the dimension specified using the syntax shown in LIMIT command (using values). When this argument is present in a LIMIT command, status is obtained by selecting the values of the dimension being limited, which are related to the *related-dimension* values. When *valuelist* is omitted, the current status of *related-dimension* is used.

IFNONE *label*
Specifies that program execution should branch to *label* when the requested status has null status or is based on a related dimension that turns out to have null status (that is, to have no values). (See the main entry for LIMIT command for complete description of this phrase.)

Notes

Limiting to a Related Dimension Is a Two-Step Process

When you limit a dimension or valueset to a related dimension, the resulting status is determined in a two-step process:

1. The dimension values are arranged in the order of the values of the related dimension.
2. When there is more than one value of the dimension for any value of the related dimension, those values are arranged in the order of their default status.

Suppressing the Sort When Limiting to a Related Dimension

You can suppress the sort that occurs when you limit a dimension or valueset to a related dimension by setting LIMIT.SORTREL to `NO`. This can significantly improve performance when the dimension you are limiting is large.

Note: When LIMIT.SORTREL is `NO`, printed output of a dimension may not appear in logical order.

Multiple Relations in a LIMIT Command

Oracle OLAP expects values that are from the dimension being limited. When you specify a related dimension and there is more than one relation between the two dimensions, Oracle OLAP chooses the relation in which the related dimension is the dimension being limited. For example, you might have two relations between `district` and `region`, as follows.

```
DEFINE REGION.DISTRICT RELATION REGION <DISTRICT>
LD The region each district belongs to

DEFINE DISTRICT.REGION RELATION DISTRICT <REGION>
LD The primary district in each region
```

When a workspace contains both of these relations, you can no longer simply specify the following statement in order to limit `district` to one region.

```
LIMIT district TO region 'East'
```

In the preceding statement, Oracle OLAP will use the relation `district.region`, which holds one district value for each region, because it holds district values. However, the relation `region.district`, which holds region values, is the one that will produce the desired result. You must limit `region` first and then limit `district`, specifying that you want to use the first relation.

```
LIMIT region TO 'EAST'
LIMIT district TO region.district
```

Examples

Example 16–15 Limiting with a Related Dimension

Here the status of a dimension is limited using a related dimension (*rel-dim* argument). This statement limits `district` to `Boston` and `Atlanta`, which are in the `East` region.

```
LIMIT district TO region 'East'
```

This statement limits `product` to `Sportswear` and `Footwear`, which are in the division that appears last in the list of `division` values.

```
LIMIT product TO division LAST 1
```

LIMIT command (using parent relation)

A LIMIT command that uses a parent relation in its limit clause to set the status of a hierarchical dimension or its dimension surrogate, or assigns vallues to a valueset, based on family relationships within the hierarchy.

Syntax

LIMIT {*dimension*|*valueset*} [*concat-component*] *limit-type*-

  [*family-keyword*] USING *parent-relation* [*inclusive_val_args*] -

  [IFNONE *label*]

where *family-keyword* has one of the following constructs:

  PARENTS

  CHILDREN

  ANCESTORS

  DESCENDANTS

  HIERARCHY [INVERTED] [NOORIGIN] [SKIP *n*] [DEPTH *n*] [RUN *textexp*]]

Arguments

dimension
The name of the dimension or dimension surrogate for which you are setting status.

valueset
The name of the valueset for which you are assigning values.

concat-component
The name of the component of the concat dimension whose values are used to determine the limit. (See the main entry for LIMIT command for complete description of this argument.)

limit-type
A keyword that specifies how Oracle OLAP should modify the current status list. (See the main entry for LIMIT command for a list and descriptions of these keywords.)

PARENTS
Finds the parent of each value in valuelist. For a dimension, when there is no *valuelist,* finds the parent for each value in status. For a valueset, when there is no *valuelist,* it finds the parent of each value in the valueset. It uses the *parent-relation* to look up the parent.

CHILDREN
Finds the children of each value in valuelist. For a dimension, when there is no *valuelist,* finds the children for each value in status. For a valueset, when there is no *valuelist,* it finds the children of each value in the valueset. It uses the *parent-relation* to look up the children.

ANCESTORS
Finds the ancestors (that is, parents, grandparents, and so on) of each value in *valuelist.* For a dimension, when there is no *valuelist,* it finds the ancestors of each value in status. For a valueset, when there is no *valuelist,* it finds the ancestors of each value in the valueset. In other words it finds "parents" for the values and the "parents of the parents" until there are no new parents.

DESCENDANTS
Finds the descendants (that is, children, grandchildren, and so on) of each value in *valuelist.* For a dimension, when there is no *valuelist,* it finds descendants for each value in status. For a valueset, when there is no *valuelist,* it finds the descendants of each value in the valueset. In other words, it finds the children of the values and the children of the children until there are no new children.

HIERARCHY
Finds the descendants (that is, children, grandchildren, and so on) based on a particular *parent-relation.* The difference is the order of the values. DESCENDANTS groups the values by level (all children, then all grandchildren, and so on); HIERARCHY places each group of children next to its parent. HIERARCHY includes the original values (that is, those in status before the LIMIT command was executed) in status.

INVERTED
Indicates that children should be listed before their parents. By default, children are listed after their parents.

NOORIGIN
Excludes the original values from the status. The default is to include original values.

SKIP
Skips *n* generations for each value in *valuelist.* For dimensions, when there is no *valuelist,* it skips *n* generations for each value in status. For a valueset, when there is no *valuelist,* it skips *n* generations for each value in the valueset. This keyword, in combination with DEPTH, is helpful when drilling down; see Example 16–17, "Drilling Down Using SKIP and DEPT".

DEPTH
Includes *n* generations down from each value of *valuelist.* For dimensions, when there is no *valuelist,* it includes *n* generations for each value in status. For a valueset, when there is no *valuelist,* it includes *n* generations of each value in the valueset. The default depth value is 99. This keyword, in combination with SKIP, is helpful when drilling down on values.

RUN
Executes a statement, represented as a text expression, every time a group of children is constructed. For example, you can sort each group of children based on information stored in an Oracle OLAP variable. In the following statement, markets will be sorted in increasing order based on unit sales every time a group of children is constructed.

```
LIMIT market TO HIERARCHY RUN 'SORT market a unit.m' USING -
  market.market
```

Note: In this example, when you use KEEP or REMOVE instead of TO with your LIMIT command, the SORT command would have no effect.

USING
Specifies the values to use when determining parent values.

parent-relation
Specifies the name of a child-parent self-relation for the *dimension*. For each dimension value, the relation holds another value of the dimension which is its parent dimension value (the one immediately above it in a given hierarchy). This parent relation can have more than one dimension.

inclusive_val_args
Specifies the values to use when determining the parent values. You can specify any inclusive valuelist argument as described in the syntax of the *inclusive_val_args* argument for the *valuelist* clause for *limit-clause*, for a list of inclusive arguments. To

limit a dimension surrogate, use the parent relation for the dimension for which it is a surrogate.

IFNONE *label*
Specifies that program execution should branch to *label* when the requested status has null status or is based on a related dimension that turns out to have null status (that is, to have no values). (See the main entry for LIMIT command for complete description of this phrase.)

Examples

Example 16–16 A Simple Drill Down

This example drills down on districts from the region level of the market dimension. First, the market dimension, which has embedded totals at the district, region, and total U.S. level, is limited to the region level data. This is done using the relation `mlv.market`, which is a relation between `market` and `market.level`.

Issuing a `REPORT mlv.market` statement produces the following output, which shows the values of `mlv.market`.

```
MARKET         MLV.MARKET
-------------- ----------
Totus          Totus
East           Region
Boston         District
Atlanta        District
Central        Region
Chicago        District
Dallas         District
West           Region
Denver         District
Seattle        District
```

The following LIMIT command limits the values of `MARKET`, and the STATUS command produces the values currently in status. The output of STATUS is shown following the statements.

```
LIMIT market TO mlv.market 'Region'
STATUS market

The current status of MARKET is:
EAST, CENTRAL, WEST
```

To drill down on the district level data from the region level, you can use LIMIT with the CHILDREN keyword. The following example uses a parent-relation called `market.market` to perform the drill down. For each value of the `market` dimension, this relation contains the name of its parent.

```
DEFINE market.market RELATION market <market>
LD Self-relation for the Market Dimension
```

A report of `market.market` produces the following output.

```
MARKET         MARKET.MARKET
-------------- -------------
Totus          NA
East           Totus
Boston         Central
Atlanta        East
Central        Totus
Chicago        Central
Dallas         Central
West           Totus
Denver         West
Seattle        West
```

You can limit `market` to the children of the `East`, `Central`, and `West` regions by using the CHILDREN keyword with LIMIT.

```
LIMIT market TO mlv.market 'Region'
Limit market TO CHILDREN USING market.market
```

A report of `market` produces the following output.

```
MARKET
-------------
Boston
Atlanta
Chicago
Dallas
Denver
Seattle
```

Example 16–17 Drilling Down Using SKIP and DEPT

Consider the following statement.

```
LIMIT market TO HIERARCHY DEPTH 2 SKIP 1 USING market.market 'Totus'
```

Oracle OLAP will look in the child-parent relation (`market.market`) to find the children and the grandchildren (`DEPTH 2`) of `Totus` and it will discard the first generation (`SKIP 1`). The resulting status follows.

```
Totus
Boston
Atlanta
Chicago
Dallas
Denver
Seattle
```

Note that `Totus` is included in status. With HIERARCHY, the original values are included in status.

LIMIT command (NOCONVERT)

The LIMIT command sets the current status list of a dimension and its dimension surrogates, or assigns values to a valueset.

A LIMIT command with the NOCONVERT keyword sets the status of one dimension based on the numeric position of values in a different dimension.

Syntax

LIMIT {*dimension*|*valueset*} [*concat-component*] *limit-type* -

NOCONVERT [{*unrelated-dimension*|*numeric-valueset*}] -

[IFNONE *label*]

Arguments

dimension
The name of the dimension or dimension surrogate for which you are setting status.

valueset
The name of the valueset for which you are assigning values.

concat-component
The name of the component of the concat dimension whose values are used to determine the limit. (See the main entry for LIMIT command for complete description of this argument.)

limit-type
A keyword that specifies how Oracle OLAP should modify the current status list. (See the main entry for LIMIT command for a list and descriptions of these keywords.)

NOCONVERT
Sets the status of a dimension based on the numeric position of the specified values in the status list of an another dimension.

unrelated-dimension
Specifies the name of a dimension not related to the dimension being limited. Using this argument specifies that the status of a dimension or valueset is set based on the numeric position of each value in status of the unrelated-dimension. This is particularly useful when the two dimensions are in different analytic workspaces

(for example, when a one-to-one correspondence exists between the product dimension in two analytic workspaces)

numeric-valueset
Specifies the a numeric valueset. When you use this argument, NOCONVERT sets the status based on the numeric values in the valueset. The numeric values represent the positions of the values in the default status of the dimension.

IFNONE *label*
(For use only within an OLAP DML program) Specifies that program execution should branch to *label* when the requested status has null status or is based on a related dimension that turns out to have null status (that is, to have no values). (See the main entry for LIMIT command for complete description of this phrase.)

LIMIT command (using POSLIST)

The LIMIT command sets the current status list of a dimension and its dimension surrogates, or assigns values to a valueset.

A LIMIT command with the POSLIST keyword sets the status of a dimension based on the position of the values within that dimension.

Syntax

LIMIT {*dimension*|*valueset*} [*concat-component*] *limit-type* -

  POSLIST *poslist-exp* [IFNONE *label*]

Arguments

dimension
The name of the dimension or dimension surrogate for which you are setting status.

valueset
The name of the valueset for which you are assigning values.

concat-component
The name of the component of the concat dimension whose values are used to determine the limit. (See the main entry for LIMIT command for complete description of this argument.)

limit-type
One of the standard keywords (documented in the main entry for LIMIT command) that specifies how Oracle OLAP should modify the current status list.

POSLIST *poslist-textexp*
Sets the status of a dimension based on the position of a value within a dimension. *poslist-textexp* is a text expression, each line of which is a numeric value that evaluates to a numeric position of the dimension being limited.

IFNONE *label*
Specifies that program execution should branch to *label* when the requested status has null status or is based on a related dimension that turns out to have null status (that is, to have no values). (See the main entry for LIMIT command for complete description of this phrase.)

LIMIT function

The LIMIT function returns the dimension or dimension surrogate values that result from a specified LIMIT command. A dimension and any surrogate for that dimension share the same status. In this entry, references to dimensions apply equally to dimension surrogates, except where noted. The LIMIT function does not change the status of a dimension or a valueset. The LIMIT function operates on the current status.

Return Value

The return value varies depending on the use of the function and whether or not you specify the INTEGER keyword. When the LIMIT function is an argument to an OLAP DML statement (includingr a user-defined command or function) that expects a valueset, it returns a valueset. When the LIMIT function returns an empty valueset, it returns it as a valueset with null status. In all other cases, the LIMIT function returns either a TEXT value or an INTEGER value depending on whether or not you include the INTEGER keyword. When it returns a TEXT value that represents empty status, it returns it as `NA`.

Syntax

LIMIT([INTEGER] {*dimension*|*valueset*} -

{TO|ADD|INSERT|KEEP|REMOVE|COMPLEMENT} -

[*limit-clause*] [IFNONE *label*])

where:

limit-clause is one of the following:

valuelist

concat-component [valuelist]

LEVELREL *relation [valueset]*

related-dimension [*related-dimension-valuelist*]

family-phrase

NOCONVERT {*unrelated-dimension*|*valueset*}]

POSLIST *poslist-exp*

Arguments

See the LIMIT command for a complete description of all arguments other than the INTEGER keyword.

INTEGER
When you use the INTEGER keyword, the function returns the position numbers of the values in the default dimension status rather than the names. When you use INTEGER with a valueset, the function returns the position numbers of the values in the default dimension status, not in the valueset.

Notes

Nesting the LIMIT Function

Use the following syntax to return the result of several LIMIT commands for the same dimension by nesting the LIMIT function.

LIMIT (LIMIT (LIMIT (*lim-exp1*) *lim-exp2*) *lim-exp3*)

Use this nested construction to find the status of a series of LIMIT commands. For example, to see the status of the following commands

```
LIMIT product TO division 'Camping'
LIMIT product KEEP -
   EVERY(sales GT 50000, product)
LIMIT product KEEP FIRST 1
```

you execute this statement.

```
REPORT LIMIT(LIMIT(LIMIT(product TO -
   division 'Camping') KEEP EVERY -
   (sales GT 50000, product))KEEP FIRST 1)
```

Limiting with a Component of a Concat Dimension

You can limit a concat dimension to the current status of one of its component dimensions as in the following statement.

```
LIMIT(reg.dist.ccdim TO district)
```

You can also limit a concat dimension to a set of the values of one of its component dimensions as in the following statement.

```
LIMIT(reg.dist.ccdim TO district 'Boston' 'Chicago' 'Seattle')
```

Returning Multidimensional Results

The LIMIT function returns multidimensional results when evaluating multidimensional expressions. In the following example, the `sales` variable has three dimensions: `product`, `district`, and `month`.

```
LIMIT product TO ALL
LIMIT district TO 'Boston'
LIMIT month TO 'Jan95' 'Feb95' 'Mar95'
```

A `REPORT sales` statement produces the following output.

```
DISTRICT: BOSTON
          -------------SALES--------------
          -------------MONTH--------------
PRODUCT     Jan95      Feb95      Mar95
--------- ---------- ---------- ----------
Tents      32,153.52  32,536.30  43,062.75
Canoes     66,013.92  76,083.84  91,748.16
Racquets   52,420.86  56,837.88  58,838.04
Sportswear 53,194.70  58,913.40  62,797.80
Footwear   91,406.82  86,827.32 100,199.46
```

Suppose you want a list of products whose sales exceed $90,000 for the status shown in the preceding report. The LIMIT function will evaluate the product sales in each month and district combination and will produce a list that is dimensioned by the months and districts in status.

A `REPORT limit (product TO sales GT 90000)` statement produces the following output.

```
          ---LIMIT (PRODUCT TO SALES GT---
          -------------90000)-------------
          -------------MONTH--------------
DISTRICT    Jan95      Feb95      Mar95
--------- ---------- ---------- ----------
Boston    Footwear   NA         Canoes
                                Footwear
```

TEXT and NTEXT

When the dimension has the NTEXT data type and an argument that represents a dimension value has the TEXT data type, the LIMIT function converts the argument value to NTEXT. Similarly, when the dimension has the TEXT data type and an argument that represents a dimension value has the NTEXT data type, LIMIT converts the argument value to TEXT; however, in this case, the conversion can result in data loss when the NTEXT value cannot be represented in the database character set.

Examples

Example 16–18 Returning Multidimensional Results

This example prints a report of the products whose sales were greater than $50,000 in the first two months of 1995 in Boston and Atlanta. Notice that the LIMIT function returns multidimensional results.

These statements

```
LIMIT month TO 'Jan95' 'Feb95'
LIMIT district TO 'Boston' 'Atlanta'
LIMIT product TO ALL
REPORT LIMIT (product TO sales GT 50000)
```

produce this report.

```
               --LIMIT (PRODUCT TO--
               ---SALES GT 50000)---
               --------MONTH--------
DISTRICT         JAn95      Feb95
-------------- ---------- ----------
Boston         Canoes     Canoes
               Racquets   Racquets
               Sportswear Sportswear
               Footwear   Footwear
Atlanta        Racquets   Canoes
               Sportswear Racquets
               Footwear   Sportswear
                          Footwear
```

Example 16–19 LIMIT Command with the LIMIT Function

The following example shows the LIMIT function being used as an argument to the LIMIT command. The result of the LIMIT function is converted to a valueset.

```
ALLSTAT
LIMIT month TO LIMIT (LIMIT (month TO LAST 10) KEEP FIRST 3)
```

After the preceding LIMIT command, a `STATUS month` statement produces this output.

```
The current status of MONTH is:
MAR97 TO MAY97
```

LIMITMAPINFO

The LIMITMAPINFO function returns the analytic workspace expression that a specified limit map uses to map data into a specified column of a relational table.

Return Value

A TEXT expression.

Syntax

LIMITMAPINFO ([*aw*], *limit-map*, *column-name*)

Arguments

aw
The name of the analytic workspace that contains the analytic workspace object.

limit-map
The limit map as a text expression.

column-name
The name of the column of a relational table as it appears in *limit-map*.

Examples

Example 16–20 Retrieving the Name of a Dimension

Assume that you have an analytic workspace named `myaw` that contains a text variable named `mylimitmap` that is a limit map that maps some of the analytic workspace data to a relational table with a column named `et_product`.

```
MEASURE sales FROM aw_f.sales
DIMENSION et_chan FROM aw_channel WITH
HIERARCHY aw_channel.parent
GID gid_chan FROM aw_channel.gid
DIMENSION et_prod FROM aw_product WITH
HIERARCHY aw_product.parent
GID gid_prod FROM aw_prod.gid
DIMENSION et_geog FROM aw_geography WITH
HIERARCHY aw_geography.parent
GID gid_geog FROM aw_geog.gid
DIMENSION et_time FROM aw_time WITH
HIERARCHY time.parent
GID gid_time FROM aw_time.gid
```

To retrieve the name of the analytic workspace object from which data for the `et_prod` column will be retrieved, you issue the following OLAP DML statement.

```
show LIMITMAPINFO ('myaw', mylimitmap, 'et_prod')
```

The following value displays because the `et_prod` column is mapped to the `aw_product` dimension.

```
aw_product
```

LIMIT.SORTREL

The LIMIT.SORTREL option controls whether or not a sort is done when you limit a dimension to a related dimension.

Data type

BOOLEAN

Syntax

LIMIT.SORTREL = {YES|NO}

Arguments

YES
Oracle OLAP performs a sort when you limit a dimension to a related dimension.

NO
Oracle OLAP does not perform a sort when limiting to a related dimension.

Notes

The Sorting Explained

Normally, when you limit a dimension to a related dimension, the values of the dimension being limited are arranged in the order of the related dimension. When there is more than one value of the first dimension related to a value of the related dimension, the values are sorted in the order of the default status of the first dimension. It is this sort that LIMIT.SORTREL suppresses.

Output Lists when LIMIT.SORTREL Is NO

When LIMIT.SORTREL is `NO`, the output for any given dimension may not list values in logical order.

Examples

Example 16–21 Efficient Processing

You are performing calculations on a variable dimensioned by a large dimension named `product`. Your `product` dimension has all levels of the product hierarchy embedded in it: category, vendor, brand, and so on. You are performing the calculations one level at a time, using the relationship between `product` and `productlevel`. Because the order of the dimension values is not important for the calculations and because you are limiting `product` using a related dimension, you use LIMIT.SORTREL to suppress unnecessary sorting. This makes the process more efficient.

```
LIMIT.SORTREL = NO
```

LINENUM

The LINENUM option contains the current line number of the output. Its value is incremented automatically as output lines are produced. The LINENUM option is meaningful only when PAGING is set to `YES` and only for output from commands such as REPORT and LISTNAMES.

Data type

INTEGER

Syntax

LINENUM = *n*

Arguments

n

An integer expression. Normally you do not want to set LINENUM explicitly, but just want to check its current value.

Notes

Starting a New Page

When PAGING is set to `YES`, LINENUM increases by `1` after each line of output. When LINENUM equals PAGESIZE minus BMARGIN, a new page automatically begins.

At the beginning of each new page, LINENUM is automatically reset to `1`.

LINENUM Compared to PAGESIZE

Since the lines in the bottom margin are included in PAGESIZE, LINENUM can never reach PAGESIZE when BMARGIN is set to a number greater than `0` (zero).

The Effect of PAGING

When PAGING is set to `NO` (its default), the value of the LINENUM option continues to increment as more output lines are produced. When you set PAGING to `YES`, LINENUM is set to `1` and it begins counting lines on the current page.

The Effect of OUTFILE

When you use the OUTFILE command to direct output to a file, LINENUM is set to `1` for the file. When you use OUTFILE with the EOF keyword to redirect output to the default outfile, LINENUM will contain the value that it last held for the default outfile.

Sending LINENUM in Output

When you produce output that contains the value of LINENUM, and a new page is created by this output, the value of LINENUM will be recorded as `1` when your output consists of a single line. However, when the output is a multiline value, the value of LINENUM may be recorded as a value that is larger than PAGESIZE.

Related Function

When the line number you are interested in obtaining is the current record number of a file that is opened for reading, see the RECNO function.

Examples

Example 16–22 Keeping the Heading Size Constant

Suppose you have a heading that varies between one and two lines from page to page. Regardless of this variation, you want to draw a line across the page at a constant position below the heading. Include the following statement in the page heading program that you use with your report program.

```
WHILE LINENUM LT 5
BLANK
ROW W LSIZE ROW CENTER '---------------------------------'
```

LINESLEFT

(Read-only) The LINESLEFT option contains the number of lines left on the current page. The LINESLEFT option is meaningful only when PAGING is set to `YES` and only for output from commands such as REPORT and LISTNAMES.

Data type

INTEGER

Syntax

LINESLEFT

Notes

Controlling Page Breaks

LINESLEFT is used primarily in report programs to check the number of lines left on a particular page. When the number of lines left is less than that required for a part of the report that you do not want interrupted by a page break, you can then use the PAGE command to skip to a new page.

The Effect of PAGESIZE

When you change the value of PAGESIZE, the value of LINESLEFT is adjusted accordingly. First, LINESLEFT is subtracted from the old value of PAGESIZE, which gives the lines already used. This result is then subtracted from the new value of PAGESIZE which gives the new value of LINESLEFT. When LINESLEFT becomes less than 1 as a result, a new page is started at the next output line.

The Effect of PAGING

When you set PAGING to `NO`, LINESLEFT is set to the value of PAGESIZE, and it keeps this value until PAGING is set to `YES`. When you set PAGING to `YES`, LINESLEFT begins counting the lines on the current page.

The Effect of OUTFILE

When you use the OUTFILE command to direct output to a file, LINESLEFT is set to `66` for the file, to match the default value of PAGESIZE. When you set PAGESIZE to a new value for the current outfile, LINESLEFT will be adjusted accordingly. For example, assume that you direct output to a file and then set PAGESIZE to `40`. In this case, Oracle OLAP will set LINESLEFT to `40` for the file. This ensures that the first line of output to the file will trigger a new page when PAGING is set to `YES`.

When you use the OUTFILE command with the EOF keyword to redirect output to the default outfile, LINESLEFT will contain the value that it last held for the default outfile.

Sending LINESLEFT in Output

When you produce output that contains the value of LINESLEFT, the lines that contain this value are never included in the value recorded for LINESLEFT.

Examples

Example 16–23 Including a Footnote

In a report, you want a one-line footnote preceded by two blank lines at the bottom of a page. Use the following statements to generate the footnote when three lines remain on the page.

```
IF LINESLEFT EQ 3
   THEN DO
   BLANK 2
   ROW W 50 'Subject To Change Without Notice.'
   DOEND
```

LISTBY

The LISTBY program produces a report of the names of all objects in a workspace that are dimensioned by or related to one or more specified dimensions or composites. You can use LISTBY with a dimension or composite in any attached workspace.

Syntax

LISTBY *dimensions*

Arguments

dimensions
A list of one or more dimensions or composites, separated by spaces. When you list more than one dimension, all the dimensions must be in the same workspace. LISTBY returns a list of objects that are dimensioned by all the dimensions you specify. When you specify an unnamed composite, use the following format:

LISTBY SPARSE *dim1 dim2* ...

Notes

Composites and Conjoint Dimensions

The report produced by LISTBY includes any named or unnamed composite, or conjoint dimension, whose base dimension list includes the dimensions you specify.

The report also includes any object whose dimension list includes a named or unnamed composite that in turn has the specified dimensions as base dimensions.

Examples

Example 16–24 Using LISTBY

LISTBY is used here to list the name of every object that is dimensioned by or related to `product`. The statement

```
LISTBY product
```

produces the following output.

```
15 objects dimensioned by or related to PRODUCT
----------------------------------------------------------------
ADVERTISING          DIVISION.PRODUCT     EXPENSE
INDUSTRY.SALES       NAME.PRODUCT         NATIONAL.SALES
PRICE                PRODUCT.MEMO         PRODUCTSET
SALES                SALES.FORECAST       SALES.PLAN
SHARE                UNITS                UNITS.M
```

Example 16–25   Specifying More Than One Dimension

In this example LISTBY is used to list the name of every object that is dimensioned by or related to *both* `product` and `market`. The statement

```
LISTBY product market
```

produces the following output.

```
1 objects dimensioned by or related to PRODUCT, MARKET
----------------------------------------------------------
UNITS.M
```

LISTFILES

The LISTFILES command lists all the open files that can be referenced by the FILEQUERY function. This includes all files opened by FILEOPEN, OUTFILE, and LOG command.

Syntax

LISTFILES

Examples

Example 16–26 Listing Open Files

The following example shows how to use the LISTFILES command to see which open files can be referenced by the FILEQUERY function.

```
DEFINE fil.unit VARIABLE INTEGER
fil.unit = FILEOPEN('report' WRITE)
LISTFILES
```

These statements produce the following output.

```
10 w  D:\WINNT35\SYSTEM32\report
```

LISTNAMES

The LISTNAMES program produces a report that lists the names of the objects in a workspace. You can limit the list to particular types of objects, and you can have the names for each type of object listed in alphabetical order.

Syntax

LISTNAMES [AW *workspace*|'*'] [*objtype-list*|<u>ALL</u>] -

[<u>SORTED</u>|UNSORTED] [LIKE '*pattern*']

Arguments

AW *workspace*
AW '*'
Specifies the name of an attached workspace whose objects you want to list. When you omit the workspace name, LISTNAMES lists the objects in the current workspace. When you use the `'*'` (asterisk) argument instead of a workspace name, LISTNAMES produces a separate report for each attached workspace.

objtype-list
ALL
Specifies one or more of the following types of objects whose names you want to list: AGGMAP, COMPOSITE, DIMENSION, FORMULA, MODEL, OPTION, PROGRAM, RELATION, VALUESET, VARIABLE, and WORKSHEET. You can include a trailing "S" on any object type, for example, `DIMENSIONS`. You can list these object types in any order. ALL (the default) specifies that the names of objects of *all* these types should be listed.

SORTED
UNSORTED
SORTED (the default, abbreviated SORT) specifies that the object names should be sorted alphabetically. UNSORTED (abbreviated UNSORT) specifies that the object names should not be sorted alphabetically.

LIKE '*pattern*'
Compares the names of the definitions in a workspace to the text pattern you specify and lists the names that match. A definition name is *like* a text pattern when corresponding characters match. Besides literal matching, LIKE lets you use

wildcard characters to match more than one character in a string. An underscore (_) character in a pattern matches any single character. A percent (%) character in a pattern matches zero or more characters.

Examples

Example 16–27 Listing of DEMO Workspace Objects

This example lists the dimensions, variables, and relations in the current workspace. The statement

```
LISTNAMES dimension variable relation
```

produces the following output for the DEMO workspace.

```
10 DIMENSIONs       18 VARIABLEs        4 RELATIONs
----------------    ----------------    ----------------
DISTRICT            ACTUAL              DIVISION.PRODUCT
DIVISION            ADVERTISING         MARKET.MARKET
LINE                BUDGET              MLV.MARKET
MARKET              DEMOVER             REGION.DISTRICT
MARKETLEVEL         EXPENSE
MONTH               FCST
PRODUCT             INDUSTRY.SALES
QUARTER             NAME.LINE
REGION              NAME.PRODUCT
YEAR                NATIONAL.SALES
                    PRICE
                    PRODUCT.MEMO
                    SALES
                    SALES.FORECAST
                    SALES.PLAN
                    SHARE
                    UNITS
                    UNITS.M
```

LOAD

The LOAD command loads the definition of a program, formula, or model into memory. It is usually used in startup programs, to save time when a program is first used in a session.

Syntax

LOAD *object*. . .

Arguments

***object*. . .**
The name of a program, formula, or model.

Notes

Definitions Loaded on First Use

All of the objects in an analytic workspace (except for programs, formulas, and models) are loaded into memory when the analytic workspace is attached. Programs, models, and formulas are loaded into memory when first used or when requested using the LOAD command. The time required for loading is small but perceptible, and an application builder fine-tuning a system might want to preload objects in a startup program so that the application runs up to speed from the beginning of a session.

Effect of Loading Many Objects

Loading too many objects into memory can cause Oracle OLAP to run out of memory when it processes a long statement. It is best to use LOAD sparingly, choosing the objects for maximum effect.

LOAD Does Not Compile Programs

When a program is not compiled, LOAD does not automatically compile it. For best performance, you should always compile the program and save the compiled code by updating your workspace. Then when you load the program in another session (for example, with an AUTOGO program), the program will be ready to run. See COMPILE for more information about compilation.

Examples

Example 16–28 Loading Two Programs

The following statement loads the two programs `choose.months` and `sales.rpt`.

```
LOAD choose.months sales.rpt
```

Example 16–29 Loading All the Programs in a Workspace

The following statements load all the programs in the analytic workspace.

```
LIMIT NAME TO OBJ(TYPE) EQ 'program'
LOAD &VALUES(NAME)
```

LOG command

The LOG command starts or stops the recording of a session to a disk file. All lines of input and output are recorded.

Syntax

LOG [APPEND] {*file-id*|EOF|SAVE}

Arguments

APPEND
When *file-id* already exists, appends the record of your session to the end of its current contents. APPEND has no effect when the file does not already exist or when you specify EOF.

When *file-id* already exists and you omit APPEND, LOG replaces the contents of *file-id* with the new session record.

file-id
A file identifier that specifies a disk file in which to record the session. *File-id* is a text expression that represents the name of the file. The name must be in a standard format for a file identifier. When a log file is already open, specifying a new *file-id* closes the previous file. Enclose the file identifier in single quotes.

EOF
Stops recording of the session and closes any opened log record file.

SAVE
Forces Oracle OLAP to update the log file. Lines of input and output are not always written to disk as they are generated. Instead, the lines are stored temporarily then written to disk periodically. `LOG SAVE` effectively issues the `LOG EOF` and `LOG APPEND` *`file-id`* commands. This ensures that all appropriate lines are written to disk by closing the log file and reopening it. Additional lines of input and output are appended to the file.

Notes

Automatic Closing of a Log File

When you use `LOG` *`file-id`* to start recording in a disk file, LOG closes any log record file that is currently open. This happens even when the new file is not actually opened (as when you specify an invalid 'file-id' in the LOG command).

Examples

Example 16–30 Keeping a Log File

To record your session in a file called `session.log`, use a statement like the following.

```
LOG 'session.log'
```

LOG function

The LOG function computes the natural logarithm of an expression.

Return Value

DECIMAL

Syntax

LOG(*expression*)

Arguments

expression
The value of *expression* must be greater than zero. When the value is equal to or less than zero, LOG returns an NA value.

Examples

Example 16–31 Calculating a Natural Logarithm

In this example the LOG function is used to calculate the natural logarithm of the expression 4,000 + 6,000. The statements

```
DECIMALS = 5
SHOW LOG(4000 + 6000)
```

produce the following result.

```
9.21034
```

LOG10

The LOG10 function computes the logarithm base `10` of an expression.

Return Value

DECIMAL

Syntax

LOG10(*expression*)

Arguments

expression

The value of *expression* must be greater than zero. When the value is equal to or less than `0` (zero), LOG10 returns an `NA` value.

Examples

Example 16–32 Calculating a Base 10 Logarithm

This example uses the LOG10 function to calculate the base 10 logarithm of 1,000. The statement

```
SHOW LOG10(1000)
```

produces the following result.

```
3.00
```

LOWCASE

The LOWCASE function converts all alphabetic characters in a text expression into lowercase.

Return Value

TEXT or NTEXT

When the expression is TEXT, the return value is TEXT. When the expression is NTEXT, the return value is NTEXT

Syntax

LOWCASE(*text-expression*)

Arguments

text-expression
The text expression whose characters are to be converted.

Examples

Example 16–33 Converting Part of an Expression to Lowercase

Suppose you get some new data to add to a mailing list. In the existing mailing list, people's names have only the first letter capitalized. In the new data, however, the whole name is capitalized. You can use LOWCASE to make the new data correspond to the current data with a statement similar to the following.

```
lastname = JOINCHARS(EXTCHARS(lastname, 1, 1), -
           LOWCASE(EXTCHARS(lastname, 2, NUMCHARS(lastname))))
```

LPAD

The LPAD function returns an expression, left-padded to a specified length with the specified characters; or, when the expression to be padded is longer than the length specified after padding, only that portion of the expression that fits into the specified length.

To right-pad a text expression, use RPAD.

Return Value

TEXT or NTEXT based on the data type of the expression you want to pad (*text-exp*).

Syntax

LPAD (*text-exp* , *length* [, *pad-exp*])

Arguments

text-exp
A text expression that you want to pad.

length
The total length of the return value as it is displayed on your screen. In most character sets, this is also the number of characters in the return value. However, in some multibyte character sets, the display length of a character string can differ from the number of characters in the string.

When you specify a value for *length* that is shorter than the length of *text-exp*, then this function returns only that portion of the expression that fits into the specified length.

pad-exp
A text expression that specifies the padding characters. The default value of *pad-exp* is a single blank.

Examples

The following example left-pads a string with the characters "*" and ".":

```
SHOW LPAD('Page 1',15,'*.')
*.*.*.*.*Page 1
```

LSIZE

The LSIZE option defines the line size within which the STDHDR program centers the standard header. LSIZE can be set in the initialization section of a report program.

Data type

INTEGER

Syntax

LSIZE = *n*

Arguments

n
An integer expression that specifies the line size within which the STDHDR program centers the standard header, or the maximum line size for output from the HEADING command. The default is 80 characters for a line.

Notes

Centering Report Segments

Since STDHDR centers the running page heading within the width of LSIZE, you can use it in conjunction with LSIZE to center parts of your report. (Start by setting LSIZE to the width of the longest line in your report.)

Creating Centered Headings

You can use LSIZE in centering your own headings for each page or at the beginning of a section. Start by setting LSIZE to the width of your line. Then use the HEADING command with a WIDTH of LSIZE and the keyword CENTER before the text of your heading. See Example 16–34, "Centering a Heading" on page 16-70.

Maximum Line Width

The maximum width of any line in a report, including a heading line, is 4000 characters. Therefore, it generally makes sense to set LSIZE to a value of 4000 or less.

Output to the Default Outfile

When you set LSIZE for the default outfile, the new value remains in effect until you reset it, regardless of intervening OUTFILE commands that send output to a file. That is, the value of LSIZE is automatically saved for the default outfile.

Output to a File

To set LSIZE for a file, first make the file your current outfile by specifying its name in an OUTFILE command, then set LSIZE to the desired value. The new value remains in effect until you reset it or until you use an OUTFILE command to direct output to a different outfile. When you direct output to a different outfile, LSIZE returns to its default value of 80 for the file.

Examples

Example 16–34 Centering a Heading

Suppose you design a quarterly sales report to have a short line width of 50 characters so that readers have plenty of room to make notes in the margins. To center your headings, include the following lines near the beginning of your report program.

```
PAGEPRG = 'stdhdr'
LSIZE = 50
PAGING = YES
PAGE
HEADING WIDTH LSIZE CENTER 'Quarterly Sales'
```

The following output will be produced at the beginning of the report.

```
96/05/13 15:05:16                            PAGE 1

                  Quarterly Sales
```

LTRIM

The LTRIM function removes characters from the left of a text expression, with all the leftmost characters that appear in another text expression removed. The function begins scanning the base text expression from its first character and removes all characters that appear in the trim expression until reaching a character that is not in the trim expression and then returns the result.

To trailing characters, use RTRIM. To trim both leading or trailing characters, use TRIM.

Return Value

TEXT or NTEXT based on the data type of the first argument.

Syntax

LTRIM (*text-exp* [, *trim-exp*])

Arguments

text-exp
A text expression that you want trimmed.

trim-exp
A text expression that is the characters to trim. The default value of *trim-exp* is a single blank.

Examples

The following example trims all of the left-most x's and y's from a string:

```
SHOW LTRIM('xyxxxyLast Word','xy')
Last Word
```

MAINTAIN

The MAINTAIN command enters and maintains the values of dimensions, composites, and partition template objects.

Note: You can also issue a MAINTAIN statement for a surrogate dimension that has a Maintain trigger. In this case, Oracle OALP only executes the Maintain trigger program; no other action occurs. See "Trigger Programs" on page 1-14 for more information for more information. Issuing a MAINTAIN statement for a surrogate dimension that does *not* have a Maintain trigger, returns an error.

Syntax

MAINTAIN *object* {ADD|DELETE|RENAME|MOVE|MERGE} *args*

The keywords that you can use with the MAINTAIN command varies by object:

- MAINTAIN *dimension* {ADD|DELETE|RENAME|MOVE|MERGE} *args*

  The keyword that you can use varies by the type of dimension that you want to maintain:

  - With a non-concat dimension, you can use the ADD, DELETE, RENAME, MOVE, or MERGE keywords to add, delete, rename, move, or merge non-concat dimension values. You can also use the ADD keyword to add temporary calculated members to a dimension.
  - With a concat dimension, you can only use the MOVE keyword to move concat dimension values.
- MAINTAIN *composite* {ADD|DELETE|MERGE} *args*
- MAINTAIN *partition-template* {ADD|DELETE|MOVE} *args*

The specific syntax varies by keyword. Consequently, there are separate topics for each keyword of the MAINTAIN command:

MAINTAIN ADD
MAINTAIN DELETE
MAINTAIN MERGE
MAINTAIN MOVE
MAINTAIN RENAME

For information that applies to the MAINTAIN command in general, see the Notes in this topic.

Notes

Triggering Program Execution When MAINTAIN Executes

Using the TRIGGER command, you can make the MAINTAIN command an event that automatically executes an OLAP DML program. See "Trigger Programs" on page 1-14 for more information.

Automatic Status Reset

When you use the ADD, DELETE, MERGE, or MOVE keyword to maintain a dimension or composite whose status is not currently ALL, the MAINTAIN command automatically resets status to ALL before performing the maintenance function. However, when you use the RENAME keyword to maintain a dimension whose status is not currently ALL, the MAINTAIN command does *not* change the status of the dimension.

Maintain Permission

You cannot perform maintenance on a dimension when a PERMIT MAINTAIN command denies maintain permission for the dimension. Maintain permission is implicitly denied whenever read permission is restricted for a dimension, even when you specify maintain permission for the dimension. (See the PERMIT command.)

TEXT and NTEXT

When the dimension has the NTEXT data type and an argument that represents a dimension value has the TEXT data type, MAINTAIN converts the argument value to NTEXT. Similarly, when the dimension has the TEXT data type and an argument that represents a dimension value has the NTEXT data type, the LIMIT command converts the argument value to TEXT; however, in this case, the conversion can result in data loss when the NTEXT value cannot be represented in the database character set.

Maintaining Dimensions in Multiwriter Analytic Workspaces

Keep the following points in mind when maintaining dimensions in an analytic workspace that is attached in multiwriter mode:

- You cannot update a variable when any of its dimensions have been acquired and modified.
- Reverting a dimension after adding dimension values is not recommended since it can result in suboptimal space allocation for variables dimensioned by the dimension.
- When an acquired variable is dimensioned by an acquired dimension that has been maintained, you cannot update the variable until the dimension is updated or released.
- You do not need to acquire composites in order for them to be maintained, Oracle OLAP automatically performs concurrent dimension maintenance for the composite dimensions.

Maintaining Dimensions in an Analytic Workspace Attached in Multiwriter

Before you can maintain dimensions in an analytic workspace that is attached in multiwriter mode, you must first acquire the dimension using the ACQUIRE command.

For example, assume that user A and user B both need to perform what-if computations on both `actuals` and `budget`. After performing the what-if computations, user A needs to modify `actuals` and B needs to modify `budget`. Finally, both user A and user B need to add a new `time` dimension value and add data corresponding to that new dimension value to `actuals` or `budget`.

User A issues the following OLAP DML statements.

```
AW ATTACH myworkspace MULTI
...make modifications
ACQUIRE actuals
...make more modifications
ACQUIRE time
MAINTAIN time ADD 'Y2002'
actuals (time 'Y2002', ...) = ...
UPDATE MULTI actuals, time
COMMIT
RELEASE actuals, time
AW DETACH myworkspace
```

User B issues the following OLAP DML statements.

```
AW ATTACH myworkspace MULTI
...make modifications
ACQUIRE budget
...make more modifications
ACQUIRE time--> failed
ACQUIRE RESYNC time WAIT
MAINTAIN time ADD 'Y2003'
budget (time 'Y2003', ...) = ...
UPDATE MULTI budget, time
COMMIT
RELEASE budget, time
AW DETACH myworkspace
```

MAINTAIN and Dimension Surrogates

You cannot use the MAINTAIN command on a dimension surrogate. You can only use MAINTAIN to add values to or delete them from a dimension. However, when you add or delete a dimension value, then Oracle OLAP adds or removes a position from surrogates of that dimension. When you add a position to a dimension, the corresponding position in a surrogate for that dimension receives an `NA` value.

Maintaining a Concat Dimension

A concat dimension contains the values of its component dimensions. You do not directly add, merge, or delete the values of a concat dimension with the MAINTAIN command. Instead, when you add, merge, or delete values from a component dimension of the concat, Oracle OLAP automatically adds or deletes the values from the concat dimension. You can use the MOVE keyword of the MAINTAIN command to change the order of the values of a concat dimension.

MAINTAIN ADD

The MAINTAIN command with the ADD keyword adds new TEXT, ID, and INTEGER values to a non-concat dimension, composite, or partition; or adds a new temporary calculated member to a dimension.

> **Note:** You can also issue a MAINTAIN ADD for TEXT, ID, and INTEGER Values statement for a surrogate dimension that has a Maintain trigger. In this case, Oracle OALP only executes the Maintain trigger program one time for each value; no other action occurs. See "Trigger Programs" on page 1-14 for more information for more information. Issuing a MAINTAIN statement for a surrogate dimension that does *not* have a Maintain trigger, returns an error.

Syntax

The syntax for using the MAINTAIN command with the ADD keyword depends on the type of the object being maintained and whether you are adding a permanent or temporary member.

For this reason, the following separate entries are provided for MAINTAIN ADD:

- MAINTAIN ADD for TEXT, ID, and INTEGER Values
- MAINTAIN ADD for DAY, WEEK, MONTH, QUARTER, and YEAR Values
- MAINTAIN ADD SESSION
- MAINTAIN ADD TO PARTITION

MAINTAIN ADD for TEXT, ID, and INTEGER Values

The MAINTAIN command with the ADD keyword adds new TEXT, ID, or INTEGER values to a non-concat dimension or composite.

Note: You can also issue this MAINTAIN ADD statement for a surrogate dimension that has a Maintain trigger. In this case, Oracle OALP only executes the Maintain trigger program one time for each value in *valuelist*; no other action occurs. See "Trigger Programs" on page 1-14 for more information for more information. Issuing a MAINTAIN statement for a surrogate dimension that does *not* have a Maintain trigger, returns an error.

Syntax

MAINTAIN *composite*|*dimension* ADD *valuelist* [FIRST|LAST|BEFORE *position*|AFTER *position*]

Arguments

dimension

A non-concat dimension, already defined in an attached analytic workspace.

composite

A composite. When the composite is a named composite, it must be defined in an attached analytic workspace. When the composite is unnamed, it must have been used in defining an object in an attached analytic workspace. Use the SPARSE keyword to refer to an unnamed composite (for example, `SPARSE <market product>`).

ADD *valuelist*

Specifies that the values in *valuelist* are to be added to the dimension or composite:

- When you use this argument to add values to a composite or a dimension of type TEXT or ID, the *valuelist* can be text literals or a TEXT or ID expression. When it is a multiline text expression, each element (line) is treated as a separate value.Do not add null dimension values (empty single quotes) or values that consists of spaces only, because there is no way you can refer to such values in the future.
- When *dimension* is INTEGER, *valuelist* can be an integer quantity, such as 5 or 100.

FIRST
LAST

Specify the logical position at which dimension values will be added. FIRST indicates that the new values will be inserted before any existing values. LAST indicates that new values will be added at the end of the current values. LAST is the default. When you are adding a certain quantity of integers to an INTEGER dimension, that quantity of integers will be added before or at the end of any existing integers (depending on your specification), and all the integers in the resulting series will be automatically adjusted into simple numerical order.

All values specified before the keyword FIRST or LAST are placed in that position, not just the one value immediately preceding the keyword in your command.

BEFORE *position*
AFTER *position*
Specify a position before or after which the dimension values are to be added. For *position* you can specify an existing dimension value, a character expression whose value is an existing dimension value, or an integer expression whose value represents the position of a dimension value. When you are adding a certain quantity of integers to an INTEGER dimension, that quantity of integers will be added before or after the integer position you specify, and the integers in the whole of the resulting series will be automatically adjusted into simple numerical order.

All values specified before the keywords BEFORE or AFTER are placed in that position, not just the one value immediately preceding the keyword in your command.

Notes

Sequence for Integer Dimension

When you use MAINTAIN to add values in an integer dimension, the values are renumbered to keep the normal sequence of integers (1, 2, 3, ...).

Conjoint Dimensions and Composites

Each value of a conjoint dimension or composite is a combination of values from each of the dimensions (and composites, if any) in its dimension list. To add values to a conjoint dimension or composite, specify each value combination enclosed in angle brackets. The values in a given combination must be in the same order as the dimensions and composites in the definition of the conjoint dimension or composite. Each dimension value in the combination must already exist as a value in the corresponding base dimension. However, when a composite value in the combination does not exist, Oracle OLAP will automatically add the value to the appropriate composite.

Examples

Example 16–35 Adding Values to a TEXT Dimension

This statement adds `Omaha` and `Seattle` to the end of the dimension values for the `city` dimension.

```
MAINTAIN city ADD 'Omaha' 'Seattle'
```

This statement adds `Atlanta` at the beginning of the list of cities and inserts `Peoria` after `Omaha`.

```
MAINTAIN city ADD 'Atlanta' FIRST, 'Peoria' AFTER 'Omaha'
```

Here the value of the TEXT variable `textvar` is inserted before the fifth dimension value of `city`. When you assign the value `Columbus` to `textvar`, you must make sure it is in mixed case, because you want the dimension value to be in mixed case.

```
textvar = 'Columbus'
MAINTAIN city ADD textvar BEFORE 5
```

Example 16–36 Adding Values to a Conjoint Dimension

The following is an example of adding values to a conjoint dimension.

```
DEFINE proddist DIMENSION <product, district>
MAINTAIN proddist ADD <'Tents' 'Boston'> <'Footwear' 'Denver'>
```

You can also assign a value of a base dimension to a text variable and use the name of the variable inside the angle brackets.

```
prodname = 'Canoes'
distname = 'Seattle'
MAINTAIN proddist ADD <prodname, distname>
```

MAINTAIN ADD for DAY, WEEK, MONTH, QUARTER, and YEAR Values

The MAINTAIN command with the ADD keyword adds new values to a dimension of type DAY, WEEK, MONTH, QUARTER, and YEAR.

Syntax

MAINTAIN *dimension* ADD {*valuelist*|{*n* PERIODS FIRST}|{*n* PERIODS LAST}}

Arguments

dimension

A non-concat dimension, already defined in an attached analytic workspace.

ADD *valuelist*

Specifies that the values in *valuelist* are to be added to the dimension. When *dimension* is of type DAY, WEEK, MONTH, QUARTER, or YEAR, then *valuelist* can be text constants or a TEXT, ID, or DATE expression. When the values are TEXT, they can be in the format specified by the VNF (value name format) for the dimension (or in the default format for the type of dimension you are maintaining when the dimension does not have a VNF) or in a valid input style for date values. When the values are specified as a TEXT expression, each element or line is treated as a separate value.

When the values are in the format specified by the VNF or in the default format for this type of dimension, each value explicitly indicates the time period you want to add. For example, assume that the VNF for a `month` dimension is `'<MTXT><YY>'`. In this case, the value `JAN99` represents the month January 1999.

When you specify a value for a DAY, WEEK, MONTH, QUARTER, or YEAR dimension as a date, you must provide only the date components that are relevant for the type of dimension you are maintaining. For a DAY or WEEK dimension, you must supply the day, month, and year components. For a MONTH or QUARTER dimension, you must supply only the month and year (for example, `'JUN98'` or `'0698'` for June 1998). For a YEAR dimension, you must specify only the year (for example, `'98'` for 1998). For information about the valid input styles for dates, see DATEORDER.

When you add a dimension value by specifying a DATE expression or a TEXT value that represents a complete date, you can specify *any* date that falls within the time period you want to add. For example, to add the month January 1999, you can specify any date from `'01JAN99'` through `'31JAN99`'. Oracle OLAP uses the DATEORDER option to resolve any ambiguities.

When adding values to a DAY, WEEK, MONTH, QUARTER, or YEAR dimension that does not yet have values, you must specify only the first and last values you want to add for the dimension. Oracle OLAP automatically fills in the gaps with appropriate values for the intervening time periods.

When a DAY, WEEK, MONTH, QUARTER, or YEAR dimension already has values, you can add values only at the beginning or the end of the existing list. To add values, you must specify only the first or last value you want to add. Oracle OLAP automatically fills in the gap between the existing list and the value you specify.

***n* PERIODS FIRST**
***n* PERIODS LAST**
Specifies a number of periods to add at the beginning or end of an existing list of dimension values.

Examples

Example 16–37 Adding Values to Dimension of Type QUARTER

In this example you define a new QUARTER dimension, called `qtr`, and you add dimension values for the quarters in 1998 and 1999. You only need to add the first and last dimension values you want. Oracle OLAP fills in the intervening values. To add the first and last quarters, you can specify any dates that fall within those quarters.

```
DEFINE qtr DIMENSION QUARTER
MAINTAIN qtr ADD '01jan98' '31dec99'
```

MAINTAIN ADD SESSION

The MAINTAIN command with the ADD SESSION keywords adds a temporary calculated member to a dimension and applies it to the specified objects; or applies a previously-defined calculated member to the specified objects. The calculated member and it's definition do not persist from session to session; both are deleted at the end of the session in which they are created.

Syntax

MAINTAIN *dimension* ADD SESSION *member_name* [= *calculation*] -

    [STEP DIMENSION (*stepdim*...)][*apply-to*]

where:

calculation is one of the following:

model-equation

AGGREGATION (*dimension-members*....)

apply-to specifies the basis on which the custom aggregation is added using one of the following phrases:

APPLY TO AGGMAP *aggmaps*

APPLY FOR VARIABLE *variables*

```
APPLY WITH RELATION relations
```

Arguments

dimension
A dimension that is already defined in an attached analytic workspace. You can specify any type of dimension for *dimension* except a non-unique concat dimension or a base dimension of either a unique or non-unique concat dimension.

ADD SESSION
ADD SESSION indicates maintenance of a temporary calculated member.

member-name
Specifies the name of the temporary calculated member.

=
Indicates that you are defining a new calculated member.

model-equation
A text expression that specifies the calculation used as a dynamic model to calculate custom member values. (See SET for more information about model equations.)

AGGREGATION
Indicates that the temporary calculated member is added as a custom aggregation using the specified dimension members. This clause effectively modifies the RELATION statement of aggmap objects that are the aggregation specification for variables dimensioned by *dimension*. Consequently, a MAINTAIN ADD SESSION statement that contains an AGGREGATION clause must also contain an APPLY WITH RELATION clause.

***dimension-membe*rs**
A text expression that specifies one or more dimension values to be used by the custom aggregation. When using a literal to specify more than one dimension member, separate the values with commas

STEP DIMENSIONS
Indicates that the calculation is a time-series function (see "Time-Series Functions" on page A-14).

stepdim
A text expression that specifies the dimension along which the time-series function is calculated. When using a literal to specify more than one dimension name, separate the names with commas.

APPLY TO AGGMAP
Indicates that the calculated temporary member is added *only* to the aggmaps identified by *aggmaps*.

aggmaps
A text expression that specifies the name of one or more aggmap objects to which the temporary calculated member is added. When using a literal to specify more than one aggmap object, separate the names with commas. The temporary calculated member is added to each of the specified aggmap objects.

APPLY FOR VARIABLE
Indicates that the temporary calculated member is added *only* to the variables identified by *variables*.

variables
A text expression that specifies the one or more variable names for which the temporary calculated member is added to. When using a literal to specify more than one variable name, separate the names with commas. The temporary calculated member is added to the default aggmap object of each specified variable.

Important: When a specified variable does not have a default aggmap, using this clause generates an error. Use AGGMAP SET or $AGGMAP to specify a default aggmap for the variable.

APPLY WITH RELATION
Indicates that the temporary calculated member is added only to those aggmap objects whose aggregation specification contains a RELATION command for the relation specified by *relation*.

relation
A text expression that specifies the name of the relation for which a temporary calculated member should be added.

Notes

Finding Out Information About Temporary Calculated Members

Once you have added a temporary calculated member using the MAINTAIN command, you can use AGGMAPINFO to discover the temporary calculated members you have added, the equations used to calculate members, and the dimension members used in the right-hand side of equations used to calculate custom members.

Examples

Example 16–38 Creating Calculated Dimension Members with Aggregated Values

Assume that an analytic workspace has a dimension named `letter` and a variable named `my_quantity` with the following definitions and permanent values.

```
DEFINE letter DIMENSION TEXT
DEFINE my_quantity VARIABLE DECIMAL <letter>

LETTER                 MY_QUANTITY
-------------- ------------------------------
A                                        10.00
B                                       100.00
```

You can define temporary dimension members for the `letter` dimension and aggregate data in `my_quantity` for those members following these steps:

1. Determine the aggregation that you want to perform and define and populate the necessary supporting objects.

   a. Create an empty child-parent relation for the `letter` dimension

   ```
   DEFINE letter.parentrel RELATION letter <letter>

   LETTER                LETTER.PARENTREL
   -------------- ------------------------------
   A              NA
   B              NA
   ```

b. Define a simple model to be used to calculate values associated with the `letter` dimension

```
DEFINE my_model MODEL
MODEL
  DIMENSION letter
 END
```

c. Define and compile a simple aggmap to be used to calculate `my_quantity` values associated with the `letter` dimension

```
DEFINE my_aggmap AGGMAP
AGGMAP
   RELATION letter.parentrel PRECOMPUTE(NA)
   MODEL my_model PRECOMPUTE(NA)
  END

COMPILE my_aggmap
```

d. Define a variable to contain the definition for the custom aggregation, This new variable will be the same as `my_quantity` except that has `my_aggmap` as its default aggmap.

```
DEFINE my_quantity_definition VARIABLE DECIMAL <letter>

CONSIDER my_quantity_definition
PROPERTY '$AGGMAP' 'my_aggmap'

REPORT my_quantity_definition

LETTER             MY_QUANTITY_DEFINITION
-------------- ------------------------------
A                                          NA
B                                          NA
```

2. Add temporary members to the `letter` dimension and specify how variable values for those members are to be calculated.

```
MAINTAIN letter ADD SESSION 'C' = 'A' * 'B'
MAINTAIN letter ADD SESSION 'D' = AGGREGATION('A', 'B') -
    APPLY TO AGGMAP my_aggmap
MAINTAIN letter ADD SESSION 'E' = 'C' + 'D' -
    APPLY WITH RELATION letter.parentrel
MAINTAIN letter ADD SESSION 'F' = 10 * 'E' -
    APPLY FOR VARIABLE my_quantity_definition
```

A report of the `letter` dimension shows the new dimension members.

```
LETTER
--------------
A
B
C
D
E
F
```

3. Aggregate `my_quantity` using the aggmap object named `my_aggmap`.

```
REPORT AGGREGATE(my_quantity USING my_aggmap)

                AGGREGATE(MY_QUANTITY USING
LETTER                   MY_AGGMAP)
-------------- ------------------------------
A                                       10.00
B                                      100.00
C                                    1,000.00
D                                      110.00
E                                    1,110.00
F                                   11,100.00
```

Assume now that you issue the UPDATE and COMMIT statements to update and commit your analytic workspace. Then you detach the analytic workspace and end your session.

Later you start a new session and attach the same analytic workspace. When you ask for a description of the analytic workspace you can see that all of the objects that were in the analytic workspace when the UPDATE was issued still exist.

```
DEFINE LETTER DIMENSION TEXT

DEFINE LETTER.PARENTREL RELATION LETTER <LETTER>

DEFINE MY_QUANTITY VARIABLE DECIMAL <LETTER>

DEFINE MY_MODEL MODEL
MODEL
DIMENSION letter
END

DEFINE MY_AGGMAP AGGMAP
AGGMAP
RELATION letter.parentrel PRECOMPUTE(NA)
MODEL my_model PRECOMPUTE(NA)
END

DEFINE MY_QUANTITY_DEFINITION VARIABLE DECIMAL <LETTER>
```

However, when you report on the `letter` dimension and the `my_quantity` variable, the temporary dimension members that you added in the previous session and their related values in the `my_quantity` variable do not exist.

```
LETTER
--------------
A
B

REPORT letter.parentrel

LETTER                LETTER.PARENTREL
-------------- ------------------------------
A              NA
B              NA

REPORT my_quantity

LETTER                  MY_QUANTITY
-------------- ------------------------------
A                                        10.00
B                                       100.00

LETTER             MY_QUANTITY_DEFINITION
-------------- ------------------------------
A                                           NA
B                                           NA

REPORT AGGREGATE(my_quantity USING my_aggmap)

                AGGREGATE(MY_QUANTITY USING
LETTER                   MY_AGGMAP)
-------------- ------------------------------
A                                        10.00
B                                       100.00
```

MAINTAIN ADD TO PARTITION

The MAINTAIN ADD TO PARTITION statement adds previously-populated dimension or composite values to a partition of a previously-defined partition template object.

> **Tip:** Use MAINTAIN MOVE TO PARTITION to maintain partition values when you have already populated a partitioned variable.

Syntax

MAINTAIN *partition-template* ADD TO PARTITION *partition valuelist*

Arguments

partition-template
A text expression that is the name of a previously-defined partition template object.

ADD TO PARTITION
Specifies that values are to be added to the partition.

partition
A text expression that is the name of a previously-defined partition in the partition template specified by *partition-template*.

valuelist
Text literals or a TEXT or ID expression specifying the values to be added. When it is a TEXT expression, each element (line) is treated as a separate value. The values in the expression are added exactly as they are typed.

For a concat dimension, you can specify a value of the concat dimension, or the name of a component dimension and a value or position of that dimension. You can use the values of a dimension surrogate as the values of *value*.

Note that you cannot partition along an `INTEGER` dimension.

TO
Indicates a range of values.

Examples

For an example of adding values to a partition, see Example 16–44, "Adding and Deleting Partition Values" on page 16-99.

MAINTAIN DELETE

The MAINTAIN command with the DELETE keyword deletes members from non-concat dimensions and composites; or deletes the data of previously-partitioned variables from one partition to another as it changes the dimension or composite values defined for a partition in the partition template which the variables are dimensioned.

Note: You can also issue a MAINTAIN DELETE statement for a surrogate dimension that has a Maintain trigger. In this case, Oracle OALP only executes the Maintain trigger program; no other action occurs. See "Trigger Programs" on page 1-14 for more information for more information. Issuing a MAINTAIN statement for a surrogate dimension that does *not* have a Maintain trigger, returns an error.

Syntax

The syntax for using the DELETE keyword of the MAINTAIN command to delete members varies depending on the type of object from which you are deleting the members. For this reason, the following separate entries are provided for MAINTAIN DELETE:

- MAINTAIN DELETE dimension
- MAINTAIN DELETE composite
- MAINTAIN DELETE FROM PARTITION

MAINTAIN DELETE dimension

The MAINTAIN command with the DELETE keyword deletes dimension members from non-concat dimensions.

Note: You can also issue a MAINTAIN DELETE statement for a surrogate dimension that has a Maintain trigger. In this case, Oracle OALP only executes the Maintain trigger program; no other action occurs. See "Trigger Programs" on page 1-14 for more information for more information. Issuing a MAINTAIN statement for a surrogate dimension that does *not* have a Maintain trigger, returns an error.

See also: MAINTAIN DELETE composite

Syntax

MAINTAIN *dimension* DELETE *dim-arg*

where *dim-arg* is one of the following constructs:

value [[TO] *value*]

ALL

rel-dim [*valuelist*]

{FIRST | LAST} *n*

n PERIODS {FIRST | LAST}

boolean-expression

{BOTTOM | TOP} *n* BASEDON *exp*

LONGLIST

NTH *n*

{BOTTOM | TOP} *n-percent* PERCENTOF *expression*

NOCONVERT *nonconarg*

POSLIST *poslistarg*

family-phrase

valueset

Arguments

dimension
A non-concat dimension, already defined in an attached analytic workspace, whose values are to be deleted.

***value* [[TO] *value*]**
Specifies one value, a list of values, or a range of values (using TO to specify an inclusive range) to be deleted from the values of a dimension. For *value* you can specify an existing value, a text expression whose value is an existing value, a valueset (containing one or more dimension names), or (except for a NUMBER dimension) an integer expression whose value represents the position of a dimension value. For dimensions of type DAY, WEEK, MONTH, QUARTER, and YEAR, *value* can also be a DATE expression or a text expression that represents a date; Oracle OLAP deletes the time period within which the date falls. For dimensions of type DAY, WEEK, MONTH, QUARTER, and YEAR, you can delete values only from the beginning or the end of the existing list of values. When you delete a certain quantity of integers from an INTEGER dimension, the integers in the whole of the resulting series will be automatically adjusted into simple numerical order.

ALL
Deletes all dimension values. This does *not* delete the definition of the dimension or composite itself.

***rel-dim* [*valuelist*]**
Deletes the dimension values that are related to the listed values of a related dimension. The *valuelist* can be one value, a list of values, or a range of values (using TO to specify an inclusive range). When you omit *valuelist,* all values related to the current status of *rel-dim* are deleted.

For dimensions of type DAY, WEEK, MONTH, QUARTER, and YEAR, the related values must be deleted from the beginning or the end of the existing list. For example, assume that the first values in the `month` dimension are the months of 1995. In this case, you can maintain `month` by specifying `year` as the *rel-dim* and `Yr95` as the *valuelist* of years.

Instead of specifying a dimension name for *rel-dim,* you can specify the name of the relation. This enables you to choose which relation is used when there is more than one. You cannot supply a *valuelist* when you specify the name of a relation.

Every dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR is related to all other dimensions of those type through an implicit relation. When you delete values of one DAY, WEEK, MONTH, QUARTER, or YEAR dimension by specifying another dimension with one of those types as the *rel-dim,* Oracle OLAP uses the implicit relation by default. However, when an explicit relation is defined between the two DAY, WEEK, MONTH, QUARTER, or YEAR dimensions, you can override the default by specifying the name of the explicit relation as the *rel-dim.*

FIRST *n*
LAST *n*
Deletes the first or last *n* dimension values in the list; *n* can be any numeric expression. DECIMAL and SHORTDECIMAL values are truncated to integers. When you delete a certain quantity of integers from an INTEGER dimension, the integers in the whole of the resulting series will be automatically adjusted into simple numerical order.

***n* PERIODS FIRST**
***n* PERIODS LAST**
These arguments are only valid for dimensions of type DAY, WEEK, MONTH, QUARTER, or YEAR dimension. Specifying this argument deletes the first or last *n* values in the list; *n* can be any numeric expression. DECIMAL and SHORTDECIMAL values are truncated to integers. The *n* PERIODS FIRST and *n* PERIODS LAST arguments have the same effect as, but are faster than, the FIRST *n* and LAST *n* arguments.

boolean-expression
Deletes all dimension values for which the Boolean expression is `TRUE`. The *boolean-expression* must be dimensioned by the *dimension* from which you the values deleted. When it has additional dimensions, their status must each be limited to one value. When you use the *boolean-expression* argument with a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR, values that meet the criterion will be deleted only when they are at the beginning or the end of the list of dimension values.

TOP *n* BASEDON *exp*
BOTTOM *n* BASEDON *exp*
Deletes the top or bottom *n* values of the dimension based on the highest (TOP) or lowest (BOTTOM) values in *exp.* The expression must be dimensioned by the *dimension* or the *composite* from which you the values deleted. When it has additional dimensions, their status must each be limited to one value. When you use TOP *n* BASEDON *exp* or BOTTOM *n* BASEDON *exp* for a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR, values that meet the criterion will be deleted only when they are at the beginning or end of the list of dimension values.

LONGLIST
Indicates a long list (up to 2,000 values) of individual dimension values to delete. When there are fewer than 300 values, LONGLIST is not needed.

NTH *n*
Deletes the *nth* value in a dimension's full set of values.

BOTTOM *n-percent* PERCENTOF *expression*
TOP *n-percent* PERCENTOF *expression*
Deletes values of a dimension by finding the top or bottom performers based on a criterion. This construction sorts values and deletes them based on their contribution, by percentage, to an expression. For example:

```
MAINTAIN product DELETE TOP 30 PERCENTOF TOTAL(sales, product)
```

will sort products in descending order by each product's contribution to `TOTAL(sales, product)` and then deletes the product, starting from the top, until the cumulative total of `sales` by `product` reaches or exceeds 30 percent of all sales.

NOCONVERT *noconargs*
Deletes a dimension value based on its numeric position. NOCONVERT takes an argument whose values are the numeric positions to be deleted in the maintained dimension. See the explanation of LIMIT command (NOCONVERT).

POSLIST *poslist-exp*
Deletes a dimension value based on its numeric position. POSLIST takes a text argument whose values are the numeric positions to be deleted in the maintained dimension. See the explanation of LIMIT command (using POSLIST).

family phrase
Deletes a dimension value based on its family tree. See the explanation of LIMIT command (using parent relation).

valueset
Deletes the values in the dimension that match the values in the valueset.

Notes

Deleting Temporary Calculated Members From Dimensions

When you use a MAINTAIN DELETE statement to delete a temporary calculated member, Oracle OLAP:

1. Deletes the member from the dimension.
2. Removes the calculation from all aggmap objects that currently contain the corresponding calculation.

Dimension Surrogates

You cannot use a dimension surrogate as the *dimension* argument of a MAINTAIN DELETE command. However, you can use a dimension surrogate a value within the command.

Sequence for Integer Dimension

When you use MAINTAIN to delete values in an integer dimension, the values are renumbered to keep the normal sequence of integers (1, 2, 3, ...).

Examples

Example 16–39 Deleting Dimension Values by Value

This statement deletes `Omaha` and `Newark` from the values for `city`.

```
MAINTAIN city DELETE 'Omaha' 'Newark'
```

Example 16–40 Deleting the First Five Values of a Dimension

In this example, you use the INTEGER variable `intvar` to remove the first five cities from the dimension `city`.

```
intvar = 5
MAINTAIN city DELETE FIRST intvar
```

Example 16–41 Deleting Dimension Values Based on a Boolean Expression

Here you remove from `city` all those cities with a population of less than 75,000 people. You use the variable `population.c`, which contains the population for each city.

```
MAINTAIN city DELETE population.c LT 75000
```

Example 16–42 Deleting Dimension Values Using Surrogate to Specify Values

Assume that `prodid` is a NUMBER dimension and `prodtype` is a TEXT dimension surrogate for `prodid`. Assume also that the values of `prodid` are `17`, `40`, and `56`. The values of `prodtype` are `Two-Person Tent`, `Three-person Tent`, and `Four-person Tent`. The following statement deletes a value from `prodid` and from its surrogate.

```
MAINTAIN prodid DELETE prodid(prodtype 'Three-Person Tent')
```

Example 16–43 Deleting Related MONTH Values

In this example, you use the related dimension `quarter` to remove values of the dimension `month`. All months related to the values of `quarter` currently in the status are deleted.

```
LIMIT quarter TO FIRST 1
```
MAINTAIN month DELETE quarter

MAINTAIN DELETE composite

The MAINTAIN command with the DELETE keyword deletes dimension members from composites.

See also: MAINTAIN DELETE dimension

Syntax

MAINTAIN *composite* DELETE *comp-arg*

where *comp-arg* is one of the following constructs:

valuelist

ALL

base-dim [*valuelist*]

boolean-expression

{TOP | BOTTOM} *n* BASEDON *exp*

{TOP | BOTTOM} *n-percent* PERCENTOF *expression*

LONGLIST

Arguments

composite

A composite whose values are to be deleted. When the composite is a named composite, it must be defined in an attached analytic workspace. When the composite is unnamed, it must have been used in defining an object in an attached analytic workspace.

Use the SPARSE keyword to refer to an unnamed composite (for example, `SPARSE <market product>`).

valuelist

Specifies one or more values to be deleted from the composite. The *valuelist* can be text constants or a text expression.

ALL

Deletes all composite values. This does *not* delete the definition of the composite itself.

***base-dim* [*valuelist*]**

Deletes the values that include the listed values of a base dimension of the composite. The argument *valuelist* can be one value, a list of values, or a range of values (using TO to specify an inclusive range). You cannot use position numbers to specify a range of values. When you omit *valuelist,* Oracle OLAP deletes all values that include *base-dim* values currently in status.

boolean-expression

Deletes all composite values for which the Boolean expression is `TRUE`. The *boolean-expression* must be dimensioned by the *dimension* or the *composite* from which you the values deleted. When it has additional dimensions, their status must each be limited to one value.

TOP *n* BASEDON *exp*
BOTTOM *n* BASEDON *exp*

Deletes the top or bottom *n* values based on the highest (TOP) or lowest (BOTTOM) values in *exp.* The expression must be dimensioned by the *composite* from which you the values deleted. When it has additional dimensions, their status must each be limited to one value.

BOTTOM *n-percent* PERCENTOF *expression*
TOP *n-percent* PERCENTOF *expression*
Deletes values by finding the top or bottom performers based on a criterion. This construction sorts values and deletes them based on their contribution, by percentage, to an expression.

LONGLIST
Indicates a long list (up to 2,000 values) of individual values to delete. When there are fewer than 300 values, LONGLIST is not needed.

MAINTAIN DELETE FROM PARTITION

The MAINTAIN DELETE FROM PARTITION command deletes the data of previously-partitioned variables from one partition to another as it changes the dimension or composite values defined for a partition in the partition template which the variables are dimensioned.

> **Tip:** Use MAINTAIN MOVE TO PARTITION to maintain partition values when you have already populated a partitioned variable.

Syntax

MAINTAIN *partition-template* DELETE FROM PARTITION *partition* { *dim-arg*| *comp-arg*}

Arguments

partition-template
A text expression that is the name of a previously-defined partition template object.

partition
A text expression that is the name of a previously-defined partition in the partition template specified by *partition-template*.

DELETE FROM PARTITION
Specifies that values are to be deleted from the partition and from partitioned variables dimensioned using a partition template that includes the partition.

dim-args
Specifies the values of a dimension that to use when deleting partitioned variable values and when redefining the values that are in the partition You can use any of the constructs specified for the *dim-arg* argument in MAINTAIN DELETE dimension.

comp--args

Specifies the values of a composite to use when deleting partitioned variable values and when redefining the values that are in the partition You can use any of the constructs specified for the *comp-arg* argument in MAINTAIN DELETE composite.

Examples

Example 16–44 Adding and Deleting Partition Values

Assume that you have defined the following objects in your analytic workspace. on

```
DEFINE time DIMENSION TEXT
DEFINE time_parentrel RELATION time <time>
DEFINE product DIMENSION TEXT
DEFINE partition_sales_by_year PARTITION TEMPLATE <time product> -
   PARTITION BY LIST (time) -
    (PARTITION time_2004 VALUES ('2004', 'Dec2004', 'Jan2004', '31Dec2004', -
          '01Dec2004', '31Jan2004', '01Jan2004') <TIME PRODUCT> -
     PARTITION time_2003 VALUES ('2003', 'Dec2003', 'Jan2003', '31Dec2003', -
         '01Dec2003', '31Jan2003', '01Jan2003') <TIME PRODUCT> -
     PARTITION time_2002 VALUES ('2002', 'Dec2002', 'Jan2002', '31Dec2002', -
         '01Dec2002', '31Jan2002', '01Jan2002') <TIME PRODUCT>)
```

Assume that instead of having all sales values dimensioned levels by all time values of a year in a partition, you want to have partitions by days and by summary time values (month and year). To change `partition_sales_by_year` to reflect this new partitioning scheme, you issue the following statements.

```
"Create the new partition
CHGDFN partition_sales_by_year DEFINE -
         (PARTITION partition_month_years VALUES () <time product>)
"Delete the values for months and years from the partitions for years
MAINTAIN partition_sales_by_year DELETE FROM PARTITION time_2004 '2004'-
        'Dec2004' 'Jan2004'
MAINTAIN partition_sales_by_year DELETE FROM PARTITION time_2003 '2003'-
        'Dec2003''Jan2003'
MAINTAIN partition_sales_by_year DELETE FROM PARTITION time_2002 '2002'-
       'Dec2002' 'Jan2002'
"Add the month and year values to the new partition for summary values
MAINTAIN partition_sales_by_year ADD TO PARTITION partition_month_years '2004'-
      'Dec2004' 'Jan2004' '2003' 'Dec2003''Jan2003' '2002' 'Dec2002' 'Jan2002'
```

The `partition_sales_by_year` partition template object now has the following definition.

```
DEFINE PARTITION_SALES_BY_YEAR PARTITION TEMPLATE <TIME PRODUCT> -
   PARTITION BY LIST (TIME) -
    (PARTITION TIME_2004 VALUES ('31Dec2004', '01Dec2004', '31Jan2004', -
       '01Jan2004') <TIME PRODUCT> -
     PARTITION TIME_2003 VALUES ('31Dec2003', '01Dec2003', '31Jan2003', -
       '01Jan2003') <TIME PRODUCT> -
     PARTITION TIME_2002 VALUES ('31Dec2002', '01Dec2002', '31Jan2002', -
       '01Jan2002') <TIME PRODUCT> -
     PARTITION PARTITION_MONTH_YEARS VALUES ('2004', 'Dec2004', 'Jan2004', -
       '2003', 'Dec2003', 'Jan2003', '2002', 'Dec2002', 'Jan2002')-
         <TIME PRODUCT>)
```

MAINTAIN MERGE

The MAINTAIN command with the MERGE keyword provides a quick way to make sure all dimension or composite values on a separate list are included in a non-concat dimension or composite. Using the MERGE keyword with the MAINTAIN command automatically adds the new values from the list and ignores the duplicates. This method of entering dimension values can save a significant amount of time when you have a large number of values to enter.

You can use MERGE with dimensions of any data type, including DAY, WEEK, MONTH, QUARTER, and YEAR dimensions. However, since Oracle OLAP provides a quick way of adding values of DAY, WEEK, MONTH, QUARTER, and YEAR dimensions through the ADD keyword, the MERGE keyword may not be as useful with DAY, WEEK, MONTH, QUARTER, and YEAR dimensions as it is with TEXT or ID dimensions.

At the same time as you are merging values into a dimension, you can also update a relation that involves that dimension.

Note: You can also issue this MAINTAIN MERGE statement for a surrogate dimension that has a Maintain trigger. In this case, Oracle OALP only executes the Maintain trigger program one time for each value in *exp*; no other action occurs. See "Trigger Programs" on page 1-14 for more information for more information. Issuing a MAINTAIN statement for a surrogate dimension that does *not* have a Maintain trigger, returns an error.

Syntax

MAINTAIN *dimension*|*composite* MERGE *exp* [RELATE *relation*]

dimension
A non-concat dimension, already defined in an attached analytic workspace, whose values are to be entered or changed.

composite
A composite whose values are to be added, deleted, or merged. When the composite is a named composite, it must be defined in an attached analytic workspace. When the composite is unnamed, it must have been used in defining an object in an attached analytic workspace. Use the SPARSE keyword to refer to an unnamed composite (for example, `SPARSE <market product>`).

exp
Specifies an expression whose values are to be merged with *dimension*; for example, the name of a dimensioned text variable that contains dimension values, or a single-cell text variable whose value is a multiline list of dimension values. MAINTAIN MERGE ignores any NAs in *exp.* When *dimension* is an integer dimension, then *exp* specifies the number of values that you want in the dimension. When the actual total is less, MAINTAIN MERGE adds enough values to reach the specified total. For example, when an integer dimension has 10 positions, `MERGE 5` has no effect; but `MERGE 15` would add 5 values.

RELATE *relation*
Specifies a relation to be updated as new values from *exp* are merged into *dimension.* At least one of the dimensions of *exp* must also appear in the definition of *relation.* When *exp* is a single-cell value, you cannot use the RELATE phrase.

Examples

Example 16–45 Using the MERGE Keyword with Composites

Suppose you want to define a composite that is made up of all combinations of the first three values of the `product` dimension and the first five values of the `district` dimension. You can efficiently include all 15 values with the following statements.

```
DEFINE comp_proddist COMPOSITE <product district>
LIMIT product TO FIRST 3
LIMIT district TO FIRST 5
MAINTAIN comp_proddist MERGE <product district>
```

This method works with conjoint dimensions as well.

MAINTAIN MOVE

A MAINTAIN command with the MOVE keyword has different effects depending on the object on which it operates:

- When maintaining a dimension, MAINTAIN MOVE changes the position of one or more values in a non-concat dimension or a dimension of type TEXT, ID, or INTEGER or adds previously-populated dimension or composite values to a partition
- When maintaining a partition, MAINTAIN MOVE moves the data of a previously-partitioned variables from one partition to another as it changes the dimension or composite values defined for a partition in the partition template which the variables are dimensioned.

Note: You can also issue a MAINTAIN MOVE dimension value statement for a surrogate dimension that has a Maintain trigger. In this case, Oracle OALP only executes the Maintain trigger program; no other action occurs. See "Trigger Programs" on page 1-14 for more information for more information. Issuing a MAINTAIN statement for a surrogate dimension that does *not* have a Maintain trigger, returns an error.

Syntax

The syntax for using the MAINTAIN command with the MOVE keyword depends on the type of the object being maintained.

For this reason, the following separate entries are provided for MAINTAIN MOVE:

- MAINTAIN MOVE dimension value
- MAINTAIN MOVE TO PARTITION

MAINTAIN MOVE dimension value

A simple MAINTAIN MOVE statement changes the position of one or more values in a non-concat dimension or a dimension of type TEXT, ID, or INTEGER. You cannot use the MOVE keyword of the MAINTAIN command with composites or with dimensions of type DAY, WEEK, MONTH, QUARTER, or YEAR.

Note: You can also issue a MAINTAIN MOVE statement for a surrogate dimension that has a Maintain trigger. In this case, Oracle OALP only executes the Maintain trigger program; no other action occurs. See "Trigger Programs" on page 1-14 for more information for more information. Issuing a MAINTAIN statement for a surrogate dimension that does *not* have a Maintain trigger, returns an error.

Syntax

MAINTAIN *dimension* MOVE *value* [TO *value*] {FIRST|<u>LAST</u>|BEFORE *position*|AFTER *position*}

Arguments

dimension
A non-concat dimension, already defined in an attached analytic workspace, whose values are to be entered or changed. The dimension must be of type TEXT, ID, or INTEGER. You cannot specify a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR.

value
Specifies one or more values of *dimension*. You can specify these values as:

- A literal value.
- An expression whose value is a dimension value.
- For all dimensions except NUMBER dimensions, an INTEGER expression whose value represents the position of a dimension value.
- A valueset.

For a concat dimension, you can specify a value of the concat dimension, or the name of a component dimension and a value or position of that dimension.

TO
Indicates a range of values.

FIRST
LAST
Specify the position to which values will be moved. FIRST indicates that the values are to be moved to the beginning of the value list. LAST (the default) indicates that the values are to be moved to the end of the value list. When you are moving a certain quantity of integers in an INTEGER dimension, that quantity of integers will be moved to the beginning or to the end of the existing series of integers, and the integers in the whole of the resulting series will be automatically adjusted into simple numerical order.

BEFORE *position*
AFTER *position*
Specify a position before or after which the dimension values are to be moved. For *position* you can specify an existing dimension value, a character expression whose value is an existing dimension value, or an integer expression whose value represents the position of a dimension value. When you move a certain quantity of integers in an INTEGER dimension, then that quantity of integers moves before or after the integer position you specify, and the integers in the whole of the resulting series automatically adjust into simple numerical order.

For a concat dimension, you can specify as *position* a value of the concat dimension or the position of a value in a component dimension. See "Sorting Values" on page 16-106.

Notes

Dimension Surrogates

You cannot use a dimension surrogate as the *dimension* argument of a MAINTAIN MOVE command. However, you can use a dimension surrogate values as a *value* to within the statement.

For example, assume that `prodid` is a NUMBER dimension and `prodtype` is a TEXT dimension surrogate for `prodid`. The values of `prodid` are `17`, `40`, and `56`. Assume also that the values of `prodtype` are `Two-Person Tent`, `Three-Person Tent`, and `Four-Person Tent`. The following statement moves the last value to the first position in both the dimension and its surrogate.

```
MAINTAIN prodid MOVE prodid(prodtype 'Four-Person Tent') FIRST
```

Sorting Values

You can sort the values of a dimension with the following statements.

```
LIMIT dimension TO ALL
SORT dimension A sort-criterion
MAINTAIN dimension MOVE VALUES(dimension) FIRST
```

The sorting criterion can be any expression you choose (see the SORT command). To sort the dimension alphabetically, use the dimension itself as the criterion (see Example 16–48, "Moving Dimension Values into Sorted Order" on page 16-106. After using the SORT command to sort the dimension values, you use the MAINTAIN command to make the sorted order permanent.

You can use the SORT command for a temporary sort of the values of a dimension with a type of DAY, WEEK, MONTH, QUARTER, or YEAR. For example, you might want to use the sorted order in a report. However, you cannot use the MAINTAIN command to save the sorted order as the permanent order of a dimension with the type of DAY, WEEK, MONTH, QUARTER, or YEAR. The values of these types of dimensions must be stored in increasing chronological order.

Examples

Example 16–46 Moving a Dimension Value to a Specific Position

This statement moves the position of the city `Houston` to the position following the fifth dimension value.

```
MAINTAIN city MOVE 'Houston' AFTER 5
```

Example 16–47 Moving a Dimension Value to the End of the Status List

In this example, you use the TEXT variable `textvar` to move `Seattle` to the end of the list of cities.

```
textvar = 'Seattle'
MAINTAIN city MOVE textvar LAST
```

Example 16–48 Moving Dimension Values into Sorted Order

Here you put the values of `city` in alphabetical order.

```
SORT city A city
MAINTAIN city MOVE VALUES(city) FIRST
```

Example 16–49 Moving Values of Concat Dimensions

The following statement moves the `reg.dist.ccdim` concat dimension value `<district: 'Denver'>` after the concat dimension value `<region: 'West'>`.

```
MAINTAIN reg.dist.ccdim MOVE <district: 'Denver'> AFTER <region: 'West'>
```

The following statement moves the concat dimension value `<district: 'Denver'>` after the position that corresponds to the first value of the component `district` dimension. If the first value in the status of `district` is `Atlanta`, then `<district: 'Denver'>` moves after the value `<district: 'Atlanta'>` in the concat dimension.

```
MAINTAIN reg.dist.ccdim MOVE <district: 'Denver'> AFTER <district: 1>
```

The following statement moves the concat dimension value `<district: 'Dallas'>` after the third value of the concat dimension.

```
MAINTAIN reg.dist.ccdim MOVE <district: 'Dallas'> AFTER 3
```

MAINTAIN MOVE TO PARTITION

A MAINTAIN MOVE TO PARTITION statement combines both add and move capabilities: You can use a MAINTAIN MOVE TO PARTITION statement to:

- Add previously-populated dimension or composite values to a partition in the same manner as MAINTAIN ADD TO PARTITION
- Change the dimension or composite values defined for a partition in the partition template by which the variables are dimensioned and, at the same time, move the data of a previously-partitioned variables dimensioned by those dimensions and composites from one partition to another.

Syntax

MAINTAIN *partition-template* MOVE TO PARTITION *partition value* [TO *value*]

Arguments

partition-template
A text expression that is the name of a previously-defined partition template object.

MOVE TO PARTITION
Specifies that values are to be added to the partition or moved from one partition to another.

partition
A text expression that is the name of a previously-defined partition in the partition template specified by *partition-template*.

value
Specifies one or more values of a previously-populated dimension or composite. You can specify these values as:

- A literal value.
- An expression whose value is a dimension value.
- For all dimensions except NUMBER dimensions, an INTEGER expression whose value represents the position of a dimension value.
- A valueset.

For a concat dimension, you can specify a value of the concat dimension, or the name of a component dimension and a value or position of that dimension. You can use the values of a dimension surrogate as the values of *value*.

TO
Indicates a range of values.

Examples

Example 16–50 Specifying the Values of a Partition Using Valuesets

Assume that you have defined a partition template object with the following definition that does not specify the actual dimension values for each partition.

```
DEFINE PARTITION_SALES_BY_YEAR PARTITION TEMPLATE <TIME PRODUCT> -
   PARTITION BY LIST (TIME) -
    (PARTITION TIME_2004 VALUES () <TIME PRODUCT> -
     PARTITION TIME_2003 VALUES () <TIME PRODUCT> -
     PARTITION TIME_2002 VALUES () <TIME PRODUCT>)
```

To specify the values of each partition using valuesets, you take the following steps:

1. Define a valueset for each year's values.

   ```
   DEFINE vs_2004 VALUESET time
   LIMIT vs_2004 to '01Dec2004' '31Dec2004' '01Jan2004''31Jan2004' -
        'Jan2004' 'Dec2004' '2004'
   DEFINE vs_2003 VALUESET time
   LIMIT vs_2003 to '01Dec2003' '31Dec2003' '01Jan2003''31Jan2003' -
        'Jan2003' 'Dec2003' '2003'
   DEFINE vs_2002 VALUESET time
   LIMIT vs_2002 to '01Dec2002' '31Dec2002' '01Jan2002''31Jan2002' -
        'Jan2002' 'Dec2002' '2002'
   ```

2. Using MAINTAIN MOVE statements, specify values for the partitions of the partition template.

   ```
   MAINTAIN partition_sales_by_year MOVE TO PARTITION time_2004 vs_2004
   MAINTAIN partition_sales_by_year MOVE TO PARTITION time_2003 vs_2003
   MAINTAIN partition_sales_by_year MOVE TO PARTITION time_2002 vs_2002
   ```

When you issue a DESCRIBE statement, you can see that the description of the `partition_sales_by_year` partition template now includes the appropriate values of `time` in each partition definition.

```
DEFINE PARTITION_SALES_BY_YEAR PARTITION TEMPLATE <TIME PRODUCT> -
    PARTITION BY LIST (TIME) -
    (PARTITION TIME_2004 VALUES -
('2004','Dec2004','Jan2004', 31Dec2004',01Dec2004','31Jan2004','01Jan2004')-
     PARTITION TIME_2003 VALUES -
('2003','Dec2003','Jan2003', 31Dec2003',01Dec2003','31Jan2003','01Jan2003')-
      PARTITION TIME_2002 VALUES -
('2002','Dec2002','Jan2002', 31Dec2002',01Dec2002','31Jan2002','01Jan2002'))
```

MAINTAIN RENAME

The MAINTAIN command with the RENAME keyword changes the spelling of one or more dimension values. You cannot use RENAME keyword with a composite or with dimensions of type INTEGER, DAY, WEEK, MONTH, QUARTER, or YEAR.

Note: You can also issue a MAINTAIN RENAME statement for a surrogate dimension that has a Maintain trigger. In this case, Oracle OALP only executes the Maintain trigger program; no other action occurs. See "Trigger Programs" on page 1-14 for more information for more information. Issuing a MAINTAIN statement for a surrogate dimension that does *not* have a Maintain trigger, returns an error.

Syntax

MAINTAIN *dimension* RENAME {*value new-value*}...

Arguments

dimension
A non-concat dimension of type TEXT or ID that is already defined in an attached analytic workspace and whose values are to be renamed. You cannot specify a dimension of type INTEGER, DAY, WEEK, MONTH, QUARTER, or YEAR.

value
Specifies an existing dimension value to be renamed. You can specify a dimension value, a character expression whose value is a dimension value, or an integer expression whose value represents the position of a dimension value.

new-value
A text constant or a TEXT or ID expression that is the new spelling for the dimension value.

Examples

Example 16–51 Renaming Values of a TEXT Dimension

This statement changes the spelling of the cities `Chic` and `Bost` to `Chicago` and `Boston`.

```
MAINTAIN city RENAME 'Chic' 'Chicago' 'Bost' 'Boston'
```

In this example you use the TEXT variable `textvar` to change the second city to `Atlanta`.

```
textvar = 'Atlanta'
MAINTAIN city RENAME 2 textvar
```

MAKEDATE

The MAKEDATE function returns the DATE value that corresponds to specified integer values for a year, month, and day.

Return Value

DATE

Syntax

MAKEDATE(*year month day*)

Arguments

year
An integer expression that represents the year of the test date. For any year, you can specify the year as a four-digit number in the range 1000 to 9999. For years in the range 1950 to 2049 (the default) or some other range (as set through the YRABSTART option), you have the alternative of specifying a two-digit number that represents the last two digits of the year (96 represents 1996, for example).

month
Any integer expression, normally in the range 1 to 12. When you specify an integer less than 1 or greater than 12, MAKEDATE returns a date in a year prior to or later than the year specified by the integer expression for *year.*

For example, if the arguments to MAKEDATE are (`97 14 21`), MAKEDATE returns the date `February 21, 1998` since, in effect, February 1998 is the fourteenth month of 1997.

day
An integer expression in the range 1 to 31.

Notes

Format of the Date

When you display the result returned by MAKEDATE, the date is formatted according to the date template in the DATEFORMAT option. When the day of the week or the name of the month is used in the date template, the day names specified in the DAYNAMES option and the month names specified in the MONTHNAMES option are used. You can use the result returned by MAKEDATE anywhere that a DATE value is expected.

DATE-to-TEXT Conversion

You can also use the result where a text value is expected. The date is converted automatically to a text value, using the current template in the DATEFORMAT option to format the text value. When you want to override the current DATEFORMAT template, you can convert the date result to text by using the CONVERT function with a date-format argument.

Invalid Dates

When the arguments to MAKEDATE do not represent a valid date between January 1, 1000, and December 31, 9999, MAKEDATE returns an `NA` value.

Examples

Example 16–52 Converting Integers to a Date

The following statements specify the date format and send the output to the current outfile.

```
DATEFORMAT = '<mtextl> <d>, <yyyy>'
SHOW MAKEDATE(97 11 14)
```

These statements produce the following output.

```
November 14, 1997
```

Example 16–53 Calculating a Date Using YYOR, MMOF, and DDOF Functions

The following statement calculates the date one year from today, and sends the output to the current outfile. The TODAY function returns today's date. The INTEGER functions YYOF, MMOF, and DDOF return the INTEGER values that correspond to the year, month, and day of today's date.

```
SHOW MAKEDATE(YYOF(TODAY) + 1 MMOF(TODAY) DDOF(TODAY))
```

When today's date is January 15, 1995, this statement produces the following output.

```
January 15, 1996
```

MAX

The MAX function calculates the larger value of two expressions.

Return Value

DECIMAL

Syntax

MAX(*expression1*, *expression2*)

Arguments

expression1
One expression to be compared.

expression2
The other expression to be compared.

Notes

Dimensions of the Result

Ordinarily, the dimensions of both the expressions you want to compare and the results of MAX are the same. When the dimensions of one expression are a subset of the other's dimensions, then the results of MAX are dimensioned by the larger set of dimensions. In any case, the results of MAX are dimensioned by the union of the dimensions of the two expressions.

Examples

Example 16–54 Calculating Whether Actual or Budget Values Are Larger

Suppose, for each of the first six months of 1996, you want to find whether the `actual` value or the `budget` value is larger for the line item Cost of Goods Sold (Cogs) in the `Sporting` division.

```
LIMIT line TO 'Cogs'
LIMIT division TO 'Sporting'
LIMIT month TO 'Jan96' TO 'Jun96'
REPORT DOWN month actual budget MAX(actual budget)
```

The preceding statements produce the following output.

```
DIVISION: SPORTING
               --------------LINE--------------
               --------------COGS--------------
                                     MAX
                                   (ACTUAL
MONTH            ACTUAL     BUDGET    BUDGET)
-------------- ---------- ---------- ----------
Jan96          287,557.87 279,773.01 287,557.87
Feb96          315,298.82 323,981.56 323,981.56
Mar96          326,184.87 302,177.88 326,184.87
Apr96          394,544.27 386,100.82 394,544.27
May96          449,862.25 433,997.89 449,862.25
Jun96          457,347.55 448,042.45 457,347.55
```

17 MAXBYTES to MODTRACE

This chapter contains the following OLAP DML statements:

MAXBYTES

The MAXBYTES function counts the number of bytes in the longest line of a multiline text expression. The result returned by MAXBYTES has the same dimensions as the specified expression.

Return Value

INTEGER

Syntax

MAXBYTES(*text-expression*)

Arguments

text-expression
The text expression whose bytes for each line are to be counted.

Notes

Single-Byte Characters

When you are using a single-byte character set, you can use the MAXCHARS function instead of the MAXBYTES function.

NTEXT Data Type

This function does not accept NTEXT arguments, because it is oriented toward byte-manipulation instead of character manipulation. It always returns values of type TEXT. When you must use this function on NTEXT values, use the CONVERT or TO_CHAR function to convert the NTEXT value to TEXT.

Examples

Example 17–1 Finding the Length of the Longest Line Using Bytes

You would like to know the length of the longest line in a text variable called `mytext`. The following example shows the value of the variable and the result returned by MAXBYTES.

The statement

```
SHOW mytext
```

produces the following output.

```
This is a multiline text variable.
The longest line is this one in the middle.
The third line is short.
```

The statement

```
SHOW MAXBYTES(mytext)
```

produces the following output.

```
43
```

MAXCHARS

The MAXCHARS function counts the number of characters in the longest line of a multiline text expression. The result returned by MAXCHARS has the same dimensions as the specified expression.

Return Value

INTEGER

Syntax

MAXCHARS(*text-expression*)

Arguments

text-expression
The text expression whose characters for each line are to be counted.

Notes

multibyte Characters

When you are using a multibyte character set, you can use the MAXBYTES function instead of the MAXCHARS function.

TEXT and NTEXT

MAXCHARS accepts either a TEXT or NTEXT argument. It does not perform an automatic conversion to either data type. It returns the information that is correct for the data type of the specified argument.

Examples

Example 17–2 Finding the Length of the Longest Line Using Characters

You would like to know the length of the longest line in a text variable called `mytext`. The following example shows the value of the variable and the result returned by MAXCHARS.

The statement

```
SHOW mytext
```

produces the following output.

```
This is a multiline text variable.
The longest line is this one in the middle.
The third line is short.
```

The statement

```
SHOW MAXCHARS(mytext)
```

produces the following output.

```
43
```

MAXFETCH

The MAXFETCH option sets an upper limit on the size of a data block generated by a FETCH command specified in the *OLAP_command* parameter of the `OLAP_TABLE` function. For more information on the FETCH command, see FETCH. For more information on the `OLAP_TABLE` function, see the *Oracle OLAP Reference*.

Return Value

INTEGER

Syntax

MAXFETCH = *integer-expression*

Arguments

integer-expression
An expression representing the maximum size in bytes of a data block generated by FETCH. The minimum value for MAXFETCH is 1K (approximately 1,000 bytes), and the maximum value is 2GB (approximately 2,000,000,000 bytes). The default value of MAXFETCH is `256K`.

Notes

Improving Performance of Queries Using OLAP_TABLE

The setting of MAXFETCH can effect the performance of queries using the `OLAP_TABLE` function. Large queries with joins of `OLAP_TABLE` function may run faster with higher settings. However, larger settings use more memory which can cause slower performance when there are multiple users. The setting of MAXFETCH does not affect a `SELECT` using only one `OLAP_TABLE` function.

MAXFETCH can cause a FETCH error

When FETCH cannot package a data block within the size limit set by MAXFETCH, it produces an error, and no data is returned to the client. By setting MAXFETCH, you can produce an error, rather than run out of memory, when you attempt to fetch too much data.

Examples

Limiting Data Blocks to 4K

The following statement limits the size of data blocks to 4K.

```
MAXFETCH = 4096
```

MEDIAN

The MEDIAN function calculates the median of the values of an expression. The *median* is the middle number in a given sequence of numbers.

Return Value

DECIMAL

Syntax

MEDIAN(*expression* [*dimensions*])

Arguments

expression
The expression whose median value is to be calculated.

dimensions
The dimensions of the result. By default, MEDIAN returns a single value. When you indicate one or more dimensions for the results, MEDIAN calculates a median for each value of the dimensions that are specified and returns an array of values. Each dimension must be a dimension of *expression.* You cannot use a related dimension as the *dimensions* argument.

Notes

NA Values

MEDIAN is affected by the NASKIP option. When NASKIP is set to `YES` (the default), MEDIAN ignores `NA` values and returns the median of the values that are not `NA`. When NASKIP is set to `NO`, MEDIAN returns `NA` when any value of the expression is `NA`. When all the values of the expression are `NA`, MEDIAN returns `NA` for either setting of NASKIP.

Examples

Example 17–3 Calculating Median Monthly Sales

This example shows how to calculate the median monthly sales of sportswear for each sales district.

```
LIMIT product TO 'Sportswear'
REPORT W 12 HEADING 'Median Sales' MEDIAN(sales district)
```

The preceding statements produce the following output.

```
DISTRICT          Median Sales
----------------- ------------
Boston               67,923.05
Atlanta             152,186.52
Chicago              94,372.06
Dallas              160,854.60
Denver               86,745.40
Seattle              53,950.28
```

MIN

The MIN function calculates the smaller value of two expressions.

Return Value

DECIMAL

Syntax

MIN(*expression1*, *expression2*)

Arguments

expression1
One expression to be compared.

expression2
The other expression to be compared.

Notes

Dimensions of the Result

Ordinarily, the dimensions of both the expressions you want to compare and the results of MIN are the same. When the dimensions of one expression are a subset of the other's dimensions, then the results of MIN are dimensioned by the larger set of dimensions. In any case, the results of MIN are dimensioned by the union of the dimensions of the two expressions.

Examples

Example 17–4 Calculating Whether Actual or Budget Values Are Smaller

Suppose, for each of the first six months of 1996, you want to find whether the `actual` value or the `budget` value is smaller for the line item Cost of Goods Sold (`Cogs`) in the `Sporting` division.

```
LIMIT line TO 'Cogs'
LIMIT division TO 'Sporting'
LIMIT month TO 'Jan96' TO 'Jun96'
REPORT DOWN month actual budget MIN(actual budget)
```

The preceding statements produce the following output.

```
DIVISION: SPORTING
               --------------LINE--------------
               --------------COGS--------------
                                          MIN
                                        (ACTUAL
MONTH            ACTUAL     BUDGET    BUDGET)
-------------- ---------- ---------- ----------
Jan96          287,557.87 279,773.01 279,773.01
Feb96          315,298.82 323,981.56 315,298.82
Mar96          326,184.87 302,177.88 302,177.88
Apr96          394,544.27 386,100.82 386,100.82
May96          449,862.25 433,997.89 433,997.89
Jun96          457,347.55 448,042.45 448,042.45
```

MMOF

The MMOF function returns an integer in the range of 1 to 12, giving the month in which a specified date falls. The result returned by MMOF has the same dimensions as the specified DATE expression.

Return Value

INTEGER

Syntax

MMOF(*date-expression*)

Arguments

date-expression

An expression that has the DATE data type, or a text expression that specifies a date. See "TEXT-to-DATE Conversion" on page 17-12.

Notes

TEXT-to-DATE Conversion

In place of a DATE expression, you can specify a text expression that has values that conform to a valid input style for dates. The values of the text expression are converted automatically to DATE values, using the current setting of the DATEORDER option to resolve any ambiguity.

Examples

Example 17–5 Finding the Current Month

The following statement determines the month in which today's date falls.

```
SHOW MMOF(TODAY)
```

When today's date is January 15, 1996, this statement produces the following output.

```
1
```

MODDAMP

The MODDAMP option specifies a weighting factor that damps out oscillations between iterations when you use the Gauss-Seidel method for solving simultaneous equations in a model. MODDAMP can allow the solution of models that would otherwise never converge because the oscillation between equations is stable. In these cases, the oscillations never decay without damping.

With the Gauss-Seidel method, Oracle OLAP tests each model equation for convergence or divergence in each iteration over a block of simultaneous equations. The tests are made by comparing the results of the current iteration to the results from the previous iteration. When MODDAMP specifies a weighting factor that is greater than zero, the value that Oracle OLAP tests and stores after each iteration is a weighted average of the current and previous results. For equations that oscillate between iterations, you can therefore use MODDAMP to damp out the oscillations and either prevent divergence or speed up the convergence of the equations.

Data type

DECIMAL

Syntax

MODDAMP = {*n*|<u>0.00</u>}

Arguments

n
A decimal value, greater than or equal to zero and less than one, that specifies the weighting factor. The closer MODDAMP is to 0.00, the more weight is given to the

value from the current iteration. The default value is 0.00, which gives full weight to the current iteration.

When MODDAMP is greater than zero, Oracle OLAP calculates the weighted average for the current iteration as follows.

calcvalue * (1 - MODDAMP) + *weightavg*

where:

calcvalue is the value calculated from the model equation in the current iteration.

weightavg is the weighted average calculated in the previous iteration.

See "Stored Weighted Average" on page 17-14.

Notes

Specifying the Solution Method

The MODDAMP option is used only with the Gauss-Seidel method for solving simultaneous equations. The MODSIMULTYPE option determines the solution method that is being used. The possible settings for MODSIMULTYPE are `GAUSS`, for the Gauss-Seidel method, and `AITKENS`, for the Aitkens delta-squared method.

Increasing Convergence Speed

MODDAMP is used in calculating the results of all model equations in every simultaneous block, whether they oscillate between iterations or not. For equations that do not oscillate, convergence is slowed down when the value of MODDAMP is greater than zero. Therefore, when your model contains some equations that oscillate and some that do not, you might be able to speed up overall convergence by setting MODDAMP to a small nonzero value, such as `0.20`. A small nonzero value will slow down the convergence of non-oscillating equations only slightly, while speeding up the convergence of oscillating equations.

Stored Weighted Average

When the model equation does not converge or diverge on the current iteration, the weighted average calculated in the current iteration is stored. In the next iteration, Oracle OLAP uses this stored average as *weightavg* (that is, the weighted average calculated in the previous iteration) in the formula for the weighted average.

In the first iteration over a block, Oracle OLAP uses the starting value of the target variable (or dimension value) as the *weightavg* (that is, the weighted average calculated in the previous iteration).

Iteration Results Compared

In tests for convergence and divergence in each iteration, Oracle OLAP compares the results of the current iteration to the results from the previous iteration. When MODDAMP is greater than zero, Oracle OLAP tests a *comparison value* that is calculated as follows.

(*weightavg* - *weightavg*) / (*weightavg* PLUS MODGAMMA)

where *weightavg* is the weighted average calculated in the previous iteration

For an explanation of the test for convergence, see MODTOLERANCE. For an explanation of the test for divergence, see MODOVERFLOW.

Options to Control the Solution of Simultaneous Blocks

Altering the value of MODDAMP is just one of the steps you can take in attempting to speed up or attain convergence of a simultaneous block. MODEL lists additional options that you can use to control the solution of simultaneous blocks and provides information on running and debugging models.

Examples

Example 17–6 Using the Default MODDAMP Value

The following statements trace a model called `income.bud`, specify that the Gauss-Seidel method should be used for solving simultaneous blocks, limit a dimension, and run the `income.bud` model.

```
MODTRACE = YES
MODSIMULTYPE = 'GAUSS'
LIMIT division TO 'Camping'
income.bud budget
```

These statements produce the following output.

```
(MOD= INCOME.BUD) BLOCK 1: SIMULTANEOUS
(MOD= INCOME.BUD) ITERATION 1: EVALUATION
(MOD= INCOME.BUD) revenue = marketing * 300 - cogs
   ...
(MOD= INCOME.BUD) BUDGET (LINE REVENUE MONTH 'JAN97' ITER 35) =  368.650399101
   ...
(MOD= INCOME.BUD) BUDGET (LINE REVENUE MONTH 'JAN97' ITER 36) =  369.209604252
   ...
(MOD= INCOME.BUD) BUDGET (LINE REVENUE MONTH 'JAN97' ITER 37) =  368.718556135
   ...
(MOD= INCOME.BUD) BUDGET (LINE REVENUE MONTH 'JAN97' ITER 38) =  369.149674626
   ...
(MOD= INCOME.BUD) BUDGET (LINE REVENUE MONTH 'JAN97' ITER 39) =  368.771110244
   ...
(MOD= INCOME.BUD) BUDGET (LINE REVENUE MONTH 'JAN97' ITER 40) =  369.103479583
(MOD= INCOME.BUD) END BLOCK 1
```

The MODDAMP option is set to its default value of 0.00. The equation for the Revenue line item converged in 40 iterations over a block of simultaneous equations. In the trace lines, you can see the results that were calculated for the Revenue line item in the final 6 iterations.

Example 17–7 Setting MODDAMP to Speed Up the Convergence of a Model

The following statements change the value of MODDAMP and run the `income.bud` model.

```
MODDAMP = 0.2
income.bud budget
```

These statements produce the following output.

```
(MOD= INCOME.BUD) BLOCK 1: SIMULTANEOUS
(MOD= INCOME.BUD) ITERATION 1: EVALUATION
(MOD= INCOME.BUD) revenue = marketing * 300 - cogs
(MOD= INCOME.BUD) BUDGET (LINE REVENUE MONTH 'JAN97' ITER 1) =  276.200000000
   ...
(MOD= INCOME.BUD) BUDGET (LINE REVENUE MONTH 'JAN97' ITER 2) =  416.187139753
    ...
(MOD= INCOME.BUD) BUDGET (LINE REVENUE MONTH 'JAN97' ITER 3) =  368.021098186
   ...
(MOD= INCOME.BUD) BUDGET (LINE REVENUE MONTH 'JAN97' ITER 4) =  367.209906847
   ...
(MOD= INCOME.BUD) BUDGET (LINE REVENUE MONTH 'JAN97' ITER 5) =  369.271224267
  ...
(MOD= INCOME.BUD) BUDGET (LINE REVENUE MONTH 'JAN97' ITER 6) =  368.965397407
(MOD= INCOME.BUD) END BLOCK 1
```

In "Using the Default MODDAMP Value" on page 17-15, the equation for the `Revenue` line item converged in 40 iterations. With MODDAMP set to `0.2` in the current example, the same equation converged in just 6 iterations.

MODE

The MODE function returns the mode (the most frequently occurring value) of a numeric expression. When there are no duplicate values in the data, then MODE returns `NA`.

Return Value

DECIMAL

Syntax

MODE(*expression* [*dimensions*])

Arguments

expression
The numeric expression whose mode is to be calculated.

dimensions
The dimensions of the result. When you do not specify any dimensions, MODE calculates the mode over all the dimensions of *expression* and it returns a single value. When you specify one or more dimensions (but fewer than all of the dimensions of *expression*) in the *dimension* argument, then MODE calculates the mode for each value of the dimensions that you specified and returns an array of values. Each dimension must be a dimension of *expression*.

Notes

The Effect of NASKIP

MODE is not affected by the NASKIP option.

More Than One Set of Duplicate Values

When multiple values qualify as having the greatest number of occurrences in the expression, then MODE sorts the values and returns the lowest one. For example, for the data series {4,5,2,3,7,4,6,2,1}, the mode for the series is 2 even though 2 and 4 both occur twice.

Examples

Example 17–8 Reporting the Mode

These examples use the following `geography` and `items` dimensions and `sales2` variable.

```
DEFINE geography DIMENSION TEXT
MAINTAIN geography ADD 'g1' 'g2' 'g3'
DEFINE items DIMENSION TEXT
MAINTAIN items ADD 'Item1' 'Item2' 'Item3' 'Item4' 'Item5'
DEFINE sales2 DECIMAL <geography items>
```

Assume the `sales2` variable has the following data values.

```
               -------------SALES2-------------
               -----------GEOGRAPHY------------
ITEMS              G1         G2         G3
-------------- ---------- ---------- ----------
Item1               30.00      15.00      12.00
Item2               10.00      20.00      18.00
Item3               15.00      20.00      24.00
Item4               30.00      25.00      25.00
Item5                  NA       7.00      21.00
```

- This statement reports the mode that is calculated over the `geography` dimension.

  ```
  REPORT W 22 MODE(sales2, geography)
  ```

  The preceding statement produces the following output.

  ```
                       MODE(SALES2,
  GEOGRAPHY             GEOGRAPHY)
  -------------- ----------------------
  g1                              30.00
  g2                              20.00
  g3                                 NA
  ```

- This statement reports the mode that is calculated over the `items` dimension.

  ```
  REPORT W 18 MODE(sales2, items)
  ```

The preceding statement produces the following output.

```
                  MODE(SALES2,
ITEMS                ITEMS)
-------------- ------------------
Item1                          NA
Item2                          NA
Item3                          NA
Item4                       25.00
ITEM5                          NA
```

- This statement reports the mode that is calculated over all of the dimensions of the `sales2` variable.

```
REPORT MODE(sales2)
```

The preceding statement produces the following output.

```
Mode
----
15
```

MODEL

The MODEL command enters a completely new specification into a new or existing model object. When the model already has a specification, Oracle OLAP overwrites it.

The MODEL command assigns the specification to the most recently defined or considered model (see the DEFINE MODEL and CONSIDER commands). In order for it to do this, there must be a current definition, so you must have defined or considered model during your current session before executing the MODEL command.

Adding a specification to a model object is just one step in modeling data which is discussed in "Models" on page 10.

Syntax

MODEL *specification*

Arguments

specification
A multiline text expression that contains one or more of the following OLAP DML statements:

Assignment statement (SET)
DIMENSION (in models)
INCLUDE

The maximum number of lines you can have in a model is 4,000. Separate statements with newline delimiters (`\n`), or use JOINLINES.

For a discussion of designing a model specification, see "Model Specification" on page 17-22.

Notes

Model Specification

The model specification consists of the following OLAP DML statements:

1. One of the following:
   - Exactly one INCLUDE statement that specifies the name of another model to include. See "Nesting Models" on page 4-11 for more information.
   - One or more DIMENSION (in models) statements coded following the "Guidelines for Writing DIMENSION Statements in a Model" on page 17-32.

   Note: When a model contains an INCLUDE statement, then it cannot contain any DIMENSION statements. However, the model referenced in the INCLUDE statement or the root model in a hierarchy must contain the DIMENSION statements needed by the parent model(s).

2. One or more SET commands or equations written following the "Rules for Equations in Models" on page 21-61. The maximum number of lines you can have in a model is 4,000.

   See also: "Dimension Status and Model Equations" on page 4-12 for information on how Oracle OLAP processes equations in a model.

3. A final END statement that indicates the end of the model specification. (Omit when coding the specification in an Edit window of the OLAP Worksheet.)

MODEL Statement in an Aggregation Specification

Within an aggmap, you can use a special MODEL statement to execute a predefined model. (See MODEL (in an aggregation) for more information.

Model Options

A number of options effect how a model solves simultaneous blocks. These options are listed in Table 17–1, "Model Options" on page 17-23.

Table 17–1 Model Options

Option	Purpose
MODDAMP	For the Gauss-Seidel solution method, specifies a weighting factor that damps out oscillations between iterations.
MODERROR	Specifies the action to be taken when a model equation diverges or a block fails to converge. The possible values are STOP, CONTINUE, and DEBUG.
MODGAMMA	A comparison factor that is used in testing for convergence and divergence. It controls the degree to which the tests compare the absolute amount of change between iterations versus the proportional change. This option is useful in models that test very small values.
MODINPUTORDER	Specifies whether equations in a simultaneous block are executed in the order in which you place them in the model or in an order determined by the model compiler.
MODMAXITERS	The maximum number of iterations to perform in seeking a solution for a block.
MODOVERFLOW	A value that is used in testing for divergence. It controls how large the change in the results must be between iterations for an equation to be considered to have diverged.
MODSIMULTYPE	The solution method to use. The possible values are AITKENS (for the Aitkens delta-squared method) and GAUSS (for the Gauss-Seidel method).
MODTOLERANCE	A value that is used in testing for convergence. It controls how closely the results must match between iterations for an equation to be considered to have converged.

Methods of Calculating Data Within a Variable

Both models and aggmap objects calculate data values within a variable based on relationships among dimension members. When a parent-child relationship exists among dimension members (that is, the dimension has a hierarchical structure) and all aggregate values can be calculated using the same method, then you can use a RELATION (for aggregation) statement within an aggregation specification to calculate the values. However, when the dimension is not hierarchical and different equations are needed to calculate the values, then you must define a model. You can use a MODEL (in an aggregation) to execute the MODEL within an aggregation specification or you can run a model at the command line using the syntax shown in "Running a Model" on page 4-15.

Deleting a Model Specification

You can remove the specification of a model without deleting the model definition. Consider the model with the CONSIDER command. Then issue a MODEL command and enter the word END as the model specification.

Examples

Example 17–9 Model Specified in a Program

In the following example, a simple model is created (or overwritten) in a program called `myprog`. The first line in the program defines or considers the model. The second line contains the MODEL command, which provides the lines of the model.

This model calculates the line items in a budget. The model equations are based on a `line` dimension.

```
DEFINE myprog PROGRAM
PROGRAM
IF NOT EXISTS('myModel')
  THEN DEFINE myModel
  ELSE CONSIDER myModel
MODEL JOINLINES(-
  'DIMENSION line month' -
  'Opr.Income = Gross.Margin - Marketing' -
  'Gross.Margin = Revenue - Cogs' -
  'Revenue = LAG(Revenue, 1, month) * 1.02' -
  'Cogs = LAG(Cogs, 1, MONTH) * 1.01' -
  'Marketing = LAG(Opr.Income, 1, month) * 0.20' -
  'END')
END
```

Example 17–10 Model from an Input File

This example presents the text of the same simple model, but it is stored in an ASCII disk file called `budget.txt`.

```
DEFINE income.budget MODEL
MODEL
DIMENSION line month
Opr.Income = Gross.Margin - Marketing
Gross.Margin = Revenue - Cogs
Revenue = LAG(Revenue, 1, month) * 1.02
Cogs = LAG(Cogs, 1, month) * 1.01
Marketing = LAG(Opr.Income, 1, month) * 0.20
END
```

To include the `income.budget` model in your analytic workspace, execute the following statement in which `myinpfiles` is a directory object.

```
INFILE 'myinpfiles/budget.txt'
```

Example 17–11 Creating a Model

Suppose that you define a model, called `income.calc`, that calculates line items in the income statement.

```
define income.calc model
ld Calculate line items in income statement
```

After defining the model, you can use the `MODEL` command or the OLAP Worksheet editor to enter the specification for the model. A model specification can contain `DIMENSION` commands, assignment statements and comments. All the `DIMENSION` commands must come before the first equation. For the current example, you can specify the lines shown in the following model.

```
DEFINE INCOME.CALC MODEL
LD Calculate line items in income statement
MODEL
DIMENSION line
net.income = opr.income - taxes
opr.income = gross.margin - (marketing + selling + r.d)
gross.margin = revenue - cogs
END
```

When you write the equations in a model, you can place them in any order. When you compile the model, either with the `COMPILE` command or by running the model, the order in which the model equations are solved is determined. When the calculated results of one equation are used as input to another equation, then the equations are solved in the order in which they are needed.

To run the `income.calc` model and use `actual` as the solution variable, you execute the following command.

```
income.calc actual
```

When the solution variable has dimensions other than the dimensions on which model equations are based, then a loop is performed automatically over the current status list of each of those dimensions. For example, `actual` is dimensioned by `month` and `division`, as well as by `line`. When `division` is limited to `ALL`, and `month` is limited to `OCT96` to `DEC96`, then the `income.calc` model is solved for the three months in the status for each of the divisions.

Example 17–12 Building a Scenario Model

Suppose, for example, you want to calculate profit figures based on optimistic, pessimistic, and best-guess revenue figures for each division. The steps for building this scenario model are explained in the following example.

You can call the scenario dimension `scenario` and give it values that represent the scenarios you want to calculate.

These commands give `scenario` the values `optimistic`, `pessimistic` and `bestguess`.

```
DEFINE scenario DIMENSION TEXT
LD Names of scenarios
MAINTAIN scenario ADD optimistic pessimistic bestguess
```

These commands create a variable named `plan` dimensioned by three other dimensions (`month`, `line`, and `division`) in addition to the `scenario` dimension.

```
DEFINE plan DECIMAL <month line division scenario>
LD Scenarios for financials
```

For this example, you need to enter input data, such as revenue and cost of goods sold, into the `plan` variable.

For the best-guess data, you can use the data in the `budget` variable. Limit the `line` dimension to the input line items, and then copy the `budget` data into the `plan` variable.

```
LIMIT scenario TO 'BESTGUESS'
LIMIT line TO 'REVENUE' 'COGS' 'MARKETING' 'SELLING' 'R.D'
plan = budget
```

You might want to base the optimistic and pessimistic data on the best-guess data. For example, optimistic data might be fifteen percent higher than best-guess data, and pessimistic data might be twelve percent less than best-guess data. With `line` still limited to the input line items, execute the following commands.

```
plan(scenario 'OPTIMISTIC') = 1.15 * plan(scenario 'BESTGUESS')
plan(scenario 'PESSIMISTIC') = .88 * plan(scenario 'BESTGUESS')
```

The final step in building a scenario model is to write a model that calculates results based on input data. The model might contain calculations very similar to those in the `budget.calc` model shown earlier in this chapter.

You can use the same equations for each scenario or you can use different equations. For example, you might want to calculate the cost of goods sold and use a different constant factor in the calculation for each scenario. To use a different constant factor for each scenario, you can define a variable dimensioned by `scenario` and place the appropriate values in the variable. When the name of your variable is `cogsval`, then your model might include the following equation for calculating the `cogs` line item.

```
cogs = cogsval * revenue
```

By using variables dimensioned by `scenario`, you can introduce a great deal of flexibility into your scenario model.

Similarly, you might want to use a different constant factor for each division. You can define a variable dimensioned by `division` to hold the values for each division. For example, when labor costs vary from division to division, then you might dimension `cogsval` by `division` as well as by `scenario`.

When you run your model, you specify `plan` as the solution variable. For example, when your model is called `scenario.calc`, then you solve the model with this command.

```
scenario.calc plan
```

A loop is performed automatically over the current status list of each of the dimensions of `plan`. Therefore, when the `scenario` dimension is limited to `ALL` when you run the `scenario.calc` model, then the model is solved for all three scenarios: `optimistic`, `pessimistic`, and `bestguess`.

DIMENSION (in models)

The DIMENSION statement at the beginning of a model tells Oracle OLAP the names of one or more dimensions to which the model assigns data or to which it refers in dimension-based equations. A dimension-based equation assigns the results of a calculation to a target that is represented by one or more values of a dimension.

Syntax

DIMENSION *dimension1* [, *dimensionN*]

Arguments

dimension
One or more dimensions, including base dimensions of composites, on which model equations are based. You can specify the name of a dimension surrogate instead of the dimension for which is a surrogate. You can then use the values of the surrogate instead of the values of the dimension.

Notes

Dimension-Based Equations

When an equation (SET) assigns data to a dimension value or refers to dimension values in its calculations, it is called a *dimension-based* equation. Note that a dimension-based equation does not need to refer to the dimension itself, but only to the *values* of the dimension. Therefore, when the model contains any dimension-based equations, you must specify the name of each of these dimensions in a DIMENSION statement at the beginning of the model. This allows Oracle OLAP to determine the dimension to which each dimension value belongs. You can specify the name of a dimension surrogate instead of the dimension for which it is a surrogate. You can then use the values of the surrogate instead of the values of the dimension.

In addition, when a model contains any dimension-based equations, you must supply the name of a *solution variable* when you run the model. The solution variable is both the source and the target of data for the model. It holds the input data used in dimension-based calculations, and Oracle OLAP stores the calculation results in designated values of the solution variable. The solution variable is generally dimensioned by all the dimensions on which the model equations are based. For example, in a financial application, the model might be based on the `line` dimension, and the solution variable might be `actual`, which has `line` as one of its dimensions.

Dimension-based equations provide flexibility in modeling. Since you do not need to specify the modeling variable until you solve a model, you can run the same model with different solution variables. For example, you might run the same model with the `actual` variable, with a "best case" budget variable, and with a "worst case" budget variable.

A dimension must be specified in a DIMENSION command when a dimension-based equation refers to a value of the dimension either as a source of the data used in the calculation or as the target to which the results will be assigned. In the following example, `Gross.Margin`, `Revenue`, and `Cogs` are values of the `line` dimension, so `line` is specified in a DIMENSION command.

```
DIMENSION line
Gross.Margin = Revenue - Cogs
```

Dimension is a Function Argument

A dimension must be specified in a DIMENSION command when the dimension is an argument to a function that uses a dimension value as its data source. In the following example, `month` must be specified in a DIMENSION command.

```
DIMENSION line, month
Revenue = lag(Revenue, 1, month) * 1.05
```

The writer of the preceding model expects to use a solution variable that is dimensioned by `line` and `month`. Therefore, when the model is run, the LAG function will operate on a solution variable that has the specified time dimension (`month`) as one of its dimensions. However, since the model compiler cannot anticipate the time dimension of the solution variable, you must specify it in a DIMENSION command. When you fail to include `month` in a DIMENSION command, an error occurs when you attempt to compile the model.

In a function that operates on time-series data (such as MOVINGTOTAL or LAG), the *dimension* argument is optional when the dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR. For example, you can omit `month` from the LAG function in the preceding example. However, you must still specify the appropriate time dimension in a DIMENSION command.

Solution Variable

When you run a model that contains dimension-based equations, you specify a solution variable, which is both the source and the target of data for the model. The solution variable is generally dimensioned by all the dimensions that are listed in the DIMENSION commands used in the model. Or, when a solution variable is dimensioned by a composite, the DIMENSION commands can list base dimensions of the composite. The DIMENSION commands can be explicit in the model or inherited through an included model. See "Incompatibility with INCLUDE" on page 17-31.

Working with Composites

When you expect to run a model with a solution variable that has a composite in its dimension list, you can specify a base dimension of the composite in a DIMENSION command. Your model equations will assign results to values of the base dimension. Oracle OLAP automatically creates any new values that are needed in the composite.

Multiple DIMENSION Commands

You can include a separate DIMENSION command for every dimension referred to or used in dimension-based equations, or you can specify all the dimensions in a single DIMENSION command.

Location of Commands

You must place all the DIMENSION commands at the beginning of the model, before any equations.

Incompatibility with INCLUDE

When a model contains an INCLUDE statement, it cannot contain any DIMENSION commands. The INCLUDE statement specifies another model to include in the current model. In this case, the current model inherits its DIMENSION commands, if any, from the included model. For more information in including models, see INCLUDE.

Inherited DIMENSION commands must satisfy all the requirements specified for explicit DIMENSION commands. See "Guidelines for Writing DIMENSION Statements in a Model" on page 17-32.

Dimension Order

When more than one dimension is specified by the DIMENSION commands in a model, the order in which the dimensions are listed is important:

- When a model equation contains a name that might be a dimension value, Oracle OLAP searches through the dimensions that appear in the model's explicit or inherited DIMENSION commands, in the order you list the dimensions, to determine whether the name matches a dimension value of a listed dimension. The search concludes as soon as a match is found. Therefore, when two or more listed dimensions have a dimension value with the same name, Oracle OLAP assumes that the value belongs to the dimension specified earliest in a DIMENSION command. When the name does not match a value of a listed dimension, Oracle OLAP then searches through the variables in the attached workspaces to find a match.
- When model equations assign results to values of a target dimension, Oracle OLAP constructs code that will loop over the values of the other, non-target, dimensions listed in the DIMENSION commands. The non-target dimension listed first in the DIMENSION commands is treated as the slowest-varying dimension. For example, when MONTH is the first non-target dimension listed in a DIMENSION command and DIVISION is the second, Oracle OLAP loops through all the divisions for the first month, then all the divisions for the second month, and so on.

Guidelines for Writing DIMENSION Statements in a Model

When you write `DIMENSION` statements, you should keep these points in mind:

- In the `DIMENSION` statements, you must list the names of all the dimensions on which model equations are based. In the following example, `gross.margin`, `revenue`, and `cogs` are values of the `line` dimension, so `line` is specified in a `DIMENSION` statement.

  ```
  DIMENSION line
  gross.margin = revenue - cogs
  ```

- `DIMENSION` statements must also list any dimension that is an argument to a function that refers to a dimension value. In the following example, `month` must be specified in a `DIMENSION` statement.

  ```
  DIMENSION line, month
  revenue = LAG(revenue, 1, month) * 1.05
  ```

- When a model equation assigns results to a dimension value, then code is constructed that loops over the values of any of the other nontarget dimensions listed in the `DIMENSION` statements. The nontarget dimension listed first in the `DIMENSION` statements is treated as the slowest-varying dimension.

- A model executes most efficiently when you observe the following guidelines for coordinating the dimensions in `DIMENSION` statements and the dimensions of the solution variable:
  - List the target dimension of the model as the *first* dimension in the `DIMENSION` statements and as the *last* dimension in the definition of the solution variable.
  - In `DIMENSION` statements, list the nontarget dimensions in the *reverse* order of their appearance in the definition of the solution variable. This means that the fastest-varying and slowest-varying nontarget dimensions are in the same order in the model and in the solution variable.
- When the solution variable has dimensions that are not used or referred to in model equations, then do not include them in `DIMENSION` statements.
- When your analytic workspace contains a variable whose name is the same as a dimension value, or when the same dimension value exists in two different dimensions, then there could be ambiguities in your model equations. Since you can use a variable and a dimension value in exactly the same way in a model equation, a name might be the name of a variable, or it might be a value of any dimension in your analytic workspace.
- Your `DIMENSION` statements are used to determine whether each name reference in an assignment statement is a variable or a dimension value. "Compiling a Model" on page 4-14 explains how the name references are resolved.

See Also: "Models" on page 4-10, SET, and MODEL for information on:

- Entering statements in a model
- How to refer to values of dimensions
- Explanation of how Oracle OLAP constructs code from the statements
- Explanation of how Oracle OLAP handles the situation in which the solution variable has more dimensions or fewer dimensions than are listed in DIMENSION commands

Examples

Example 17–13 Simplified Model for Budget Estimates

The following statements define a simplified model that estimates budget values for the items on an income statement.

```
DEFINE income.budget MODEL
LD Model for estimating budget line items
MODEL
dimension line, month
Revenue = 1.05 * LAG(Revenue 1 month)
Gross.Margin = Revenue - Cogs
Opr.Income = Gross.Margin - (Marketing + Selling + R.D)
Net.Income = Opr.Income - Taxes
END
```

The model equations are based on the `line` dimension, so `line` is specified in the DIMENSION command. The dimension `month` is the time dimension in the LAG function that operates on REVENUE values, so `month` is also specified in the DIMENSION command.

When you run the model, Oracle OLAP loops over the values in the current status of the `month` dimension.

INCLUDE

The INCLUDE command includes one model within another model. You can use the INCLUDE command only within models.

Syntax

INCLUDE *model*

Arguments

model
The name of a model to include in the current model. The current model is referred to as the parent model. The model that you include is referred to as the base model.

Notes

Limitations

A model can contain only one INCLUDE command, and the INCLUDE command can specify the name of just one base model to be included.

Nesting Models

You can nest models by placing an INCLUDE command in a base model. For example, model `myModel1` can include model `myModel2`, and model `myModel2` can include model `myModel3`. The nested models form a hierarchy. In this example, `myModel1` is at the top of the hierarchy, and `myModel3` is at the root.

A base model cannot include a model at a higher level in the hierarchy. In the preceding example, `myModel2` cannot include `myModel1`, and `myModel3` cannot include `myModel1` or `myModel2`.

Incompatibility with DIMENSION

When a model contains an INCLUDE command, it cannot contain any DIMENSION (in models) commands. A parent model inherits its dimensions, if any, from the DIMENSION commands in the root model of the included hierarchy. For example, suppose model `myModel1` includes model `myModel2`, and model `myModel2` includes model `myModel3`. When model `myModel3` contains one or more DIMENSION commands, then models `myModel1` and `myModel2` both inherit their dimensions from the DIMENSION commands in model `myModel3`.

Location

You must place the INCLUDE command before any equations in the model.

Dependencies Among Equations

When compiling a model that contains an INCLUDE command, the compiler considers the dependencies among the equations from all the included models when it orders and blocks the equations. Therefore, when you run the `MODEL.COMPRPT` program to examine the results of the compilation or when you set the MODTRACE option to `YES` before running the parent model, you might find that equations from different levels in the hierarchy of included models are interspersed. See Example 17–15, "Producing a Compilation Report" on page 17-39**.**

When the compiler finds no dependencies among the equations from the included models, it executes the equations in the root model first and the equations in the parent model last.

Compiling a Parent Model

When you compile a parent model, the compiler will compile all the base models under it in the included hierarchy when compiled code does not already exist. When the compiler detects an error in an included model, neither it nor any model above it in the hierarchy is compiled. When the root model of the included hierarchy contains an error, the higher-level models are unable to inherit any DIMENSION (in models) commands from the root model. In this case, the compiler might report an error in a parent model when the source of the error is actually in the root model. For example, the compiler might report that a target dimension value does not exist in any attached analytic workspace.

On the other hand, when the compiler detects an error in a parent model but finds no errors in the included models, the included models are compiled even though the parent model is not.

Modular Models

The INCLUDE command creates modular models. When certain equations are common to several models, you can place these equations in a separate model and include that model in other models as needed.

Facilitating What-If Analyses

The INCLUDE command also facilitates what-if analyses. An experimental model can draw equations from a base model and selectively replace them with new equations.

Masking Equations

To support what-if analyses, Oracle OLAP allows equations in a model to mask previous equations. The previous equations can come from the same model or from included models. A masked equation is not executed. When you run the MODEL.COMPRPT program after compiling the model, you will see that the masked equation is not shown in the report on the compiled model.

Masking can take place when an equation assigns a value to a variable or dimension value that is also the target of a previous equation. The masking rules are as follows:

- When the target in the earlier equation is qualified exactly the same as the target in the later equation, the earlier equation is masked and the later equation is executed. The following example illustrates two equations with targets that are identically qualified.

  ```
  Equation from a base model:      BUDGET(LINE REVENUE) = 5000
  Equation from the parent model:  BUDGET(LINE REVENUE) = 3500
  ```

  In this example, the equation from the base model is masked and the equation from the parent model is executed.

- When the target in the earlier equation is more qualified than the target in the later equation, the earlier equation is masked. The later equation is executed.

  The target that is more qualified is the one that will affect the fewest dimension values. Consider the following equations from a base model and a parent model.

  ```
  Equation from a base model:      BUDGET(LINE REVENUE) = 2500
  Equation from the parent model:  BUDGET = 4000
  ```

  The equation from the base model is more qualified because it assigns data only for the REVENUE value of the LINE dimension. The equation from the parent model assigns data to all the values of the LINE dimension. In this example, the equation from the base model is masked and the equation from the parent model is executed.

- When the target in the earlier equation is less qualified than the target in the later equation, no masking takes place. Both equations are executed.

  Consider the following equations from a base model and a parent model.

  ```
  Equation from a base model:      BUDGET = LAG(ACTUAL, 1, MONTH)
  Equation from the parent model:  BUDGET(LINE REVENUE) = 6500
  Equation from the parent model:  BUDGET(LINE COGS) = 4000
  ```

The equation from the base model assigns data to all the values of the LINE dimension. The equations from the parent model are more qualified because each assigns data only for a single value of the LINE dimension. In this example, the equation from the base model is executed first, and then the equations from the parent model are executed.

This functionality enables you to assign a large number of values with one equation and use subsequent equations to replace or test individual values.

- When the target in the earlier equation is qualified differently from the target in the later equation, no masking takes place. Both equations are executed. In the following example, both equations are executed.

```
Equation from a base model:      BUDGET(LINE REVENUE) = 5000
Equation from the parent model:  BUDGET(LINE COGS) = 4500
```

Examples

Example 17–14 Including a Model

This example shows a parent model named `income.plan` that includes a base model named `base.lines`.

```
DEFINE income.plan MODEL
MODEL
INCLUDE base.lines
revenue = LAG(revenue, 1, month) * 1.02
cogs = LAG(cogs, 1, month) * 1.01
taxes = 0.3 * opr.income
END

DEFINE BASE.LINES MODEL
MODEL
DIMENSION line month
net.income = opr.income - taxes
opr.income = gross.margin - marketing
gross.margin = revenue - cogs
END
```

Example 17–15 Producing a Compilation Report

The following statements compile the parent model and produce a compilation report.

```
COMPILE income.plan
MODEL.COMPRPT income.plan
```

These statements produce the following output.

```
MODEL INCOME.PLAN <LINE MONTH>
                      BLOCK 1 (SIMPLE)
INCOME.PLAN     2:    revenue = lag(revenue, 1, month) * 1.02
INCOME.PLAN     3:    cogs = lag(cogs, 1, month) * 1.01
BASE.LINES      4:    gross.margin = revenue - cogs
BASE.LINES      3:    opr.income = gross.margin - marketing
INCOME.PLAN     4:    taxes = 0.3 * opr.income
BASE.LINES      2:    net.income = opr.income - taxes
                      END BLOCK 1
```

MODEL.COMPRPT

The MODEL.COMPRPT program produces a report that shows how model equations are grouped into blocks. For step blocks and for simultaneous blocks with a cross-dimensional dependence, the report lists the dimensions involved in the dependence.

Syntax

MODEL.COMPRPT

Examples

Example 17–16   A Compilation Report for the income.budget Model

The MODEL.COMPRPT program produces a compilation report that shows the block structure of the model that you specify as the program argument and the order of equations within each block. Each equation is identified with the name of the model and its statement number within that model.

The following statements compile the model and invoke MODEL.COMPRPT.

```
COMPILE income.budget
MODEL.COMPRPT income.budget
```

The `MODEL.COMPRPT` statement produces the following compilation report.

```
MODEL INCOME.BUDGET <LINE MONTH>
                   BLOCK 1 (SIMPLE)
INCOME.BUDGET  4:  revenue = lag(revenue, 1, month) * 1.02
INCOME.BUDGET  5:  cogs = lag(cogs, 1, month) * 1.01
INCOME.BUDGET  3:  gross.margin = revenue - cogs
                    BLOCK 2 (STEP-FORWARD <MONTH>)
INCOME.BUDGET  6:   marketing = lag(opr.income, 1, month) * 0.20
INCOME.BUDGET  2:   opr.income = gross.margin - marketing
                    END BLOCK 2
                   END BLOCK 1
```

Example 17–17 A Compilation Report for the income.est Model

The following statement runs the `MODEL.COMPRPT` program, which produces a compilation report for a model named `income.est`.

```
MODEL.COMPRPT income.est
```

The compilation report contains the following output.

```
MODEL INCOME.EST <LINE MONTH>
              BLOCK 1 (STEP-FORWARD <MONTH>)
INCOME.EST 5: revenue = lag(revenue,1,month)+2*lag(marketing,1,month)
INCOME.EST 4: gross.margin = revenue - cogs
               BLOCK 2 (SIMULTANEOUS)
INCOME.EST 2:  net.income = opr.income - taxes
INCOME.EST 3:  opr.income = gross.margin - marketing - selling - r.d
INCOME.EST 6:  marketing = .15 * net.income
INCOME.EST 7:  taxes = .3 * opr.income
               END BLOCK 2
              END BLOCK 1
```

MODEL.DEPRT

The MODEL.DEPRPT program produces a report that lists the variables and dimension values on which each model equation depends. When a dependence is dimensional, the report gives the name of the dimension.

Syntax

MODEL.DEPRT

Examples

Example 17–18 Producing a Dependency Report

The MODEL.DEPRPT program produces a dependency report that lists the variables and dimension values that are the assignment target and data sources for each model equation. For each equation, the assignment target and each data source is listed on a separate line. When a target or data source is a dimension value, its line is marked by an asterisk enclosed in square brackets (`[*]`).

When a target or data source depends on a qualifier, the report specifies the dimension of the qualifier and indicates the type of dependence. The type of dependence can be any of the following:

- LAG -- One-way dependence on previous dimension values
- LEAD -- One-way dependence on later dimension values
- BOTH -- Two-way dependence on both previous and later values
- VARIABLE -- Dependence on either previous or later values, depending on the value of a variable when the model is run
- QDR -- Qualified data reference

Assume that you want to produce a dependency report for the `income.budget` model. The following statement and report illustrate this process.

```
MODEL.DEPRPT income.budget

MODEL INCOME.BUDGET <LINE MONTH>
2     [*](LINE OPR.INCOME):
        [*](LINE GROSS.MARGIN)
        [*](LINE MARKETING)
3     [*](LINE GROSS.MARGIN):
        [*](LINE REVENUE)
        [*](LINE COGS)
4     [*](LINE REVENUE):
        [*](LINE REVENUE)(LAG <MONTH>)
5     [*](LINE COGS):
        [*](LINE COGS)(LAG <MONTH>)
6     [*](LINE MARKETING):
        [*](LINE OPR.INCOME)(LAG <MONTH>)
```

The data sources in statements 4, 5, and 6 have a LAG dependence on the `month` dimension.

MODEL.XEQRPT

The MODEL.XEQRPT program produces a report about the execution of the model. The report specifies the block where the solution failed and shows the values of the model options that were used in solving simultaneous blocks.

Syntax

```
MODEL.XEQRPT
```

Notes

Running MODEL.XEQRPT

Before you can run the MODEL.XEQRPT program, you must

1. Set MODERROR to STOP or CONTINUE.
2. Execute the model.

When the model halts because of an error, run the `MODEL.XEQRPT` program.

Effect of MODERROR on MODEL.XEQRPT

The results returned by MODEL.XEQRPT vary depending on the setting of MODERROR:

- When MODERROR is set to `STOP` and execution of the model halts because of an error, you can run the MODEL.XEQRPT program to produce a report about the execution of the model. The report specifies the block where the solution failed and shows the values of the model options that were used in solving simultaneous blocks.
- When MODERROR is set to `CONTINUE` and one of the blocks in the model is a simultaneous block that either diverges or fails to converge, Oracle OLAP executes any remaining blocks in the model.

  Moreover, Oracle OLAP executes the model for the remaining values in the status of any additional dimensions of the solution variable that are not dimensions of the model. In this case, when you run the MODEL.XEQRPT program when Oracle OLAP finishes executing the model, you will see a report on the solution for the final values of the additional dimensions.

When the simultaneous blocks in the model converged when the model was executed for the final values of the additional dimensions, then `MODEL.XEQRPT` will report that the blocks were solved, even though an earlier execution of the model for another dimension value might have failed. When you wish to see the `MODEL.XEQRPT` for the dimension values where the failure occurred, you can set MODERROR to `STOP` and rerun the model.

Examples

Example 17–19 Producing an Execution Report for the income.est Model

After running the income.est model, you can use the MODEL.XEQRPT program to produce a report on the execution of the model.

The following statement runs the MODEL.XEQRPT program, which produces an execution report for the model.

```
MODEL.XEQRPT income.est
```

The execution report contains the following output.

```
MODEL INCOME.EST <LINE MONTH>
Solution status:     SOLVED
Model options in use:
   MODSIMULTYPE:     AITKENS
   MODMAXITERS:      50
   MODTOLERANCE:     3
   MODOVERFLOW:      3
   MODGAMMA:         1
BLOCK 1 (STEP-FORWARD <MONTH>)
  Solution status:   SOLVED
BLOCK 2 (SIMULTANEOUS)
  Solution status:   SOLVED
  Iterations:        15
```

The report shows the values of the model options that were used in solving the simultaneous blocks and indicates whether each block was solved.

MODERROR

The MODERROR option determines the action that Oracle OLAP takes when a block of simultaneous equations in a model cannot be solved within a specified number of iterations.

> **See:** Table 17–1, "Model Options" on page 17-23 for descriptions of all of the options that control the solution of simultaneous blocks.

Data type

ID

Syntax

MODERROR = {'STOP'|'CONTINUE'}

Arguments

'STOP'
Oracle OLAP sends an error message to the current outfile and stops executing the model. (Default)

'CONTINUE'
Oracle OLAP sends a warning message to the current outfile, stops executing the current block, and resumes execution at the next block in the model.

Notes

Block-Solution Failure

When every equation in a simultaneous block passes a convergence test, the block is considered solved. When any equation diverges or fails to converge within a specified number of iterations, the solution of the block fails and an error occurs. MODERROR controls the action that Oracle OLAP takes when an error occurs.

Attaining Convergence

When an error occurs, you might be able to attain convergence for the block by changing the value of one or more options that control the solution of simultaneous blocks. For example, you can increase the number of iterations that will be attempted or you can change the criteria used in testing for convergence and divergence.

Using 'STOP'

When MODERROR is set to `STOP` and execution of the model halts because of an error, you can run the MODEL.XEQRPT program to produce a report about the execution of the model. The report specifies the block where the solution failed and shows the values of the model options that were used in solving simultaneous blocks.

Using 'CONTINUE'

When MODERROR is set to `CONTINUE` and one of the blocks in the model is a simultaneous block that either diverges or fails to converge, Oracle OLAP executes any remaining blocks in the model.

Moreover, Oracle OLAP executes the model for the remaining values in the status of any additional dimensions of the solution variable that are not dimensions of the model. In this case, when you run the MODEL.XEQRPT program when Oracle OLAP finishes executing the model, you will see a report on the solution for the final values of the additional dimensions.

When the simultaneous blocks in the model converged when the model was executed for the final values of the additional dimensions, then `MODEL.XEQRPT` will report that the blocks were solved, even though an earlier execution of the model for another dimension value might have failed. When you wish to see the `MODEL.XEQRPT` for the dimension values where the failure occurred, you can set MODERROR to `STOP` and rerun the model.

Diagnosing a Problem

You can also use the MODTRACE option to help diagnose a problem in a simultaneous block. When you set MODTRACE to `YES`, Oracle OLAP records each equation in the current outfile before executing it and then records the results of the calculation in the current outfile. By examining the trace, you can observe progress and problems as they develop during the solution process.

Examples

Example 17–20 Debugging a Model

This example assumes that you are connected through OLAP Worksheet and enter the following statements in the Command Input window. The statements set MODERROR to `DEBUG` so that you will be able to debug the `myModel` model (which contains a block of simultaneous equations) when the simultaneous block fails to converge.

```
MODERROR = 'DEBUG'
myModel actual
```

When the simultaneous block fails to converge, you can type an Oracle OLAP or debugger command in the Command Input window in OLAP Worksheet. Since the solution variable, `actual`, is dimensioned by `division`, you might want to display the current value of `division`.

```
SHOW division
Camping
```

MODGAMMA

The MODGAMMA option specifies a value to use in testing how much an equation in a simultaneous block of a model is changing between iterations. MODGAMMA controls the degree to which the test compares the absolute amount of the change between iterations versus the proportional change. MODGAMMA is especially important in testing equations that result in very small values.

> **See:** Table 17–1, "Model Options" on page 17-23 for descriptions of all of the options that control the solution of simultaneous blocks.

Data type

INTEGER

Syntax

MODGAMMA = {*n*|<u>1</u>}

Arguments

n
An integer value to use in testing for convergence and divergence. As Oracle OLAP calculates each equation in a simultaneous block, it constructs a comparison value that is based on the results of the equation for the current iteration and the previous iteration. When the comparison value passes a tolerance test, the equation is considered to have converged. When the comparison value meets an overflow test, the equation is considered to have diverged.

The comparison value that is tested is as follows.

```
(thisResult - prevResult) DIVIDED BY (prevResult PLUS MODGAMMA)
```

where *thisResult* is the result of this iteration and *prevResult* is the result of the previous iteration.

Oracle OLAP calculates the absolute value of the enclosed expression. The default value of MODGAMMA is `1`.

Notes

Testing Convergence

In the test for convergence, the MODTOLERANCE option determines how closely the results of an equation must match between successive iterations. With the default value of `3` for MODTOLERANCE, the equation is considered to have converged when the comparison value is less than `0.001`.

Testing Divergence

In the test for divergence, the MODOVERFLOW option determines how dissimilar the results of an equation must be in successive iterations. With the default value of `3` for MODOVERFLOW, the equation is considered to have diverged when the comparison value is greater than `1000`.

Comparison Value

The comparison value that Oracle OLAP evaluates in tests of convergence and divergence is fundamentally a proportional value. It expresses the change between iterations as a proportion of the previous results. In the test for convergence, the change between iterations must be small in proportion to the previous results. In the test for divergence, the change between iterations must be large in proportion to the previous results. By testing a proportional value, rather than testing the absolute amount of change, Oracle OLAP can apply the same test criteria to all equations, regardless of the magnitude of the equation results.

However, the comparison value that Oracle OLAP tests is not strictly proportional. When the results of an equation are very close to zero, the denominator of a strictly proportional comparison value would also be very close to zero, and thus the comparison value itself would generally be large. Therefore, the test for convergence would be difficult to satisfy, while the test for divergence would be easy to meet. To solve this problem, Oracle OLAP adds the value of MODGAMMA to the denominator of the comparison value. When the default value of 1 is used for MODGAMMA, the effect of MODGAMMA is as follows:

- When the equation results are close to zero, the denominator is close to one and the test is essentially a test of the absolute change between iterations.
- When the equation results are very large, the effect of adding MODGAMMA to the denominator is negligible, and the test is close to being a strictly proportional test.

Controlling Test Sensitivity

For equation values close to zero, you can control the sensitivity to the tests for convergence and divergence by changing the value of MODGAMMA. When equation values are very small, you essentially scale the changes in model values between iterations when you change the value of MODGAMMA. For example, when you change MODGAMMA from 1 to 2, the comparison value is essentially cut in half. As a consequence, you reduce the likelihood that divergence will occur.

Increasing Convergence Speed

The default value of MODGAMMA is appropriate in most situations. When you increase the value of MODGAMMA, the model equations will converge more quickly, but the results will be less precise. The smaller the equation value, the more pronounced is the effect of increasing MODGAMMA; convergence is attained relatively more quickly for small model values, while more precision is lost.

You can also force the simultaneous blocks of a model to converge more quickly by decreasing the value of MODTOLERANCE and thereby relaxing the test for convergence. However, when you do this, you sacrifice the precision of all the results, not just the results of equations with small values.

Therefore, when a model contains some equations with large values and some equations with very small values, it might be preferable to increase MODGAMMA rather than decreasing MODTOLERANCE. By increasing MODGAMMA, you might be able to force equations with small values to converge more quickly while retaining the precision of equations with large values.

Examples

Example 17–21 Using the Default MODGAMMA Value

The following statements specify a trace for a model called `income.bud`, specify that the Gauss-Seidel method should be used for solving simultaneous blocks, limit a dimension, and run the model.

```
MODTRACE = YES
MODSIMULTYPE = 'GAUSS'
LIMIT division TO 'Camping'
income.bud budget
```

These statements produce the following output.

```
(MOD= INCOME.BUD) BLOCK 1: SIMULTANEOUS
   ...
(MOD= INCOME.BUD) BUDGET (LINE NET.INCOME MONTH 'JAN97' ITER 16) = 0.026243533
     ...
(MOD= INCOME.BUD) BUDGET (LINE NET.INCOME MONTH 'JAN97' ITER 17) = 0.024054312
   ...
(MOD= INCOME.BUD) BUDGET (LINE NET.INCOME MONTH 'JAN97' ITER 18) = 0.025788293
   ...
(MOD= INCOME.BUD) BUDGET (LINE NET.INCOME MONTH 'JAN97' ITER 19) = 0.024390642
    ...
(MOD= INCOME.BUD) BUDGET (LINE NET.INCOME MONTH 'JAN97' ITER 20) = 0.025501664
    ...
(MOD= INCOME.BUD) BUDGET (LINE NET.INCOME MONTH 'JAN97' ITER 21) = 0.024608562
```

In the trace, you can see the results that were calculated for the NET.INCOME line item in the final six iterations over a block of simultaneous equations.

The value of MODTOLERANCE is its default value of `3`. This means that for an equation to pass the convergence test, its comparison value must be less than `.001`.

MODGAMMA is set to its default value of `1`. The equation for the NET.INCOME line item passed the convergence test in the twenty-first iteration. The comparison value for `Net.Income` in the twenty-first iteration was calculated as follows.

```
(0.024608562967 - 0.025501664970 = 0.00087) / (0.025501664970 + 1)
```

Example 17–22 Setting MODGAMMA to Speed up the Convergence of a Model

The following statements change the MODGAMMA setting and run the `income.bud` model.

```
MODGAMMA = 2
income.bud budget
```

With MODGAMMA set to `2`, the equation for `Net.Income` converges in the eighteenth iteration. The comparison value for `Net.Income` in the eighteenth iteration is calculated as follows.

```
(0.025788293304 - 0.024054312748 = 0.00086) / (0.024054312748 + 2)
```

MODINPUTORDER

The MODINPUTORDER option controls whether the equations in a simultaneous block of a model are executed in the order in which you place them or in an order determined by the model compiler. MODINPUTORDER has no effect on the order of equations in simple blocks and step blocks.

> **See:** Table 17–1, "Model Options" on page 17-23 for descriptions of all of the options that control the solution of simultaneous blocks.

Data type

BOOLEAN

Syntax

MODINPUTORDER = {YES|<u>NO</u>}

Arguments

YES
The equations in a simultaneous block of a model are executed in the order in which they appear in the model.

NO
The equations in a simultaneous block are executed in an order determined by the model compiler. (Default)

Examples

Example 17–23 Using the Default Order

The following statements define the `income.calc` model.

```
DEFINE income.calc MODEL
MODEL
DIMENSION line month
Net.Income = Opr.Income - Taxes
Opr.Income = Gross.Margin - TOTAL(Marketing + Selling + R.D)
Marketing = LAG(Opr.Income, 1, month)
Gross.Margin = Revenue - Cogs
END
```

The following statements compile the model and produce a compilation report using the MODEL.COMPRPT program.

```
COMPILE income.calc
MODEL.COMPRPT income.calc
```

These statements produce the following output.

```
MODEL INCOME.CALC <LINE MONTH>
             BLOCK 1 (SIMPLE)
INCOME.CALC 5: gross.margin = revenue - cogs
              BLOCK 2 (SIMULTANEOUS <MONTH>)
INCOME.CALC 4: marketing = lag(opr.income, 1, month)
INCOME.CALC 3: opr.income = gross.margin - total(marketing + selling + r.d)
              END BLOCK 2
INCOME.CALC 2: net.income = opr.income - taxes
             END BLOCK 1
```

When you compile `income.calc` with MODINPUTORDER set to its default value of `NO`, you can see that the compiler reverses the order of the equations in the simultaneous block.

Example 17–24 Changing the MODINPUT Value

The following statements set the value of MODINPUTORDER to `YES`, compile the model, and produce a compilation report.

```
MODINPUTORDER = YES
COMPILE income.calc
MODEL.COMPRPT income.calc
```

These statements produce the following output.

```
MODEL INCOME.CALC <LINE MONTH>
                BLOCK 1 (SIMPLE)
INCOME.CALC 5: gross.margin = revenue - cogs
                 BLOCK 2 (SIMULTANEOUS <MONTH>)
INCOME.CALC 3:  opr.income = gross.margin - total(marketing + selling + r.d)
INCOME.CALC 4:  marketing = lag(opr.income, 1, month)
                 END BLOCK 2
INCOME.CALC 2: net.income = opr.income - taxes
                END BLOCK 1
```

You can see that the compiler leaves the simultaneous equations in the order in which you placed them.

MODMAXITERS

The MODMAXITERS option determines the maximum number of iterations Oracle OLAP will perform in attempting to solve a block of simultaneous equations in a model.

> **See:** Table 17–1, "Model Options" on page 17-23 for descriptions of all of the options that control the solution of simultaneous blocks.

Data type

INTEGER

Syntax

MODMAXITERS = {*n*|<u>50</u>}

Arguments

n
A positive integer value that indicates the maximum number of iterations Oracle OLAP should perform in attempting to solve a simultaneous block. The default is `50`.

Notes

Reporting Model Execution Results

When any equation in a simultaneous block diverges or fails to converge within the number of iterations specified by MODMAXITERS, the solution of the block fails and an error occurs. You can use the MODEL.XEQRPT program to produce a report on the results of the model's execution. The report indicates whether a simultaneous block diverged or failed to converge. When a block failed to converge, you can experiment with increasing the value of MODMAXITERS to see if convergence can be attained.

Attaining Convergence

Increasing the value of MODMAXITERS is just one of the steps you can take in attempting to attain convergence for a simultaneous block. For example, you can also experiment with changing the criteria used in testing for convergence and divergence.

Diagnosing a Problem

To see the results of each calculation as Oracle OLAP executes a model, set the MODTRACE option to `YES` before you run the model. Oracle OLAP records each equation in the current outfile before executing it and then records the results of the calculation in the current outfile. By examining the trace, you can observe progress and problems as they develop during the solution process.

Examples

Example 17–25 Model with MODMAXITERS

Suppose a model named MYMODEL contains a block of simultaneous equations that failed to converge within 50 iterations. The following statements increase the value of MODMAXITERS and run the model again.

```
MODMAXITERS = 100
myModel actual
```

MODOVERFLOW

The MODOVERFLOW option is used in testing whether any equation in a simultaneous block of a model has diverged. MODOVERFLOW determines how dissimilar the results of an equation must be in successive iterations for the equation to be considered to have diverged.

> **See:** Table 17–1, "Model Options" on page 17-23 for descriptions of all of the options that control the solution of simultaneous blocks.

Data type

INTEGER

Syntax

MODOVERFLOW = {*n*|<u>3</u>}

Arguments

n

An integer value to use in testing for divergence. As Oracle OLAP calculates each equation in a simultaneous block, it constructs a comparison value that is based on the results of the equation for the current iteration and the previous iteration. When the comparison value meets a divergence test, the equation is considered to have diverged.

The comparison value that is tested is as follows.

(*thisResult* - *prevResult*) / (*prevResult* + MODGAMMA)

where *thisResult* is the result of this iteration and *prevResult* is the result of the previous iteration

In the preceding calculation, MODGAMMA is an INTEGER option that controls the degree to which the comparison value represents the absolute amount of change between iterations versus the proportional change. The default value of MODGAMMA is `1`.

In the divergence test, Oracle OLAP tests whether the comparison value is greater than `10` to the power of MODOVERFLOW. The calculation for this test is as follows.

```
Comparison value  >  10**MODOVERFLOW
```

For the equation to be considered to have diverged, the comparison value must meet the test described earlier. The default value of MODOVERFLOW is `3`. With this default, the comparison value meets the test when it is greater than `1000`.

Notes

Equation Divergence

When an equation diverges, an error occurs. The MODERROR option controls the action that Oracle OLAP takes when an error occurs.

Attaining Convergence

Even when the results of an equation change drastically between successive iterations in the early stages of a solution, the equation might eventually converge. Therefore, when an equation diverges, you can try increasing the value of MODOVERFLOW. You might thereby prevent the equation from meeting the divergence test in a situation where a successful solution can actually be attained.

Faster Divergence During Development

While you are developing a model, you can sometimes save time by using a small value for MODOVERFLOW. When Oracle OLAP is performing many iterations over a particular simultaneous block, a smaller value of MODOVERFLOW can cause rapid divergence of that block. When you set the MODOVERFLOW option to `CONTINUE`, execution of the model will continue when the divergence occurs, and you can concentrate on debugging the other blocks in the model. After you have debugged the model, you can use a larger value for MODOVERFLOW.

Examples

Example 17–26 Using the Default MODOVERFLOW Value

The following statements specify a trace for a model called `income.est`, limit a dimension, and run the model.

```
MODTRACE = YES
LIMIT division TO 'Camping'
income.est budget
```

These statements produce the following output.

```
(MOD= INCOME.EST) BLOCK 1: SIMULTANEOUS
(MOD= INCOME.EST) ITERATION 1: EVALUATION
(MOD= INCOME.EST) selling = marketing * 3
(MOD= INCOME.EST) BUDGET (LINE SELLING MONTH 'JAN97' ITER 1) = 3
  ...
(MOD= INCOME.EST) BUDGET (LINE SELLING MONTH 'JAN97' ITER 2) = -997
   ...
(MOD= INCOME.EST) BUDGET (LINE SELLING MONTH 'JAN97' ITER 3) = 6.00902708124
   ...
(MOD= INCOME.EST) BUDGET (LINE SELLING MONTH 'JAN97' ITER 49) = 34.2715693388
   ...
(MOD= INCOME.EST) BUDGET (LINE SELLING MONTH 'JAN97' ITER 50) = -7.22300601861
```

In the trace, you can see the results that were calculated for the `Selling` line item in the first three iterations and the forty-ninth and fiftieth iterations over a block of simultaneous equations. The block failed to converge after 50 iterations.

The value of MODOVERFLOW is its default value of `3`. This means that for an equation to meet the divergence test, its comparison value must be greater than `1000`.

Example 17–27 Speeding Up the Divergence

The following statements change the MODOVERFLOW setting and run the `income.est` model.

```
MODOVERFLOW = 2
income.est budget
```

With MODOVERFLOW set to `2`, any comparison value of more than 100 meets the test for divergence. In this example, the equation for `Selling` meets the test in the second iteration. In the second iteration, Oracle OLAP calculates the comparison value for `Selling` as follows.

```
(-997 - 3) / (3 + 1) = 250
```

Since this comparison value is greater than 100, the equation for `Selling` diverges in the second iteration.

MODSIMULTYPE

The MODSIMULTYPE option specifies the method to use for solving simultaneous blocks in a model.

> **See:** Table 17–1, "Model Options" on page 17-23 for descriptions of all of the options that control the solution of simultaneous blocks.

Data type

ID

Syntax

MODSIMULTYPE = {'AITKENS'|'GAUSS'}

Arguments

'AITKENS'
Oracle OLAP uses the Aitkens delta-squared solution method. In the first two of every three iterations over a block of simultaneous equations, the equations are solved using the values from the previous iteration, and the results are tested for convergence and divergence. In every third iteration, the results are obtained not by solving the equations, but by making a next-guess calculation. This calculation uses the results of the previous three iterations. The results of the guesses are not tested for convergence and divergence, and the solution always continues to the next iteration. (Default)

'GAUSS'
Oracle OLAP uses the Gauss-Seidel solution method. Equations in a simultaneous block are solved in each iteration over the block. The results are tested for convergence and divergence in each iteration.

Notes

Solving Simultaneous Blocks

Oracle OLAP uses an iterative method to solve the equations in a simultaneous block. In each iteration, except the next-guess iterations in an Aitkens solution, a comparison value is calculated from the result of the current iteration and the result of the previous iteration. When the comparison value falls within a specified tolerance (see the MODTOLERANCE option), the equation is considered to have converged. When the comparison value is too great (see the MODOVERFLOW option), the equation is considered to have diverged and solution of the block ends.

When all equations in a block converge, the block is considered solved. When any equation diverges or when any equation fails to converge after a specified number of iterations (see the MODMAXITERS option), solution of the block (and of the model) fails and Oracle OLAP generates an error.

Next-Guess Calculation

The Aitkens method requires three values to perform a next-guess calculation. Therefore, in the first three iterations over a simultaneous block, Oracle OLAP solves the equations. The fourth iteration is a next-guess iteration that uses the results from the first three iterations in its calculation.

Thereafter, every third iteration is a next-guess iteration that calculates results by using the previous guess and the equation results from the intervening two iterations. For example, the seventh iteration makes a next-guess calculation that is based on the guess from the fourth iteration and the equation results from the fifth and sixth iterations.

Memory Required

The Aitkens method usually speeds convergence, and it generally produces more accurate results than the Gauss-Seidel method. However, the Aitkens method requires more memory because the results of three previous iterations are stored.

In general, you should use the Aitkens method. You should use the Gauss-Seidel method only when limited memory is a problem on your system.

Handling NA Values

In calculating equation results and making next-guess calculations, Oracle OLAP observes the setting of the NASKIP2 option. NASKIP2 controls how `NA` values are handled when `+` (plus) and `-` (minus) operations are performed. The setting of NASKIP2 is important when you specify a solution variable that contains `NA` values. Since the values in the solution variable are used as the initial values in the first iteration over a simultaneous block, the results of the equations might be `NA` when there are `NA` values in the solution variable. An `NA` result in the first iteration might also produce `NA` results in later iterations. Therefore, to avoid obtaining `NA` for the results, you can make sure that the solution variable does not contain `NA` values or you can set NASKIP2 to `YES` before running the model.

Data Type Problems

A simultaneous equation might fail to converge when it assigns data to a variable with an INTEGER data type or when you specify a solution variable with an INTEGER data type for a dimension-based model. Oracle OLAP converts the data to decimal values when it calculates the equation in each iteration, but the results are stored in the INTEGER variable between iterations. This has the effect of rounding the values and thereby interfering with a progression toward convergence.

Function Problems

A simultaneous equation might fail to converge when it contains a function that produces rounded values (such as INSTRB or ROUND) or when it contains a function that introduces discontinuities in the data (such as MAX or MIN).

Starting-Value Problems

The solution of a simultaneous block is sensitive to starting values. For example, when a model has a proportional relationship between two model values, then starting values close to zero will inhibit convergence. You should thus attempt to use starting values that are reasonable for the equations being solved.

Order of Equations

The solution of a simultaneous block is also sensitive to the order of the equations. When you compile a model, the model compiler determines an optimal equation order that is based on the dependencies among the equations.

To force the equations in a simultaneous block to be solved in a particular order, you can write the equations in the desired order and set the MODINPUTORDER option to `YES` before compiling the model. When MODINPUTORDER is `YES`, the model compiler leaves the equations in a simultaneous block in the order in which they appear in the model.

By placing simultaneous equations in a particular order and setting MODINPUTORDER to `YES` before compiling the model, you might be able to encourage convergence in some models. In general, however, it is preferable to rely on the model compiler to order the equations.

Producing an Execution Report

After running a model, you can use the MODEL.XEQRPT program to produce a report about the execution of the model.

Examples

Example 17–28 Economizing on Memory Requirements

When a model named `budget98` is a complex model that will iterate over a large number of dimension values in a simultaneous block, you can economize on the memory requirements of the model solution by using the Gauss-Seidel method.

The following statements specify the Gauss-Seidel method and run the model.

```
MODSIMULTYPE = 'GAUSS'
budget98 budget
```

MODTOLERANCE

The MODTOLERANCE option is used in testing whether each equation in a simultaneous block of a model has converged. MODTOLERANCE determines how closely the results of an equation must match between successive iterations for the equation to be considered to have converged.

Data type

INTEGER

Syntax

MODTOLERANCE = {*n*|3}

Arguments

n

An integer value to use in testing for convergence. As Oracle OLAP calculates each equation in a simultaneous block, it constructs a comparison value that is based on the results of the equation for the current iteration and the previous iteration. When the comparison value passes a tolerance test, the equation is considered to have converged.

The comparison value that is tested is as follows.

```
(thisResult - prevResult) / (prevResult+ MODGAMMA)
```

where *thisResult* is the result of this iteration and *prevResult* is the result of the previous iteration

In the preceding calculation, MODGAMMA is an INTEGER option that controls the degree to which the comparison value represents the absolute amount of change between iterations versus the proportional change. The default value of MODGAMMA is 1.

In the tolerance test, Oracle OLAP tests whether the comparison value is less than `10` to the negative power of MODTOLERANCE. The calculation for this test is as follows.

```
Comparison value  <  10**-MODTOLERANCE
```

An equivalent way of writing this calculation is as follows.

```
Comparison value  < (1 / (10**MODTOLERANCE))
```

For the equation to be considered to have converged, the comparison value must meet the test described earlier. The default value of MODTOLERANCE is `3`. With this default, the comparison value meets the test when it is less than `0.001`.

Notes

Failure to Converge

When an equation fails to converge after a specified number of iterations, an error occurs. The MODMAXITERS option controls the maximum number of iterations that are attempted. The MODERROR option controls the action that Oracle OLAP takes when an error occurs.

Precision of Results

Since MODTOLERANCE controls how closely results of an equation must match between iterations, it therefore controls the precision of the results of the solution. A small value of MODTOLERANCE will result in less precision, while a large value will provide more precision.

Large and Small Values

When a model contains some equations with large values and some equations with very small values, it might be preferable to increase the value of the MODGAMMA option rather than decreasing MODTOLERANCE. By increasing MODGAMMA, you might be able to force equations with small values to converge more quickly while retaining the precision of equations with large values.

Faster Convergence During Development

While you are developing a model, you might want to use a small value for MODTOLERANCE. While this gives less precise results, the model equations will converge more quickly. After you have debugged the model, you can increase the value of MODTOLERANCE and thereby increase the precision of the final results.

Options for Controlling the Solution of Simultaneous Blocks

For a list of all the options that you can use to control the solution of simultaneous blocks, see Table 17–1, "Model Options" on page 17-23.

Examples

Example 17–29 Using the Default MODTOLERANCE Value

The following statements specify a trace for a model called `income.plan`, specify that the Gauss-Seidel method should be used for solving simultaneous blocks, limit a dimension, and run the model.

```
MODTRACE = YES
MODSIMULTYPE = 'GAUSS'
LIMIT division TO 'Camping'
income.plan budget
```

These statements produce the following output.

```
(MOD= INCOME.PLAN) BLOCK 1: SIMULTANEOUS
(MOD= INCOME.PLAN) ITERATION 1: EVALUATION
(MOD= INCOME.PLAN) marketing = .15 * net.income
(MOD= INCOME.PLAN) BUDGET(LINE MARKETING MONTH 'JAN97' ITER 1) = 11887.403671736
  ...
(MOD= INCOME.PLAN) BUDGET(LINE MARKETING MONTH 'JAN97' ITER 6) = 73379.713232251
    ...
(MOD= INCOME.PLAN) BUDGET(LINE MARKETING MONTH 'JAN97' ITER 7) = 73474.784648631
    ...
(MOD= INCOME.PLAN) BUDGET(LINE MARKETING MONTH 'JAN97' ITER 8) = 73446.025848156
(MOD= INCOME.PLAN) END BLOCK 1
```

In the trace, you can see the results that were calculated for the `Marketing` line item in the final three iterations over a block of simultaneous equations.

MODTOLERANCE is set to its default value of `3`. This means that for an equation to pass the convergence test, its comparison value must be less than `0.001`. In the seventh iteration, Oracle OLAP calculates the comparison value for `Marketing` as follows.

```
(73474.784648631100 - 73379.713232251300) / (73379.713232251300 + 1) = 0.0013
```

This comparison value is greater than `0.001`, so it did not pass the test for convergence.

In the eighth iteration, Oracle OLAP calculated the comparison value as follows.

```
(73446.025848156700 - 73474.784648631100) /(73474.784648631100 + 1) = 0.0004
```

Since this comparison value is less than `0.001`, it passed the convergence test.

Example 17–30 Setting MODTOLERANCE to Speed Up the Convergence of a Model

The following statements change the MODTOLERANCE value and run the `income.bud` model.

```
MODTOLERANCE = 2
income.plan budget
```

With MODTOLERANCE set to `2`, any comparison value of less than `0.01` will pass the test for convergence. In this example, the equation for `Marketing` passes the test in the seventh iteration.

MODTRACE

The MODTRACE option controls whether each equation in a model is recorded in a file during execution of the model. MODTRACE is used primarily as a debugging tool to uncover problems by tracing the execution of a model.

Data type

BOOLEAN

Syntax

MODTRACE = {YES|NO}

Arguments

YES
Oracle OLAP sends the text of each model equation to the current outfile before calculating the model equation, and then sends the results of the calculation to the current outfile.

When you have used the DBGOUTFILE command to specify a debugging file, Oracle OLAP sends MODTRACE output to the debugging file instead of the current outfile.

NO
Oracle OLAP does not send the text of model equations and results to a file while a model executes. (Default)

Notes

Previewing the Solution Order
MODTRACE sends the equations of a model to the current outfile in the order in which they are being solved. Before you run the model, you might want to use the MODEL.COMPRPT program to get a preview of the solution order. A preview can be especially helpful when the model is large and complex. The `MODEL.COMPRPT` program, which you can run after compiling a model, produces a report that shows how the compiler has organized the model equations into blocks and the order in which the blocks and equations will be solved.

Understanding Trace Information

MODTRACE shows the name of the current model on each line of the trace. The trace includes the following types of lines.

- *Block*. A block line gives the block number and block type of the block that is about to be executed. The type of block can be simple, step-forward, step-backward, or simultaneous. For a step-forward or step-backward block, the block line specifies the dimension being stepped over. For a simultaneous block with a cross-dimensional dependency, the block line specifies the dimensions involved in the dependency. See MODEL for information on blocks in a model.
- *Iteration*. These lines occur in simultaneous blocks and specify the number of the iteration that is about to be performed for the current block. When you are using the Aitkens solution method, the next-guess iterations are identified. (The MODSIMULTYPE option determines the solution method being used.)
- *Equation*. The equation that is about to be calculated.
- *Results*. A results line follows each equation line and shows the results assigned by the equation. It shows the variable to which the results were assigned and the current value of each model dimension. In a simultaneous block, it also shows the current iteration number. For example, when `actual` is the solution variable and the model dimensions are `line` and `month`, a results line in a simultaneous block might look like the following one.

```
(MOD= INCOME.CALC) ACTUAL (LINE OPR.INCOME MONTH 'JAN96'
   ITER 1) = 108.9600000
```

Dimension-Based Equations

When you run a model that contains dimension-based equations, Oracle OLAP automatically loops over all the dimensions of the solution variable. In the trace, the results lines show the current value of each dimension listed in a DIMENSION statement, but they do not show the current values of extra dimensions that are not listed in DIMENSION statement. (See DIMENSION (in models) for more information about using DIMENSION statements.)

Thus, when the model dimensions are `line` and `month`, and when the solution variable is dimensioned by `line`, `month`, and `division`, the current value of `division` is not shown in the results lines. Oracle OLAP executes the entire model for the first value in the status of `division`, then for the second value in the status, and so on.

When you run a model that assigns values to variables, Oracle OLAP automatically loops over all the dimensions (or bases of a composite) of those variables. In this case, the current value of each of the variable's dimensions is shown in the trace.

Additional Tool for Debugging

The INFO function lets you obtain specific items of information about the structure of the compiled model and the solution status of a model you have run. See INFO (MODEL).

Examples

Example 17–31 Debugging a Model with MODTRACE

The following statements define a model named `income.budget`.

```
DEFINE income.budget MODEL
LD Model for estimating budget items
MODEL
DIMENSION line month
Opr.Income = Gross.Margin - Marketing
Gross.Margin = Revenue - Cogs
Revenue = LAG(Revenue, 1, month) * 1.02
Cogs = LAG(Cogs, 1, month) * 1.01
Marketing = LAG(Opr.Income, 1, month) * 0.20
END
```

This model estimates budget line items on an income statement. The model equations are based on a `line` dimension.

The following statements compile the model and run the MODEL.COMPRPT program.

```
COMPILE income.budget
MODEL.COMPRPT income.budget
```

The `MODEL.COMPRPT` statement produces the following compilation report.

```
MODEL INCOME.BUDGET <LINE MONTH>
                   BLOCK 1 (SIMPLE)
INCOME.BUDGET  4:  revenue = lag(revenue, 1, month) * 1.02
INCOME.BUDGET  5:  cogs = lag(cogs, 1, month) * 1.01
INCOME.BUDGET  3:  gross.margin = revenue - cogs
                    BLOCK 2 (STEP-FORWARD <MONTH>)
INCOME.BUDGET  6:   marketing = lag(opr.income, 1, month) * 0.20
INCOME.BUDGET  2:   opr.income = gross.margin - marketing
                    END BLOCK 2
                   END BLOCK 1
```

When you want to debug this model, you can trace its execution, line by line, by turning on MODTRACE before running the model.

The following statements limit dimensions, specify tracing, and run the model.

```
LIMIT month TO 'Jan97' TO 'Mar97'
LIMIT division TO 'Camping'
MODTRACE = YES
income.budget budget
```

These statements produce the following line-by-line results.

```
(MOD= INCOME.BUDGET) BLOCK 1: SIMPLE
(MOD= INCOME.BUDGET) revenue = lag(revenue, 1, month) * 1.02
(MOD= INCOME.BUDGET) BUDGET (LINE REVENUE MONTH 'JAN97') = 744491.1966
(MOD= INCOME.BUDGET) BUDGET (LINE REVENUE MONTH 'FEB97') = 759381.020532
(MOD= INCOME.BUDGET) BUDGET (LINE REVENUE MONTH 'MAR97') = 774568.64094264
(MOD= INCOME.BUDGET) cogs = lag(cogs, 1, month) * 1.01
(MOD= INCOME.BUDGET) BUDGET (LINE COGS MONTH 'JAN97') = 382386.2323
(MOD= INCOME.BUDGET) BUDGET (LINE COGS MONTH 'FEB97') = 386210.094623
(MOD= INCOME.BUDGET) BUDGET (LINE COGS MONTH 'MAR97') = 390072.19556923
(MOD= INCOME.BUDGET) gross.margin = revenue - cogs
(MOD= INCOME.BUDGET) BUDGET (LINE GROSS.MARGIN MONTH 'JAN97') = 362104.9643
(MOD= INCOME.BUDGET) BUDGET (LINE GROSS.MARGIN MONTH 'FEB97') = 373170.925909
(MOD= INCOME.BUDGET) BUDGET (LINE GROSS.MARGIN MONTH 'MAR97') = 384496.44537341
(MOD= INCOME.BUDGET) BLOCK 2 STEP-FORWARD <MONTH>
(MOD= INCOME.BUDGET) marketing = lag(opr.income, 1, month) * 0.20
(MOD= INCOME.BUDGET) BUDGET (LINE MARKETING MONTH 'JAN97') = 39938.192
(MOD= INCOME.BUDGET) opr.income = gross.margin - marketing
(MOD= INCOME.BUDGET) BUDGET (LINE OPR.INCOME MONTH 'JAN97') = 322166.7723
(MOD= INCOME.BUDGET) marketing = lag(opr.income, 1, month) * 0.20
(MOD= INCOME.BUDGET) BUDGET (LINE MARKETING MONTH 'FEB97') = 64433.35446
(MOD= INCOME.BUDGET) opr.income = gross.margin - marketing
(MOD= INCOME.BUDGET) BUDGET (LINE OPR.INCOME MONTH 'FEB97') = 308737.571449
(MOD= INCOME.BUDGET) marketing = lag(opr.income, 1, month) * 0.20
(MOD= INCOME.BUDGET) BUDGET (LINE MARKETING MONTH 'MAR97') = 61747.5142898
(MOD= INCOME.BUDGET) opr.income = gross.margin - marketing
(MOD= INCOME.BUDGET) BUDGET (LINE OPR.INCOME MONTH 'MAR97') = 322748.93108361
(MOD= INCOME.BUDGET) END BLOCK 2
(MOD= INCOME.BUDGET) END BLOCK 1
```

In Block 1, which is a simple block, Oracle OLAP solved the equations one at a time, looping over the three values in the status of `month` as it solved each equation. In Block 2, which is a step-forward block over the `month` dimension, Oracle OLAP stepped over the values in the status of `month`, solving all the equations in the block for each month in turn.

18 MONITOR to NVL2

This chapter contains the following OLAP DML statements:

MONITOR

The MONITOR command records data on the performance cost of each line in a specified program. To get meaningful information from MONITOR, your session must be the only one running in Oracle OLAP. Furthermore, the accuracy of the results of MONITOR decreases as more processes are started on the host computer.

You use the MONITOR command first to specify a program for monitoring; then you run the program and use MONITOR again to obtain the results. When the program executes a given line repeatedly, MONITOR records the cumulative cost of all the executions on the single line of its monitor list that is devoted to that program line.

A line of code is considered to have a high performance cost when it takes a long time to execute. Use the TRACKPRG command to identify programs that have relatively high costs and then use MONITOR to identify the time-consuming lines within those programs. When you wish, you can use both TRACKPRG and MONITOR simultaneously.

Syntax

MONITOR ON [ALL|*awlist*] | OFF | INIT | FILE [[APPEND] *file-name*] | RESET

Arguments

ON
Starts looking for the specified programs to be run so that Oracle OLAP can gather line-by-line timing data in a monitor list. (It continues the current monitoring process without interruption when monitoring is already on, or resumes with a gap when monitoring was off.)

When you do not specify either ALL or *awlist*, the default is the program specified in the last `MONITOR ON` statement that did specify one. When there was no such command in your current session, no data is collected and no error is produced.

ALL
Specifies that all programs are monitored.

awlist
The fully-qualified name of one or more analytic workspaces (optionally separated by commas) whose programs you want monitored.

OFF
Stops monitoring the specified program and freezes any timing data currently in the monitor list. This lets you immediately, or later in your session, send the list to the current outfile or to a text file.

RESET
(Useful only when monitoring is on.) Retains the program name that is currently specified for monitoring and the Oracle OLAP memory that is allocated for the current monitor list, but discards any timing data currently in the list. In addition, RESET causes MONITOR to again begin waiting for you to run the same program. When you do, MONITOR automatically gathers new timing data into a new monitor list for the same program in the same memory allocation as before.

INIT
(Useful only when monitoring is on.) Initializes the monitoring environment. Initialization consists of discarding the program name and the timing data associated with the current monitor list, and releasing the Oracle OLAP memory previously used for that list so it can be used for other purposes or for collecting new data in a new monitor list.

FILE
Specifies that the timing data that is currently in the monitor list is sent to a file.

APPEND
Specifies that the timing data is appended to an existing file. When you omit this argument, the new output replaces the current contents of the file.

file-name
A text expression that is the name of the file to receive the output. Unless the file is in the current directory, you must include the name of the directory object in the name of the file. When *file-name* is specified, the data is sent to the named text file. FILE has no effect on the monitor list, so you can send the same list repeatedly to different destinations. When *file-name* is omitted, Oracle OLAP sends the timing data that is currently in the monitor list to the current outfile.

Note: Directory objects are defined in the database, and they control access to directories and file in those directories. You can use the CDA command to identify and specify a current directory object. Contact your Oracle DBA for access rights to a directory object where your database user name can read and write files.

Notes

Monitor List

Each entry (that is, line) in the monitor list focuses on the execution of a single program line, regardless of how many times it is executed. Each entry is divided into the following four sections:

- Cumulative total time of all executions in seconds, in columns 1 through 11
- Number of times executed, in columns 12 through 18
- Line number, in columns 19 through 23
- Text of the line, in column 24 and subsequent columns

Here is a sample of MONITOR output with column numbers included for reference.

```
12345678901234567890123456789012345678901234567890

60            1    1 push name
30            1    2 trap on GETOUT noprint
51            1    3 limit name to obj(type) eq 'DIMENSION'
0             1    4 for name
0             0    5    do
450           6    6    limit &name to ALL ifnone BYPASS
0             0    7 BYPASS:
0             0    8    doend
0             0    9 GETOUT:
0             1   10 pop name
```

The following is the full description of the program used for the preceding output. Note that in the output, the line with the LIMIT command is truncated because it is too long to fit.

```
DEFINE allstat PROGRAM
LD Program to set the status of all dimensions in the analytic workspace to ALL
PROGRAM
PUSH NAME
TRAP ON getout NOPRINT
LIMIT NAME TO OBJ(TYPE) EQ 'Dimension' IFNONE getout
FOR NAME
   DO
   LIMIT &NAME TO ALL IFNONE bypass
bypass:
   DOEND
getout:
POP NAME
END
```

Attaching, Detaching, and Reattaching Analytic Workspaces

When Oracle OLAP executes a program in an analytic workspace that has been attached, detached, a new block of data is collected.

Truncated Statement Lines

When a program line is too long, the MONITOR output truncates it. Continuation lines do not appear in the output.

Producing a Report

When you want to produce an Oracle OLAP report from the timing data in the MONITOR file, you can write a program that uses the FILEREAD command to read the data into an Oracle OLAP variable, and then use Oracle OLAP reporting capabilities to produce a report. You can use the `TRACKREPORT` program as a model. However, keep in mind that the `TRACKREPORT` program was written to produce a report on TRACKPRG output, not MONITOR output.

Bracketing Lines

When you just want to time a particular line or group of lines in a program, you can insert `MONITOR ON` and `MONITOR OFF` statements in the program to bracket just the line or lines in which you are interested.

Nested Programs

When you do not want to run a nested program by itself, you can specify its name in a `MONITOR ON` statement and then run the program that calls it. MONITOR will gather timing data only for the specified (nested) program. When the specified program is called more than once, for each program line, MONITOR will accumulate the total seconds taken by all the times the line was run and provide the number of times it was run.

When you just want to time a particular execution of a nested program that is called more than once, you can insert `MONITOR ON` and `MONITOR OFF` statements in the calling program to bracket the single call in which you are interested.

Very Small Programs

You might not be able to reproduce the results exactly for very small programs. When the CPU interrupts processing to do other tasks, that time is a greater percentage of the total execution time.

Unit of Measure

The MONITOR and TRACKPRG commands use milliseconds as the unit for recording execution time. The execution time does not include time spent on I/O and time spent waiting for the next statement.

Examples

Example 18–1 Collecting Timing Data Using MONITOR

In this example, MONITOR is used to collect timing data on the execution of the individual lines of code in `prog1` and then to send the data to a text file. The `MONITOR ON` statement is then used to discard the `prog1` timing data and start collecting data on `prog2`. After the data for `prog2` is sent to a second file, `MONITOR INIT` is used to discard the current monitor list and release the memory used for it.

```
MONITOR ON prog1
prog1
MONITOR FILE prog1.mon
MONITOR ON prog2
prog2
MONITOR OFF
MONITOR FILE prog2.mon
MONITOR INIT
```

MONTHABBRLEN

The MONTHABBRLEN option specifies the number of characters to use for abbreviations of month names that are stored in the MONTHNAMES option. You can specify how many characters to use for abbreviating particular month names when you specify the `<MT>`, `<MTXT>`, and `<MTXTL>` formats with the DATEFORMAT text option or the VNF (value name format) command used for a dimension of type dimensions of type DAY, WEEK, MONTH, QUARTER, or YEAR.

Data type

TEXT

Syntax

MONTHABBRLEN = *specification* [;|, *specification*]...

where:

specification is a text expression that has the following form:

startpos [- *endpos*] : *length*

Arguments

startpos [-endpos]

Numbers that represent the first and last months whose abbreviation length is defined by *length*. These numerical positions apply to the corresponding lines of text in the MONTHNAMES option. You can specify these ranges of values in reverse order, *endpos* [-*startpos*], if you prefer.

The MONTHNAMES option can have more than 12 lines, so you can specify *startpos* and *endpos* greater than 12 in the setting of MONTHABBRLEN. When you specify a range where neither *startpos* nor *endpos* has a corresponding text value in the MONTHNAMES option, MONTHABBRLEN has no text values to abbreviate for that range. When you later change your month names list so that *startpos* is valid, the specified abbreviation is applied.

length

A number that specifies the length in characters (not bytes) of abbreviated month names.

Notes

Abbreviation Lengths

You can define many different groups of months, each with different abbreviation lengths. When you do so, separate the groups with a comma or a semicolon as shown in the syntax.

Default Abbreviations

When you do not specify an abbreviation length for a given position in the MONTHNAMES option, or when you explicitly set a given position to zero, the default abbreviation is used. The default abbreviations are one character for `<MT>` and three characters for `<MTXT>` and `<MTXTL>`. Abbreviations are never used when you have designated the full name specifications `<MTEXT>` and `<MTEXTL>`.

Ambiguous Month Names

You can use MONTHABBRLEN to interpret ambiguous names, for example, whether `A` stands for `April` or `August`. When the MONTHABBRLEN for April was `1` and for August was `2`, then `A` would always match `April`, and it would require at least `Au` to match `August`. This does *not* depend on the order of `April` and `August` in the year; it would work the same way when the two months were reversed. If, on the other hand, the MONTHABBRLEN for each of these was `2`, then `A` would not match either one, and you would have to enter at least `Ap` or `Au` to get a match.

Examples

Example 18–2 Specifying Month Abbreviations

The following MONTHABBRLEN setting specifies that the first 10 months of the year are abbreviated to one character and the last 2 months are abbreviated to two characters.

```
MONTHABBRLEN = '1-10:1, 11-12:2'
SHOW CONVERT ('2 August 2005' DATE)
```

These statements product the following result, with August abbreviated to the letter A.

```
02A05
```

MONTHNAMES

The MONTHNAMES option holds the list of valid names for months that is used in handling values with a DATE data type and values of dimensions of type DAY, WEEK, MONTH, QUARTER, and YEAR. The list of names is used to interpret dates that are entered and to format dates that are displayed or converted to text values.

Data type

TEXT

Syntax

MONTHNAMES = *name-list*

Arguments

name-list
A multiline text expression that lists the names of the 12 months of the year. Each month name occupies a separate line. Regardless of which month you are treating as the first month of the year, the list must begin with the name for January. The default value is the list of English month names, all in capital letters.

Notes

Extra Sets of Names

You can include more than 1 set of 12 names in your list. Any name in the list is considered a valid name for input. The thirteenth name is a synonym for the first name, the fourteenth name is a synonym for the second name, and so on.

How MONTHNAMES Is Used

The MONTHNAMES list is used when you enter a date that includes a month name or abbreviation. See DATEORDER for a discussion of methods for entering DATE values.

The MONTHNAMES list is also used when you display or convert a date using the `<MT>`, `<MTXT>`, `<MTXTL>`, `<MTEXT>`, or `<MTEXTL>` formats. These formats are specified in the DATEFORMAT option. When you have more than one set of month names, Oracle OLAP chooses the synonym whose number of characters and capitalization pattern best match the DATEFORMAT specification.

Abbreviations

You can set the MONTHABBRLEN option to control the number of characters used for abbreviations of month names.

Examples

Example 18–3 Specifying Two Sets of Month Names

The following statement creates two sets of month names, one in uppercase English and the second in lowercase French.

```
MONTHNAMES = -
'JANUARY -
...
DECEMBER -
janvier -
...
decembre'
```

Example 18–4 Specifying English Month Names

The following statements define a DATE variable, assign a value to that variable, assign a setting to DATEFORMAT, and send the output to the current outfile. The DATEFORMAT value includes `<MTEXT>`, which specifies uppercase, so the English month names are used.

```
DEFINE datevar DATE
datevar = '27feb98'
DATEFORMAT = '<MTEXT> <D>, <YYYY>'
SHOW datevar
```

These statements produce the following output.

```
FEBRUARY 27, 1998
```

Example 18–5 Specifying French Month Names

The following statements assign a new setting to DATEFORMAT and send the output to the current outfile. The DATEFORMAT value includes `<MTEXTL>`, which specifies lowercase, so the French month names are used.

```
DATEFORMAT = 'le <D> <MTEXTL> <YYYY>'
SHOW datevar
```

These statements produce the following output.

```
le 27 fevrier 1998
```

MONTHS_BETWEEN

The MONTHS_BETWEEN function calculates the number of months between two dates. When the two dates have the same day component or are both the last day of the month, then the return value is a whole number. Otherwise, the return value includes a fraction that considers the difference in the days based on a 31-day month. The return value is positive when the first date is later than the second date, and negative when the first date is earlier than the second date.

Return Value

NUMBER

Syntax

MONTHS_BETWEEN(*datetime_expression1*, *datetime_expression2*)

Arguments

datetime-expression1
One expression that has the DATETIME data type, or a text expression that specifies a date.

datetime-expression2
A second expression that has the DATETIME data type, or a text expression that specifies a date.

Examples

Example 18–6 Calculating the Number of Months Between Dates

The following statement calculates the number of months between March 26, 2004, and July 6, 2001.

```
SHOW months_between('06Jul2005' '17Jul2003')
23.65
```

Example 18–7 Last Days

The return value is a whole number when both dates are the last day of the month.

```
SHOW months_between('29Feb2000', '30Sep2000')
-7.00
```

MOVE

The MOVE command moves an object name to a new position in the NAME dimension of a workspace. The reorganizing effect of the MOVE command on the workspace is cosmetic. That is, no physical changes take place in workspace storage. Users often reorganize workspace objects so the output from DESCRIBE is easier to read.

Syntax

MOVE *name...* {FIRST|<u>LAST</u>|{BEFORE|AFTER} *name2*} [AW *workspace*]

Arguments

name...
The names of one or more objects to move. You can specify the names individually, or use one of the following forms to specify a group of names:

> *name* TO *name*
> FIRST *n*
> LAST *n*
> *boolean-expression* (dimensioned by NAME)

You can specify a qualified object name to indicate the attached workspace in which the object resides. As an alternative, you can use the AW argument to specify the workspace. Do not use both.

When you do not use a qualified object name or the AW argument to specify a workspace, Oracle OLAP looks for the object in the current workspace.

FIRST
LAST
The logical position in the NAME dimension to which Oracle OLAP moves the objects specified by the *name* argument. Specifying FIRST moves the objects to the beginning of the NAME dimension. Specifying LAST (the default) moves the names to the end of the NAME dimension.

BEFORE *name2*
AFTER *name2*
The position before or after a particular object (*name2*) to which Oracle OLAP moves the objects specified by the *name* argument.

AW workspace
The name of an attached workspace in which you wish to move the object. When you do not use a qualified object name or the AW argument to specify a workspace, objects are moved in the current workspace.

Notes

Alphabetizing Your Objects

You can arrange your workspace objects alphabetically with the following statements, which work on the NAME dimension.

```
SORT NAME A NAME
MOVE CHARLIST(NAME) FIRST
```

Examples

Example 18–8 Moving a Relation

This example shows how to move the relation `desc.product` after `product`. The OLAP DML statement

```
SHOW CHARLIST(NAME)
```

produces the following list (annotation has been added).

```
product           <--- Position of product
district
division
line
region
marketlevel
market
month
year
quarter
desc.product      <--- Old position of desc.product
region.district
division.product
...
```

The following statements

```
MOVE desc.product AFTER product
SHOW CHARLIST(NAME)
```

change the workspace order and produce the following list (annotation has been added).

```
product            <--- Position of product
desc.product       <--- New position of desc.product
district
division
line
region
marketlevel
market
month
year
quarter
region.district
division.product
...
```

MOVINGAVERAGE

The MOVINGAVERAGE function (abbreviated MVAVG) computes a series of averages for the values of a dimensioned variable or expression over a specified dimension. For each dimension value in status, MOVINGAVERAGE computes the average of the data in the range specified, relative to the current dimension value.

When the data being averaged has only one dimension, MOVINGAVERAGE produces a single series of averages, one for each dimension value in status. When the data has dimensions other than the one being averaged over, MOVINGAVERAGE produces a separate series of averages for each combination of values in the status list of the other dimensions.

Return Value

DECIMAL

Syntax

MOVINGAVERAGE(*expression*, *start*, *stop*, *step*, -

  [*dimension* [STATUS|*limit-clause*]])

Arguments

expression
A numeric variable or calculation whose values you want to average; for example, `units` or `sales-expense`.

start
A whole number that specifies the starting point of the range over which you want to average. The range is specified relative to the current value of *dimension*. Zero (`0`) refers to the current value, and `-1` refers to the value preceding the current value. A comma is required before a negative start number.

Each average is based on data for a specified range of dimension values preceding, including, or following the one for which the average is being calculated. To count the values in the range, MOVINGAVERAGE uses the default status, unless you use the STATUS keyword or the *limit-clause* argument to specify a different dimension status.

stop
A whole number that specifies the ending point of the range over which you want to average. A negative *stop* number must be preceded by a comma.

step
A positive whole number that specifies whether to average over every value in the range, every other value, every third value, and so on. A value of `1` for *step* means average over every value. A value of `2` means average over the first value, the third value, the fifth value, and so on. For example, when the current month is `Jun96` and the *start* and *stop* values are `-3` and `3`, a *step* value of `2` means average over `Mar96`, `May96`, `Jul96`, and `Sep96`.

dimension
The dimension over which the moving average is calculated. While this can be any dimension, it is typically a hierarchical time dimension of type TEXT that is limited to a single level (for example, the month or year level) or a dimension with a type of DAY, WEEK, MONTH, QUARTER, or YEAR.

When *expression* has a dimension with a type of DAY, WEEK, MONTH, QUARTER, or YEAR and you want MOVINGAVERAGE to use that dimension, you can omit the *dimension* argument.

STATUS
Specifies that MOVINGAVERAGE should use the current status list (that is, only the dimension values currently in status in their current status order) when calculating the moving average.

limit-clause
Specifies that MOVINGAVERAGE should use the default status limited by *limit-clause* when calculating the moving average. You can use any valid LIMIT clause (see the entry for the LIMIT command for further information). To specify that MOVINGAVERAGE should use the current status limited by *limit-clause* when calculating the moving average, specify a LIMIT function for *limit-clause.*

Notes

NASKIP Option

MOVINGAVERAGE is affected by the NASKIP option. When NASKIP is set to `YES` (the default), MOVINGAVERAGE ignores `NA` values and returns the average of the values that are not `NA`. Likewise, when some dimension values do not exist for a given range, MOVINGAVERAGE returns the average using whatever values do exist.

Suppose, for example, that `Jan96` is the first month value in the workspace. When the current period being calculated is `Feb95` and the range is `-3` to `-1`, `Jan95` is the only month in the range `-3` to `-1`. The average for `Feb95` therefore uses only the `Jan95` value.

When NASKIP is set to `NO`, MOVINGAVERAGE returns `NA` when any value in the current range has a value of `NA`, or when there are any dimension values that do not exist in the range.

When all data values for a calculation are `NA`, or when no dimension values exist in the specified range, MOVINGAVERAGE returns `NA` for either setting of NASKIP.

Examples

Example 18–9 Calculating a Moving Average

Suppose you have a variable called `sales` that is dimensioned by a hierarchical dimension named `time`, a dimension named `product`, a dimension named `timelevelnames` that contains the names of the levels of `time` (for example, `Quarter` and `Year`), and a relation named `time.levelrels` that relates the values of `time` to the values of `timelevelnames`. Assume also that using the following statements you limit `product` to `Womens - Trousers` and time to quarters from `Q4-1999` to present.

```
LIMIT product TO 'Womens - Trousers'
LIMIT timelevelnames TO 'Quarter'
LIMIT time TO time.levelrels
LIMIT time REMOVE 'Q1-1999' 'Q2-1999' 'Q3-1999'
```

After you have limited `product` and **`sales`**, you issue the following report statement.

```
REPORT DOWN time sales -
HEADING 'Running Yearly\nTotal' MOVINGTOTAL(sales, -4, 0, 1, time, -
     LEVELREL time.levelrels) -
HEADING 'Minimum\nQuarter' MOVINGMIN(sales, -4, 0, 1, time, -
     LEVELREL time.levelrels) -
HEADING 'Maximum\nQuarter' MOVINGMAX(sales, -4, 0, 1, time, -
     LEVELREL time.levelrels) -
HEADING 'Average\nQuarter' MOVINGAVERAGE(sales, -4, 0, 1, time, -
     LEVELREL time.levelrels)
```

The following report was created by the preceding statement.

```
               -----------------------PRODUCT------------------------
               ------------------Womens - Trousers-------------------
                          Running
                          Yearly     Minimum    Maximum    Average
TIME           SALES      Total      Quarter    Quarter    Quarter
-------------- ---------- ---------- ---------- ---------- ----------
Q4-1999         416            1,386        233        480     346.50
Q1-2000         465            1,851        233        480     370.20
Q2-2000         351            1,969        257        480     393.80
Q3-2000         403            2,115        351        480     423.00
Q4-2000         281            1,916        281        465     383.20
Q1-2001         419            1,919        281        465     383.80
Q2-2001         349            1,803        281        419     360.60
Q3-2001         467            1,919        281        467     383.80
Q4-2001         484            2,000        281        484     400.00
Q1-2002         362            2,081        349        484     416.20
Q2-2002         237            1,899        237        484     379.80
Q3-2002         497            2,047        237        497     409.40
Q4-2002         390            1,970        237        497     394.00
```

MOVINGMAX

The MOVINGMAX function (abbreviated MVMAX) returns a series of maximum values of a dimensioned variable or expression over a specified dimension. For each dimension value in status, MOVINGMAX searches the data for the maximum value in the range specified, relative to the current dimension value.

When the variable or expression has only the specified dimension, MOVINGMAX produces a single series of maximum values, one for each dimension value in the status. When the variable or expression has dimensions other than the one specified, MOVINGMAX produces a separate series of maximum values for each combination of values in the status list of the other dimensions

Return Value

DECIMAL

Syntax

MOVINGMAX(*expression*, *start*, *stop*, *step*, [*dimension* [STATUS|*limit-clause*]])

Arguments

expression
A numeric variable or calculation from whose values you want to find the maximum values; for example, `units` or `sales-expense`.

start
A whole number that specifies the starting point of the range over which you want to search. The range is specified relative to the current value of *dimension*. Zero (`0`) refers to the current value, and `-1` refers to the period preceding the current value. A comma is required before a negative *start* number.

Each maximum value is based on data for a specified range of dimension values preceding, including, or following the one for which the maximum value is being returned. To count the values in the range, MOVINGMAX uses the default status, unless you use the STATUS keyword or the *limit-clause* argument to specify a different dimension status.

stop

A whole number that specifies the ending point of the range over which you want to search. A negative *stop* number must be preceded by a comma.

step

A positive whole number that specifies whether to search every value in the range, every other value, every third value, and so on. A value of `1` for step means search every value. A value of `2` means check the first value, the third value, the fifth value, and so on. For example, when the current month is `Jun96` and the *start* and *stop* values are `-3` and `3`, a *step* value of `2` means search the months `Mar96`, `May96`, `Jul96`, and `Sep96` and return the maximum value that occurs in one of those four months.

dimension

The dimension over which the moving maximum is calculated. While this can be any dimension, it is typically a hierarchical time dimension of type TEXT that is limited to a single level (for example, the month or year level) or a dimension with a type of DAY, WEEK, MONTH, Quarter, or YEAR.

When *expression* has a dimension with a type of DAY, WEEK, MONTH, QUARTER, or YEAR and you want MOVINGMAX to use that dimension, you can omit the dimension argument.

STATUS

Specifies that MOVINGMAX should use the current status list (that is, only the dimension values currently in status in their current status order) when calculating the moving maximum.

limit-clause

Specifies that MOVINGMAX should use the default status limited by *limit-clause* when calculating the moving maximum. You can use any valid LIMIT clause (see the entry for the LIMIT command for further information). To specify that MOVINGMAX should use the current status limited by *limit-clause* when calculating the moving maximum, specify a LIMIT function for *limit-clause.*

Notes

NASKIP Option

MOVINGMAX is affected by the NASKIP option. When NASKIP is set to `YES` (the default), MOVINGMAX ignores `NA` values and returns the maximum value. Likewise, when some dimension values do not exist for a given range, MOVINGMAX returns the maximum value using whatever values do exist.

Suppose, for example, that `Jan96` is the first month value in the workspace. When the current period is `Feb96` and the range is `-3` to `-1`, `Jan96` is the only month in the range `-3` to `-1`. The maximum for `Feb96` therefore uses only the `Jan96` value.

When NASKIPis set to `NO`, MOVINGMAX returns `NA` when any value in the current range has a value of `NA` or when there are any dimension values that do not exist in the range.

When all data values for a calculation are `NA`, or when no dimension values exist in the specified range, MOVINGMAX returns `NA` for either setting of NASKIP.

Examples

For an example of calculating maximum sales, see Example 18–9, "Calculating a Moving Average" on page 18-20.

MOVINGMIN

The MOVINGMIN function (abbreviated MVMIN) returns a series of minimum values for the values of a dimensioned variable or expression over a specified dimension. For each dimension value in status, MOVINGMIN searches the data for the minimum value in the range specified, relative to the current dimension value.

When the variable or expression has only the specified dimension, MOVINGMIN produces a single series of minimum values, one for each dimension value in the status. When the variable or expression has dimensions other than the one specified, MOVINGMIN produces a separate series of minimum values for each combination of values in the status list of the other dimensions.

Return Value

DECIMAL

Syntax

MOVINGMIN(*expression*, *start*, *stop*, *step*, [*dimension* [STATUS|*limit-clause*]])

Arguments

expression
A numeric variable or calculation from whose values you want to find the minimum values; for example, UNITS or SALES-EXPENSE.

start
A whole number that specifies the starting point of the range over which you want to search. The range is specified relative to the current value of *dimension*. Zero (`0`) refers to the current value, and `-1` refers to the value preceding the current value. A comma is required before a negative start number.

Each minimum value is based on data for a specified range of dimension values preceding, including, or following the one for which the minimum value is being returned. To count the values in the range, MOVINGMIN uses the default status, unless you use the STATUS keyword or the *limit-clause* argument to specify a different dimension status.

stop
A whole number that specifies the ending point of the range over which you want to search. A negative *stop* number must be preceded by a comma.

step
A positive whole number that specifies whether to search every value in the range, or every other value, or every third value, and so on. A value of `1` for step means search every value. A value of `2` means check the first value, the third value, the fifth value, and so on. For example, when the current month is `Jun96` and the *start* and *stop* values are `-3` and `3`, a *step* value of `2` means search the months `Mar96`, `May96`, `Jul96` and `Sep96` and return the minimum value that occurs in one of those four months.

dimension
The dimension over which the moving minimum is calculated. While this can be any dimension, it is typically a hierarchical time dimension of type TEXT that is limited to a single level (for example, the month or year level) or a dimension with a type of DAY, WEEK, MONTH, Quarter, or YEAR.

When *expression* has a dimension with a type of DAY, WEEK, MONTH, QUARTER, or YEAR and you want MOVINGMIN to use that dimension, you can omit the *dimension* argument.

STATUS
Specifies that MOVINGMIN should use the current status list (that is, only the dimension values currently in status in their current status order) when calculating the moving minimum.

limit-clause
Specifies that MOVINGMIN should use the default status limited by *limit-clause* when calculating the moving minimum. You can use any valid LIMIT clause (see the entry for the LIMIT command for further information). To specify that MOVINGMIN should use the current status limited by limit-clause when calculating the moving minimum, specify a LIMIT function for *limit-clause.*

Notes

NASKIP Option

MOVINGMIN is affected by the NASKIP option. When NASKIP is set to `YES` (the default), MOVINGMIN ignores `NA` values and returns the minimum value. Likewise, when some dimension values do not exist for a given range, MOVINGMIN returns the minimum value using whatever values do exist.

Suppose, for example, that `Jan95` is the first month value in the workspace. When the current period is `Feb95` and the range is `-3` to `-1`, `Jan95` is the only month in the range `-3` to `-1`. The minimum value for `Feb95` therefore uses only the `Jan95` value.

When NASKIP is set to `NO`, MOVINGMIN returns `NA` when any value in the current range has a value of `NA` or when there are any dimension values that do not exist in the range.

When all data values for a calculation are `NA`, or when no dimension values exist in the specified range, MOVINGMIN returns `NA` for either setting of NASKIP.

Examples

For an example of calculating minimum sales, see Example 18–9, "Calculating a Moving Average" on page 18-20.

MOVINGTOTAL

The MOVINGTOTAL function (abbreviated MVTOT) computes a series of totals for the values of a dimensioned variable or expression over a specified dimension. For each dimension value in status, MOVINGTOTAL computes the total of the data in the range specified, relative to the current dimension value.

When the variable or expression has only the specified dimension, MOVINGTOTAL produces a single series of totals, one for each dimension value in the status. When the variable or expression has dimensions other than the one specified, MOVINGTOTAL produces a separate series of totals for each combination of values in the status list of the other dimensions.

Return Value

DECIMAL

Syntax

MOVINGTOTAL(*expression*, *start*, *stop*, *step*, [*dimension* [STATUS|*limit-clause*]])

Arguments

expression
A numeric variable or calculation whose values you want to total; for example, UNITS or SALES-EXPENSE.

start
A whole number that specifies the starting point of the range over which you want to total. The range is specified relative to the current value. Zero (`0`)refers to the current value, and `-1` refers to the value preceding the current value. A comma is required before a negative *start* number.

Each total is based on data for a specified range of dimension values preceding, including, or following the one for which the total is being calculated. To count the values in the range, MOVINGTOTAL uses the default status, unless you use the STATUS keyword or the *limit-clause* argument to specify a different dimension status.

stop
A whole number that specifies the ending point of the range over which you want to total. A negative *stop* number must be preceded by a comma.

step
A positive whole number that specifies whether to total over every value in the range, every other value, every third value, and so on. A value of `1` for *step* means total over every value. A value of `2` means total over the first value, the third value, the fifth value, and so on. When the current month is `Jun96` and the *start* and *stop* values are `-3` and `3`, a *step* value of `2` means total over `Mar96`, `May96`, `Jul96`, and `Sep96`.

dimension
The dimension over which the moving total is calculated. While this can be any dimension, it is typically a time dimension.

When *expression* has a dimension with a type of DAY, WEEK, MONTH, QUARTER, or YEAR and you want MOVINGTOTAL to use that dimension, you can omit the *dimension* argument.

STATUS
Specifies that MOVINGTOTAL should use the current status list (that is, only the dimension values currently in status in their current status order) when calculating the moving total.

limit-clause
Specifies that MOVINGTOTAL should use the default status limited by *limit-clause* when calculating the moving total. You can use any valid LIMIT clause (see the entry for the LIMIT command for further information). To specify that MOVINGTOTAL should use the current status limited by limit-clause when calculating the moving total, specify a LIMIT function for *limit-clause*.

Notes

NASKIP Option
MOVINGTOTAL is affected by the NASKIP option. When NASKIP is set to `YES` (the default), MOVINGTOTAL ignores `NA` values and returns the total of the values that are not `NA`. Likewise, when some dimension values do not exist for a given range, MOVINGTOTAL returns the total using whatever values do exist.

Suppose, for example, that `Jan95` is the first month value in the workspace. When the current period is `Feb95` and the range is `-3` to `-1`, `Jan95` is the only month in the range `-3` to `-1`. The total for `Feb95` therefore uses only the `Jan95` value.

When NASKIPis set to `NO`, MOVINGTOTAL returns `NA` when any value in the current range has a value of `NA` or when there are any dimension values that do not exist in the range.

When all data values for a calculation are `NA`, or when no dimension values exist in the specified range, MOVINGTOTAL returns `NA` for either setting of NASKIP.

Examples

For an example of calculating a moving total sales, see Example 18–9, "Calculating a Moving Average" on page 18-20.

MULTIPATHHIER

The MULTIPATHHIER option specifies that a given cell that contains detail data can have more than one path into a cell that contains aggregated data. Certain calculations require this kind of multiple-path aggregation.

Syntax

MULTIPATHHIER = {YES|<u>NO</u>}

Arguments

YES
Allows a detail data cell to aggregate in multiple paths to the same ancestor cell.

NO
Disallows a detail data cell to aggregate in multiple paths to the same ancestor cell. (Default)

Notes

When to Use MULTIPATHHIER

The only time you should set the MULTIPATHHIER option to `YES` is when a calculation requires the use of multiple paths.

Interpreting an XSHIERCK01 Error Message

When you use the AGGREGATE command, dimension hierarchies are automatically checked for circularity. When MULTIPATHHIER is set to `NO`, or when the default of `NO` has not been changed, then the following error message is displayed when a detail data cell uses multiple paths to the same aggregate data cell.

```
ERROR: (XSHIERCK01) One or more loops have been detected
in your hierarchy n over N. The loops include 2 items
(UNDIRECTED: X and Y).
```

In the preceding error message, `X` is the name of the detail data cell, and `Y` is the name of the ancestor cell into which the detail data cell takes more than one path to aggregate. For more information, see Example 18–10, "Defining Multiple Paths in a Hierarchy" on page 18-32.

This error message is displayed because the multiple paths taken by the detail data cell have been interpreted as a circular hierarchy. When this is a mistake and you did not intend to create multiple paths, then change the hierarchy. Otherwise, set the MULTIPATHHIER option to `YES`.

Examples

Example 18–10 Defining Multiple Paths in a Hierarchy

This example shows how you can define multiple paths in a hierarchy, the error message that results when you attempt to aggregate data, how to interpret that message, and how to resolve the problem.

The following statements create two paths from a detail data cell to an ancestor cell that contains aggregated data.

```
DEFINE geog TEXT DIMENSION
DEFINE path INTEGER DIMENSION
DEFINE geog.geog RELATION geog <geog path>
MAINTAIN geog ADD 'A1' 'b1' 'b2' 'Top'
MAINTAIN path ADD 2
geog.geog(geog 'A1' path 1) = 'B1'
geog.geog(geog 'A1' path 2) = 'B2'
geog.geog(geog 'B1' path 1) = 'Top'
geog.geog(geog 'B2' path 1) = 'Top'
```

First, a geography dimension named `geog` and a second dimension named `path` are defined.

A relation named `geog.geog` is defined, in which the geography dimension is dimensioned by itself and the `path` dimension.

Dimension values named `A1`, `B1`, `B2`, and `Top` are added to the `geog` dimension.

Two dimension values are added to the `path` dimension. Because `path` was defined with an integer data type, the dimension values that are automatically assigned to it are the integers `1` and `2`.

Finally, the hierarchy for the `geog` dimension is created. The `A1` dimension value is the detail data. The `B1` and `B2` dimension values are the second level of the hierarchy. The `Top` dimension value is the top of the hierarchy.

A1 has two aggregation paths: A1 aggregates into B1, which aggregates into Top; A1 aggregates into B2, which aggregates into Top.

The following statements define a variable named myvar, assign a data value of 1 to its detail data level (A1), and define an aggmap for that variable.

```
DEFINE myvar INTEGER VARIABLE <geog>
myvar(geog 'A1') = 1
DEFINE myvar.aggmap <geog>
AGGMAP 'RELATION geog.geog'
```

An attempt to aggregate myvar generates the following error message.

```
AGGREGATE myvar USING myvar.aggmap
ERROR: (XSHIERCK01) One or more loops have been detected
in your hierarchy GEOG.GEOG over GEOG. The loops include 2
items (UNDIRECTED: A1 and TOP).
```

The multiple paths of aggregation that have been created for A1 have been interpreted as a circular hierarchy, because the MULTIPATHHIER option is currently set to NO.

When you had made a mistake and created these multiple paths by mistake, you would fix the problem in the hierarchy.

However, in this case, the multiple paths have been created because a calculation requires them. Therefore, the solution is to set MULTIPATHHIER to YES. Now you can execute the AGGREGATE command without error.

NAFILL

The NAFILL function returns the values of the source expression with any `NA` values replaced with the specified fill expression.

Return Value

The value returned is the same data type as *source-expression*.

Syntax

NAFILL(*source-expression fill-expression*)

Arguments

source-expression
The expression being evaluated. For values of *source-expression* that do not equal `NA`, NAFILL returns the corresponding values of *source-expression*. *Source-expression* determines the dimensions and data type of the result.

fill-expression
The expression to be substituted in the return value. *Fill-expression* must have the same data type as *source-expression*. *Fill-expression* is only evaluated for values of *source-expression* that equal `NA`.

Notes

Mismatched Data Types

When the fill and source expressions do not have the same data type, Oracle OLAP converts the fill expression to the data type of the source expression when possible. Otherwise, an error is produced.

Functions in the Fill Expression

You can use any functions in the fill expression as long as they return the same data type as the source expression.

NA Fill Expression

When both the source and fill expressions equal `NA`, then NAFILL returns `NA`.

NATRIGGER Takes Precedence Over NAFILL

Oracle OLAP evaluates an $NATRIGGER property expression before applying the NAFILL function. When the $NATRIGGER expression is `NA`, then the NAFILL function has an effect.

Examples

Example 18–11 Filling NA Values with Zeros

Suppose you have `NA` values in the variable `sales` and you want to calculate an average that counts those values as zeros. Ordinarily, AVERAGE ignores `NA` values and does not count them in the number of values being averaged. You can use NAFILL inside the AVERAGE function to temporarily treat those values as zeros so they will count in calculating the average.

```
REPORT AVERAGE(NAFILL(sales 0.0))
```

NAME

NAME is a special dimension that is used to organize the list of objects in a workspace. Its values are the names of the objects defined in the workspace.

Data type

TEXT

Syntax

NAME

Notes

LIMIT and STATUS with NAME

To see the names of all your workspace objects, use the LISTNAMES command. To see only some of the names, first limit the NAME dimension to the values in which you are interested, then use the STATUS command or the STATLIST function. See Example 18–12, "Listing Dimensions" on page 18-37.

Changing the Values of NAME

You cannot change the values of the NAME dimension with the MAINTAIN command. You must use DEFINE, DELETE, MOVE, or RENAME to change its values. Also, you cannot define an object dimensioned by NAME.

Listing Objects in an Attached Workspace

When you are have more than one workspace attached, the values in the NAME dimension include only the objects in the current workspace, which is listed first on the workspace list. You cannot list the objects in another attached workspace unless you make it the current workspace by reattaching it with the FIRST keyword. See AW ATTACH for more information.

NAME Dimension and QONs

You cannot use a qualified object name to specify the NAME dimension of a workspace that is not the current workspace, for example when you are using the LIMIT command on the NAME dimension.

TEXT, Not NTEXT

All object names must have the TEXT data type, not the NTEXT data type. Therefore, object names cannot contain characters that do not exist in the database character set.

Examples

Example 18–12 Listing Dimensions

Suppose you want a list of all the dimensions in a workspace. First, use the LIMIT command and the OBJ function to limit the status of the NAME dimension. Then use the STATUS command to produce a list of dimensions. Since the values returned by `OBJ(TYPE)` are always in uppercase, you must use `'DIMENSION'` (not `'dimension'`) in the LIMIT command to get a match. The statements

```
LIMIT NAME TO OBJ(TYPE) EQ 'DIMENSION'
STATUS NAME
```

produce the following output.

```
The current status of NAME is:
PRODUCT, DISTRICT, DIVISION, LINE, REGION, MARKETLEVEL, MARKET,
MONTH, YEAR, QUARTER
```

Example 18–13 Listing Relations

Suppose you want to see the definitions of all the relations in a workspace. Use the LIMIT command and the OBJ function to select these names. Then use DESCRIBE to produce a list of their definitions. The statements

```
LIMIT NAME TO OBJ(TYPE) EQ 'RELATION'
DESCRIBE
```

produce the following output.

```
DEFINE REGION.DISTRICT RELATION REGION <DISTRICT>
LD REGION for each DISTRICT

DEFINE DIVISION.PRODUCT RELATION DIVISION <PRODUCT>
LD DIVISION for each PRODUCT

DEFINE MLV.MARKET RELATION MARKETLEVEL <MARKET>

DEFINE MARKET.MARKET RELATION MARKET <MARKET>
LD Self-relation for the Market Dimension
```

NASKIP

The NASKIP option controls whether NA values are considered as input to aggregation functions.

Data type

BOOLEAN

Syntax

NASKIP = NO|YES

Arguments

NO
NA values are considered by aggregation functions. When any of the values being considered are NA, the function returns NA for that value.

YES
NA values are ignored by aggregation functions. Only expressions with actual values are used in calculations. (Default)

Notes

Statements Affected by NASKIP

The following OLAP DML statements are affected by NASKIP.

AGGREGATE command
AGGREGATE function
ANY
AVERAGE
COUNT
CUMSUM
DEPRDECL
DEPRDECLSW
DEPRSL
DEPRSOYD
EVERY
FINTSCHED

FPMTSCHED
IRR
LARGEST
MEDIAN
MOVINGAVERAGE
MOVINGMAX
MOVINGMIN
MOVINGTOTAL
NONE
NPV
SMALLEST
STDDEV
TCONVERT
TOTAL
VINTSCHED
VPMTSCHED

Other statements are not affected by the setting of NASKIP, they always ignore `NA` values.

Arithmetic Operators in Function Arguments

NASKIP does not affect arithmetic operators; the NASKIP2 option controls how `NA` values are treated with the `+` (plus) and `-` (minus) operators. When NASKIP2 is set to `YES`, zeroes are substituted for `NA` values in arithmetic operations with the `+` (plus) and `-` (minus) operators. This means that when a `+` (plus) and `-` (minus) operator are used in an *expression* argument to an aggregation function, the result of the calculation depends on the settings of both NASKIP and NASKIP2.

$NATRIGGER Takes Precedence over NAFILL or NA Options

An $NATRIGGER property expression is evaluated before the NAFILL function or the NASKIP, NASKIP2, or NASPELLoption is applied. When the $NATRIGGER expression is `NA`, the NAFILL function and the `NA` options have an effect.

Related Statements

NASKIP2.

Examples

Example 18–14 The Effect of NASKIP on the TOTAL Function

In the `demo` workspace, the 1997 values for sales are `NA`. The TOTAL function returns different results depending on the setting of NASKIP.

The statements

```
ALLSTAT
NASKIP = YES
SHOW TOTAL(sales)
```

produce the following result.

```
63,181,743.50
```

In contrast, the OLAP DML statements

```
NASKIP = NO
SHOW TOTAL(sales)
```

produce the following result.

```
NA
```

Example 18–15 The Effect of NASKIP on the MOVINGMIN Function

This example aggregates values for three months: the current month and the two months before it. The first report of SALES shows the `NA` values for months in 1997. When NASKIP is `YES`, the MOVINGMIN function returns `NA` only for March 1997 because all the values considered for that month were `NA`. When NASKIP is `NO`, the third statement (`REPORT DOWN month sales`) shows `NA` values for January through March 1997, because at least one value considered by MOVINGMIN for those months was `NA`.

```
LIMIT district TO 'Seattle'
LIMIT month TO 'Jul96' TO 'Mar97'
REPORT DOWN month sales
```

The preceding statements produce the following report of SALES data.

```
DISTRICT: SEATTLE
      ------------------------SALES-------------------------
      -----------------------PRODUCT------------------------
MONTH   Tents      Canoes    Racquets  Sportswear  Footwear
----- ---------- ---------- ---------- ----------  ---------
Jul96 123,700.17 157,274.03  60,198.52  78,305.97  78,019.87
Aug96 120,650.72 128,660.89  45,046.71  66,853.26  83,347.55
Sep96  97,188.43 122,702.13  42,257.14  63,777.36  99,464.05
Oct96  91,578.77  79,925.93  39,729.25  55,021.85  83,537.58
Nov96  56,044.34  77,357.10  39,024.93  44,004.12  65,216.94
Dec96  41,576.26  67,609.36  36,156.10  40,575.34  62,113.72
Jan97         NA         NA         NA         NA         NA
Feb97         NA         NA         NA         NA         NA
Mar97         NA         NA         NA         NA         NA
```

The statements

```
NASKIP = YES
REPORT DOWN month MOVINGMIN(sales -2, 0, 1, month)
```

produce the following report, which shows `NA` values for March 1997.

```
DISTRICT: SEATTLE
      -----------MOVINGMIN(SALES -2, 0, 1, MONTH)-----------
      ---------------------PRODUCT--------------------------
MONTH   Tents      Canoes    Racquets  Sportswear  Footwear
----- ---------- ---------- ---------- ----------  ---------
Jul96 108,663.59 125,823.37 57,666.37  57,713.27   73,085.88
Aug96 119,066.18 128,660.89 45,046.71  60,322.88   78,019.87
Sep96  97,188.43 122,702.13 42,257.14  63,777.36   78,019.87
Oct96  91,578.77  79,925.93 39,729.25  55,021.85   83,347.55
Nov96  56,044.34  77,357.10 39,024.93  44,004.12   65,216.94
Dec96  41,576.26  67,609.36 36,156.10  40,575.34   62,113.72
Jan97  41,576.26  67,609.36 36,156.10  40,575.34   62,113.72
Feb97  41,576.26  67,609.36 36,156.10  40,575.34   62,113.72
Mar97         NA         NA         NA         NA         NA
```

The statements

```
NASKIP = NO
REPORT DOWN month MOVINGMIN(sales -2, 0, 1, month)
```

produce the following report, which shows `NA` values for January through March 1997.

```
DISTRICT: SEATTLE
      ----------MOVINGMIN(SALES -2, 0, 1, MONTH)-------------
      -----------------------PRODUCT------------------------
MONTH   Tents      Canoes    Racquets  Sportswear  Footwear
----- ---------- ---------- ---------- ----------  ----------
Jul96 108,663.59 125,823.37  57,666.37  57,713.27   73,085.88
Aug96 119,066.18 128,660.89  45,046.71  60,322.88   78,019.87
Sep96  97,188.43 122,702.13  42,257.14  63,777.36   78,019.87
Oct96  91,578.77  79,925.93  39,729.25  55,021.85   83,347.55
Nov96  56,044.34  77,357.10  39,024.93  44,004.12   65,216.94
Dec96  41,576.26  67,609.36  36,156.10  40,575.34   62,113.72
Jan97         NA         NA         NA         NA          NA
Feb97         NA         NA         NA         NA          NA
Mar97         NA         NA         NA         NA          NA
```

NASKIP2

The NASKIP2 option controls how `NA` values are treated in arithmetic operations with the + (plus) and - (minus) operators. The result is `NA` when any operand is `NA` unless NASKIP2 is set to `YES`.

Data type

BOOLEAN

Syntax

NASKIP2 = YES|NO

Arguments

YES
Zeroes are substituted for `NA` values in arithmetic operations using the + (plus) and - (minus) operators. The two special cases of `NA + NA` and `NA - NA` both result in `NA`.

NO
`NA` values are treated as NAs in arithmetic operations using the + (plus) and - (minus) operators. When any of the operands being considered is `NA`, the arithmetic operation evaluates to `NA`. (Default)

Notes

Operators in Function Arguments

NASKIP2 is independent of NASKIP. NASKIP2 applies only to arithmetic operations with the + (plus) and - (minus) operators. NASKIP applies only to aggregation functions. However, when an *expression* argument to an aggregation function contains a+ (plus) and - (minus) operator, the results of the calculation depend on both NASKIP and NASKIP2. See "The Effects of NASKIP2 and NASKIP" on page 18-44.

How NASKIP2 Works

The following four lines show four steps in the evaluation of a complex expression that contains NAs when NASKIP2 is set to `YES`.

```
3 * (NA + NA) - 5 * (NA + 3)
   3 * NA     -    5 *  3
     NA       -      15
             -15
```

$NATRIGGER Takes Precedence over NAFILL or NA Options

An $NATRIGGER property expression is evaluated before the NAFILL function or the NASKIP, NASKIP2, or NASPELL option is applied. When the $NATRIGGER expression is `NA`, the NAFILL function and the `NA` options have an effect.

Examples

Example 18–16 The Effects of NASKIP2 and NASKIP

In the following example, INTEGER variables `X` and `Z`, dimensioned by the INTEGER dimension INTDIM, have the values shown in the second and third columns of the report. The sum of X + Z is given for each combination of NASKIP and NASKIP2 settings, starting with their defaults. The example also shows that when the + (plus) operator is used in the *expression* argument to the TOTAL function, the results that are returned by TOTAL depend on the settings of both NASKIP and NASKIP2.

Example 18–17 NASKIP Set to YES, NASKIP2 Set to NO

In the first set of examples, NASKIP is set to `YES`, which means `NA` values are ignored by the TOTAL function. NASKIP2 is set to `NO`, which means that the result of a + (plus) operation will be `NA` when any of the operands are `NA`.

```
NASKIP = YES
NASKIP2 = NO
COLWIDTH = 5
REPORT LEFT W 6 DOWN intdim x, z, x + z
```

These statements produce the following output. With NASKIP2 set to `NO`, the expression X + Z evaluates to `NA` when either X or Z is `NA`.

```
INTDIM   X      Z    x + z
------ ----- ----- -----
1         NA     2    NA
2          3    NA    NA
3          7     6    13
```

The following statement uses a + (plus) operator within the *expression* argument to the TOTAL function.

```
SHOW TOTAL(x + z)
```

This statement produces the following result.

```
13
```

The next statement uses the + (plus) operator to add the results that are returned by two TOTAL functions.

```
SHOW TOTAL(x) + TOTAL(z)
```

This statement produces the following result.

```
18
```

Example 18–18 NASKIP Set to YES, NASKIP2 Set to YES

In the next set of examples, NASKIP is again set to `YES`, which means `NA` values are ignored by the TOTAL function. NASKIP2 is now set to `YES`, which means that `NA` values are ignored by the + (plus) operator

```
NASKIP = YES
NASKIP2 = YES
REPORT LEFT W 6 DOWN intdim x, z, x + z
```

These statements produce the following output. With NASKIP2 set to `YES`, `NA` values are ignored when the expression `X + Z` is evaluated.

```
INTDIM   X      Z    X + Z
------ ----- ----- -----
1         NA     2     2
2          3    NA     3
3          7     6    13
```

The following statement uses a + (plus) operator within the *expression* argument to the TOTAL function.

```
SHOW TOTAL(x + z)
```

This statement produces the following result.

```
18
```

The next statement uses the + (plus) operator to add the results that are returned by two TOTAL functions.

```
SHOW TOTAL(x) + TOTAL(z)
```

This statement produces the following result.

```
18
```

Example 18–19   NASKIP Set to NO, NASKIP2 Set to YES

In the next set of examples, NASKIP is set to `NO`, which means that when any values considered by the TOTAL function are `NA`, TOTAL will return `NA`. NASKIP2 is again set to `YES`, which means that `NA` values are ignored by the + (plus) operator.

```
NASKIP = NO
NASKIP2 = YES
REPORT LEFT W 6 DOWN intdim x, z, x + z
```

These statements produce the following result.

```
INTDIM   X      Z   X + Z
------ ----- ----- -----
1         NA     2     2
2          3    NA     3
3          7     6    13
```

The following statement uses a + (plus) operator within the *expression* argument to the TOTAL function.

```
SHOW TOTAL(x + z)
```

This statement produces the following result.

```
18
```

The next statement uses the + (plus) operator to add the results that are returned by two TOTAL functions.

```
SHOW TOTAL(x) + TOTAL(z)
```

This statement produces the following result.

```
NA
```

Example 18–20 NASKIP Set to NO, NASKIP Set to NO

In the final set of examples, NASKIP is again set to `NO`, which means that when any values considered by the TOTAL function are `NA`, TOTAL will return `NA`. NASKIP2 is now set to `NO`, which means that the result of a + (plus) operation will be `NA` when any of the operands are `NA`.

```
NASKIP = NO
NASKIP2 = NO
REPORT LEFT W 6 DOWN intdim x, z, x + z
```

These statements produce the following result.

```
INTDIM   X      Z   X + Z
------ ----- ----- -----
1         NA     2    NA
2          3    NA    NA
3          7     6    13
```

The following statement uses a + (plus) operator within the *expression* argument to the TOTAL function.

```
SHOW TOTAL(x + z)
```

This statement produces the following result.

```
NA
```

The next statement uses the + (plus) operator to add the results that are returned by two TOTAL functions.

```
SHOW TOTAL(x) + TOTAL(z)
```

This statement produces the following result.

```
NA
```

NASPELL

The NASPELL option controls the spelling that is used for `NA` values in output.

Data type

TEXT

Syntax

NASPELL = {'*text*'|'NA'}

Arguments

text

The spelling to use for any `NA` value in output. When you specify an expression rather than a text literal, you can omit the single quotes. The default is `NA`.

Notes

Setting NASPELL to "0"

Setting NASPELL to the text character `0` (zero) causes `NA` values to appear as `0`. However, they are still treated as NAs in calculations.

Assigning NA Values

NASPELL affects only Oracle OLAP output; it does not affect the way you assign an `NA` value. For example, even when you have set NASPELL to NONE, you assign an `NA` value as follows.

```
var1 = NA
```

$NATRIGGER Takes Precedence over NASPELL

Oracle OLAP evaluates an $NATRIGGER property expression before applying the NASPELL option. When the $NATRIGGER expression is `NA`, then the NASPELL option has an effect.

Examples

Example 18–21 Showing NA Values as "NONE"

Suppose you have a variable called `current.month`, which has a value of `NA` whenever no current month has been specified. In this case, you would like the value to appear as `None` rather than `NA`.

When NASPELL is set to its default value of `NA`, the OLAP DML statement

```
SHOW current.month
```

produces the following output.

```
NA
```

In contrast, the OLAP DML statements

```
NASPELL = 'None'
SHOW current.month
```

produce the following output.

```
None
```

NEW_TIME

The NEW_TIME function converts a date and time from one time zone to another.

Return Value

DATETIME

Syntax

NEW_TIME(*datetime-exp this_zone new_zone*)

Arguments

this_zone
A text expression that indicates the time zone from which you want to convert *datetime-exp*. It must be a valid time zone, as listed in the following table.

new_zone
A text expression that indicates the time zone into which you want to convert *datetime-exp*. It is the time zone of the return value. It must be a valid time zone, as listed in Table 18–1, " Time Zones".

Table 18–1 Time Zones

AST	Atlantic Standard Time
ADT	Atlantic Daylight Time
BST	Bering Standard Time
BDT	Bering Daylight Time
CST	Central Standard Time
CDT	Central Daylight Time
EST	Eastern Standard Time
EDT	Eastern Daylight Time
GMT	Greenwich Mean Time
HST	Alaska-Hawaii Standard Time
HDT	Alaska-Hawaii Daylight Time

Table 18–1 (Cont.) Time Zones

AST	Atlantic Standard Time
MST	Mountain Standard Time
MDT	Mountain Daylight Time
NST	Newfoundland Standard Time
PST	Pacific Standard Time
PDT	Pacific Daylight Time
YST	Yukon Standard Time
YDT	Yukon Daylight Time

Examples

Example 18–22 Using the Current Time of day

The SYSDATE function returns the current date and time to the NEW_TIME function.

```
SHOW new_time(SYSDATE 'EST' 'PST')
```

When the date and time in Eastern Standard Time are October 20, 2000, at 1:20 A.M., then the date in Pacific Standard Time, which is three hours earlier, is October 19, 2000. Because SYSDATE uses the format specified by NLS_DATE_FORMAT, which by default only shows the date, the time is not displayed.

```
19-OCT-00
```

Example 18–23 Specifying the Time of day

In the following example, the TO_DATE function converts a text string to a valid date and time. The TO_CHAR function includes a date format that temporarily overrides the date format specified by the NLS_DATE_FORMAT option.

```
SHOW TO_CHAR(NEW_TIME(TO_DATE('11-27-00 22:15:00', 'MM-DD-YY HH24:MI:SS'), -
   'HST' 'PST') 'MM-DD-YY HH24:MI:SS')
```

This statement converts November 27 at 10:15 P.M. (22:15:00) Alaska-Hawaii Standard Time to November 28 at 12:15 A.M. (00:15:00) Pacific Standard Time. The date format specified in the TO_CHAR function allows the time to be displayed along with the date.

```
11-28-00 00:15:00
```

Alternatively, you can change the value of NLS_DATE_FORMAT.

```
NLS_DATE_FORMAT = 'MM-DD-YY HH24:MI:SS'
```

Then this statement produces the same result, without requiring the use of TO_CHAR.

```
SHOW NEW_TIME(TO_DATE('11-27-00 22:15:00', 'MM-DD-YY HH24:MI:SS'), -
   'HST' 'PST')
```

NEXT_DAY

The NEXT_DAY function returns the date of the first instance of a particular day of the week that follows the specified date.

Return Value

DATETIME

Syntax

NEXT_DAY(*datetime-expression*, *weekday*)

Arguments

datetime-expression
An expression that has the DATETIME data type.

weekday
A text expression that identifies a day of the week (for example, Monday). Valid names are controlled by the NLS_DATE_LANGUAGE option.

Examples

Example 18–24 Getting a Future Date

The following statement returns the date of the first Tuesday following today's date.

```
SHOW NEXT_DAY(SYSDATE, 'Tues')
```

When today is Friday, September 8, 2000, then the following Tuesday is:

```
11-SEP-00
```

NLS Options

Oracle bases its globalization support on the values of parameters that begin with NLS. The Oracle OLAP DML includes a number options that correspond to these parameters.

Within a session, you can dynamically modify the value of some of these NLS parameters by setting them using the OLAP DML options described in Table 18–2, " OLAP DML NLS Options" or by using the SQL statement `ALTER SESSION SET` *`option`* = *`value`*.

Table 18–2 OLAP DML NLS Options

Option Name	Description
NLS_CALENDAR	The calendar for the session.
NLS_CURRENCY	The local currency symbol for the L number format element for the session.
NLS_DATE_FORMAT	The default format for DATETIME values.
NLS_DATE_LANGUAGE	The language for days, months, and similar language-dependent DATE format elements.
NLS_DUAL_CURRENCY	A second currency symbol in addition to the local currency symbol (which is identified by NLS_CURRENCY). It is used primarily to identify the Euro symbol.
NLS_ISO_CURRENCY	The international currency symbol for the C number format element.
NLS_LANG	The current language, territory, and database character set, which are determined by session-wide globalization parameters.
NLS_LANGUAGE	The current language for the session.
NLS_NUMERIC_CHARACTERS	The decimal marker and thousands group marker for the session.
NLS_SORT	the sequence of character values used when sorting or comparing text.
NLS_TERRITORY	The current territory for the session.

Data Type

TEXT

Syntax

option-name = option-value

Arguments

See *Oracle Database SQL Reference* for more information about NLS parameters, including valid values.

Notes

Date Formats

The calendar system affects the input format for the TO_DATE function and the output produced by the TO_CHAR function.

Default Value of NLS_CURRENCY

The default value of NLS_CURRENCY controlled by the value of the NLS_TERRITORY option. Resetting NLS_TERRITORY restores NLS_CURRENCY to its default setting

Output Formats

The TO_CHAR and TO_NUMBER functions use the value of NLS_CURRENCY as their default currency symbol. Both functions can accept an optional NLS_CURRENCY argument that overrides the NLS_CURRENCY option.

Date Formats

NLS_DATE_LANGUAGE specifies the language to use for the spelling of day and month names and date abbreviations (a.m., p.m., AD, BC) returned by the TO_DATE function and the output produced by the TO_CHAR function.

Default Value of NLS_DUAL_CURENCH

The default value is the dual currency symbol defined in the territory of your current language environment which is controlled by the value of the NLS_TERRITORY option. Resetting NLS_TERRITORY restores NLS_DUAL_CURRENCY to its default setting.

Supported Symbols

When you want to identify the Euro symbol as the value of NLS_DUAL_CURRENCY, the instance character set must support that symbol.

Number Formats

The value of NLS_DUAL_CURRENCY takes the place of the letter `U` in a number format model. See the TO_NUMBER function for a description of number format elements.

Default Value of NLS_ISO_CURRENCY

The NLS_TERRITORY option sets the default value for NLS_ISO_CURRENCY. Resetting NLS_TERRITORY restores NLS_ISO_CURRENCY to its default value

ISO Currency Symbols

Local currency symbols can be ambiguous. For example, a dollar sign (`$`) can indicate Canadian dollars, U.S. dollars, or Australian dollars. `ISO 4217` defines a unique three-character code for each currency, composed of a two-character country code and a one-character currency designator. For example, the code for Canadian dollars is `CAD` (`CA` for Canada, and `D` for dollars). Similarly, the code for U.S. dollars is `USD`, and for Australian dollars in `AUD`.

Output Format

The TO_CHAR and TO_NUMBER functions use the value of NLS_ISO_CURRENCY for the `C` number format element. These functions can accept an optional NLS_ISO_CURRENCY argument that overrides the NLS_ISO_CURRENCY option.

Default Values

The NLS_LANGUAGE setting controls the default values of the NLS_DATE_LANGUAGE and NLS_SORT parameters, as well as the default values for messages, day and month names, symbols for AD, BC, a.m., and p.m., and the default sorting mechanism. Some of these default values can be overridden individually by other OLAP DML options. See "Locale Settings" on page 18-57.

Interaction with NLS_LANG

The value of NLS_LANGUAGE sets the language component of the read-only NLS_LANG option.

Locale Settings

NLS_LANGUAGE sets the default values for the NLS_DATE_LANGUAGE and NLS_SORT options. It also affects the results of the TO_CHAR function.

YESSPELL and NOSPELL

The values of the read-only YESSPELL and NOSPELL options reflect the NLS_LANGUAGE setting.

Date Formats

The value of NLS_LANGUAGE controls whether the AD and BC era designators use periods after the letters. You can change this element of the date format by using the TO_CHAR function.

Default Value of NLS_NUMERIC_CHARACTERS

The appropriate decimal and thousands markers vary from one geographic area to the next. Thus, the default value of NLS_NUMERIC_CHARACTERS is determined by the value of the NLS_TERRITORY option. Resetting NLS_TERRITORY restores NLS_NUMERIC_CHARACTERS to its default value

Format of Numeric Input

NLS_NUMERIC_CHARACTERS affects only the display of numeric data. When supplying values for input to an OLAP DML statement, do not use a thousands group marker, and use a period (.) for the decimal marker. To use a different markers for input data, enclose the value in single quotes and use the TO_NUMBER function to convert the value from text to a valid number.

Effect on Other Options

A change to NLS_NUMERIC_CHARACTERS causes an equivalent change in either the THOUSANDSCHAR option, the DECIMALCHAR option, or both.

Output Format Overrides

The TO_CHAR, and TO_NUMBER functions have an optional NLS_NUMERIC_CHARACTERS argument that overrides the NLS_NUMERIC_CHARACTERS option.

Performance

Linguistically correct sorting is slower than binary sorting.

Asian Languages

East Asian languages are typically sorted by their binary values, since linguistically appropriate sorting is often too difficult and complex to be feasible.

Comparison Operators

The value of NLS_SORT affects the `GT`, `GE`, `LT`, and `LE` operators.

Sorting Statements

The value of NLS_SORT affects the SORT command and the SORTLINES function.

Default Values

The setting of NLS_TERRITORY controls the default values of the thousands separator, the decimal marker, the currency symbol, and the sort order. These default values can be overridden individually by other OLAP DML options. See "Locale Settings" on page 18-57.

Interaction with NLS_LANG

The value of NLS_TERRITORY sets the territory component of the read-only NLS_LANG option.

Locale Settings

NLS_TERRITORY sets the default values for the NLS_DATE_FORMAT, NLS_CURRENCY, NLS_ISO_CURRENCY, NLS_NUMERIC_CHARACTERS, and NLS_DUAL_CURRENCY options. Setting NLS_TERRITORY will reset these options to their default values.

Numeric Formatting

NLS_TERRITORY affects numeric formatting in the TO_CHAR function.

Date Formatting

NLS_TERRITORY affects date formatting in the TO_DATE function.

Examples

Example 18–25 Changing Calendar Systems

The following statement sets NLS_CALENDAR to the Thai Buddha calendar.

```
NLS_CALENDAR = 'THAI BUDDHA'
```

Example 18–26 Setting the Language for Dates

The following statements set the language for dates to Spanish and change the default date format.

```
NLS_DATE_LANGUAGE = 'SPANISH'
NLS_DATE_FORMAT = 'Month DD, YYYY'
```

A `SHOW SYSDATE` statement now generates the date in Spanish.

```
Septiembre 08, 2000
```

Example 18–27 Setting the Second Currency Symbol

The following statement sets NLS_DUAL_CURRENCY to the symbol for pounds sterling.

```
NLS_DUAL_CURRENCY = '
£
'
```

Example 18–28 Setting the International Currency Symbol

The following statement changes the ISO symbol to French francs (`FRF`).

```
NLS_ISO_CURRENCY = 'FRANCE'
```

Example 18–29 Checking the Current Value

A `SHOW NLS_LANG` statement might produce the following.

```
AMERICAN_AMERICA.WE8ISO8859P1
```

Example 18–30 Effects of Changing NLS_LANGUAGE

In this example, the NLS_LANG option is initially set to:

```
AMERICAN_AMERICA.WE8ISO8859P1
```

The value of YESSPELL is `yes`.

A change to the language setting:

```
NLS_LANGUAGE = 'FRENCH'
```

changes the value of NLS_LANG to:

```
FRENCH_AMERICAN.WE8ISO8859P1
```

The value of YESSPELL is now `oui`.

Example 18–31 Changing the Decimal Marker to a Comma

The following statement changes the decimal marker to a comma, and the thousands marker to a space.

```
NLS_NUMERIC_CHARACTERS = ', '
```

The result of the following statement

```
show 1234.56
```

is now

```
1 234,56
```

Example 18–32 Binary and Linguistic Sorts

A dimension named `words` has the following values.

```
cerveza, Colorado, cheremoya, llama, luna, lago
```

This example shows the results of a binary sort.

```
NLS_SORT = 'BINARY'
SORT words A words
STATUS words
The current status of WORDS is:
Colorado, cerveza, cheremoya, lago, llama, luna
```

A Spanish language sort results in this order.

```
NLS_SORT = 'SPANISH'
SORT words A words
STATUS words
The current status of WORDS is:
cerveza, cheremoya, Colorado, lago, llama, luna
```

An extended Spanish language sort results in this order.

```
NLS_SORT = 'XSPANISH'
SORT words A words
STATUS words
The current status of WORDS is:
cerveza TO cheremoya, lago TO llama
```

Example 18–33 Effects of Changing NLS_TERRITORY

In this example, the NLS_LANG option is initially set to:

```
AMERICAN_AMERICA.WE8ISO8859P1
```

The thousands marker is a comma (`,`), and the decimal marker is a period (`.`).

```
SHOW TO_NUMBER('12345')
12,345.00
```

A change to the territory setting:

```
NLS_TERRITORY = 'FRANCE'
```

changes the value of NLS_LANG to:

```
AMERICAN_FRANCE.WE8ISO8859P1
```

The thousands marker is now a period (`.`), and the decimal marker is a comma (`,`).

```
SHOW TO_NUMBER('12345')
12.345,00
```

NONE

The NONE function returns `YES` when no values of a Boolean expression are `TRUE`. It returns `NO` when any value of the expression is `TRUE`.

Return Value

BOOLEAN

Syntax

NONE(*boolean-expression* [[STATUS] *dimensions*])

Arguments

boolean-expression
The Boolean expression to be evaluated.

STATUS
Can be specified when one or more of the dimensions of the result of the function are not dimensions of the Boolean expression. (See the description of the *dimensions* argument.) When you specify the STATUS keyword when this is not the case, Oracle OLAP produces an error.

In cases where one or more of the dimensions of the result of the function are not dimensions of the Boolean expression, the STATUS keyword might be *required* in order for Oracle OLAP to process the function successfully, or the STATUS keyword might provide a performance enhancement. See "The STATUS Keyword" on page 18-64.

dimensions
The dimensions of the result. By default, NONE returns a single value. When you indicate one or more dimensions for the result, NONE tests for `TRUE` values along the dimensions that are specified and returns an array of values. Each dimension must be either a dimension of *boolean-expression* or related to one of its dimensions. When it is a related dimension, you can specify the name of the relation instead of the dimension name. This enables you to choose which relation is used when there is more than one.

Notes

NA Values

When the Boolean expression involves an NA value, NONE returns a YES or NO result when it can, as shown in the following table.

Boolean expression	**Result**
NA EQ NA	YES
NA NE NA	NO
NA EQ non-NA	NO
NA NE non-NA	YES
NA AND NO	NO
NA OR YES	YES

However, in cases where a YES or NO result would be misleading, NONE returns NA. For example, when you test whether an NA value is greater than a non-NA value, NONE returns NA.

The Effect of NASKIP

NONE is affected by the NASKIP option. When NASKIP is set to YES (the default), NONE ignores NA values and returns YES when no value of the Boolean expression is TRUE and returns NO when any values are TRUE. When NASKIP is set to NO, NONE returns NA when any value of the expression is NA. When all the values of the expression are NA, NONE returns NA for either setting of NASKIP.

Data with a Time Dimension

When *boolean-expression* is dimensioned by a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR, you can specify any other DAY, WEEK, MONTH, QUARTER, or YEAR dimension as a related *dimension.* Oracle OLAP uses the implicit relation between the dimensions. To control the mapping of one DAY, WEEK, MONTH, QUARTER, or YEAR dimension to another (for example, from weeks to months), you can define an explicit relation between the two dimensions and specify the name of the relation as the *dimension* argument to the NONE function.

For each time period in the related dimension, Oracle OLAP tests the data values for all the source time periods that end in the target time period. This method is used regardless of which dimension has the more aggregate time periods.

The STATUS Keyword

When one or more of the dimensions of the result of the function are not dimensions of the Boolean expression, Oracle OLAP creates a temporary variable to use while processing the function. When you specify the STATUS keyword, Oracle OLAP uses the current status instead of the default status of the related dimensions for calculating the size of this temporary variable.

When the size of the temporary variable for the results of the function would exceed 2 gigabytes, you *must* specify the STATUS keyword in order for Oracle OLAP to execute the function successfully. When the dimensions of the Boolean expression are limited to a few values and are physically fragmented, you can specify the STATUS keyword to improve the performance of the function.

When you use NONE with the STATUS keyword in an expression that requires going outside of the status for results (for example, with the LEAD or LAG functions or with a qualified data reference), the results outside of the status will be returned as `NA`.

Related Statements

ANY, COUNT, and EVERY.

Examples

Example 18–34 Testing for No True Values by District

Suppose you want to find out which districts had no months in which sales fell below $50,000. Use the NONE function to determine whether the Boolean expression (`SALES LT 50000`) is `TRUE` for no months. To have the results dimensioned by district, specify `district` as the second argument to NONE.

```
LIMIT product TO 'Sportswear'
REPORT NONE(sales LT 50000, district)
```

The preceding statements produce the following output.

```
               NONE(SALES
               LT 50000,
DISTRICT       DISTRICT)
-------------- ----------
Boston                 NO
Atlanta               YES
Chicago               YES
Dallas                YES
Denver                YES
Seattle                NO
```

Example 18–35 Testing for No True Values by Region

You might also want to find out which regions had no months in which no districts had sportswear sales of less than $50,000. Since the `region` dimension is related to the `district` dimension, you can specify `region` instead of `district` as a dimension for the results of ANY.

```
REPORT NONE(sales LT 50000, region)
```

The preceding statement produces the following output.

```
               NONE(SALES
               LT 50000,
REGION          REGION)
-------------- ----------
East                   NO
Central               YES
West                   NO
```

NORMAL

The NORMAL function returns a random value from a normal distribution with a specified mean and standard deviation. The result returned by NORMAL is dimensioned by all the dimensions of the mean and standard deviation expressions.

Return Value

DECIMAL

Syntax

NORMAL(*mean standard-deviation*)

Arguments

mean
A numeric expression that represents the mean of a normal distribution.

standard-deviation
A numeric expression that represents the standard deviation of a normal distribution.

Notes

NA Values

When *mean* is `NA`, NORMAL returns `NA`. When *standard-deviation* is `NA`, NORMAL returns the mean.

Examples

Example 18–36 Showing Random Values

Each of the following examples shows a random number that might be returned from a normal distribution with a mean of 0 and a standard deviation of 1.

The statement

```
SHOW NORMAL(0 1)
```

might produce the following result.

```
-0.75
```

When you execute the same statement again

```
SHOW NORMAL(0 1)
```

it might produce the following result.

```
0.87
```

The following examples show a random number that might be returned from a normal distribution with a mean of 250 and a standard deviation of 50.

The statement

```
SHOW NORMAL(250 50)
```

might produce the following result.

```
262.24
```

When you execute the same statement again

```
SHOW NORMAL(250 50)
```

it might produce the following result.

```
217.02
```

NOSPELL

(Read-only) The NOSPELL option holds the text that is used for `FALSE` Boolean values in the output of OLAP DML statements.

The value of the NOSPELL option is the word for "no" in the current language, as specified by the NLS_LANGUAGE option. For example, when NLS_LANGUAGE is set to "American," then the default value of NOSPELL is `NO`.

Data type

TEXT

Syntax

NOSPELL

Examples

Example 18–37 Seeing the Effect of the NOSPELL Option

Suppose you have a variable called BOOLVAR that currently has a value of `NO`. When "non" is the word for "no" in the language specified by the NLS_LANGUAGE option,

```
SHOW boolvar
```

produces the following output.

```
non
```

NPV

The NPV function computes the net present value of a series of cash flow values.

Return Value

DECIMAL

Syntax

NPV(*cashflows*, *discount-rate*, [*time-dimension]*)

Arguments

cashflows
A numeric expression that is dimensioned by *time-dimension* and specifies the series of cash flow values.

discount-rate
A numeric expression that specifies the interest rate for each period to be used to discount the cash flow values. It can either be a single value or an array of values with one or more non-time dimensions. You should express the discount rate as a decimal quantity; for example, 8.25 percent as `.0825`.

time-dimension
A name that specifies the time dimension. When *cashflows* has a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR, NPV will automatically use that dimension, and you can omit the *time-dimension* argument.

Notes

The Dimensions of the Result

The result returned by the NPV function is dimensioned by all the dimensions of *cashflows* except its time dimension. When *cashflows* is dimensioned only by the time dimension, NPV returns a single value.

Cash Flow Occurrences

All cash flows are assumed to occur at the beginning of the time period with which they are associated.

Cash Flows Discounted

The cash flows are discounted back to the beginning of the earliest time period that appears in the current status of the time dimension.

Ignored Cash Flows

NPV ignores cash flows that corresponds to out-of-status dimension positions.

NASKIP Option Settings

NPV is affected by the NASKIP option. When NASKIP is set to `YES` (the default), NPV ignores `NA` cash flows and computes net present value using the cash flows that are available. When NASKIP is set to `NO`, NPV returns `NA` when any cash flow has a value of `NA`. When all the cash flows are `NA`, NPV returns `NA` for either setting of NASKIP.

NA Discount Rate

When the discount rate used to compute a result value is equal to `NA`, then that result value is `NA`.

Negative Discount Rates

NPV accepts any positive discount rate, and it also accepts a negative discount rate when the rate is greater than minus one (that is, rate `> -1`). When you supply a negative rate, you must precede it with a comma.

Cash Flow Timing

Different assumptions about the intra-period timing of the cash flows, or the base time point for the present value calculations, can be accommodated by multiplying the result of the NPV function by the following quantity: one plus the discount rate, raised to an appropriate positive or negative power.

Examples

Example 18–38 Computing the Net Present Value

The following statements create a dimension called `project`, add values to it, and create a variable called `cflow`, which is dimensioned by `year` and `project`.

```
DEFINE project DIMENSION TEXT
MAINTAIN project ADD 'a' 'b' 'c' 'd' 'e'
DEFINE cflow VARIABLE DECIMAL <project year>
```

When you assign the following values to CFLOW,

```
             ------------------------CFLOW----------------------
             -----------------------PROJECT---------------------
YEAR             a          b          c          d          e
------------ ---------- ---------- ---------- ---------- -------
Yr95           -200.00    -200.00    -300.00    -100.00  -200.00
Yr96            100.00     150.00     200.00      25.00    25.00
Yr97            100.00     400.00     200.00     100.00   200.00
```

then the following statement

```
REPORT NPV(cflow, .08, year)
```

uses a discount rate of 8 percent to create the following report of the net present value of the `cflow` data.

```
               NPV(CFLOW,
PROJECT        .08, YEAR)
-------------- ----------
a                  -21.67
b                  281.82
c                   56.65
d                    8.88
e                   -5.38
```

NULLIF

The NULLIF function compares one expression with another and returns `NA` when the expressions are equal, or the base expression when they are not.

Return Value

`NA` when the expressions are equal, or the base expression when they are not.

Syntax

NULLIF (*expr1* , *expr2*)

Arguments

expr1
An expression. The base expression for the comparison

expr2
An expression to compare to *expr1*.

NUMBYTES

The NUMBYTES function counts the number of bytes in a text expression. When the value is a multiline text value, NUMBYTES returns the total number of bytes in all the lines. The result returned by NUMBYTES has the same dimensions as the specified expression.

Return Value

INTEGER

Syntax

NUMBYTES(*text-expression*)

Arguments

text-expression
The text expression whose bytes are to be counted.

Notes

Single-Byte Characters

When you are using a single-byte character, you can use the NUMCHARS function instead of the NUMBYTES function.

NTEXT Data Type

This function does not accept NTEXT arguments, because it is oriented toward byte-manipulation instead of character manipulation. It always returns values of type TEXT. When you must use this function on NTEXT values, use the CONVERT or TO_CHAR function to convert the NTEXT value to TEXT.

Examples

Example 18–39 Counting the Bytes in the Longest Name

You would like to know the length of the names of your products so you can specify the appropriate width for the label column in a report. You can use the NUMBYTES function in combination with the LARGEST function to find the length of the longest label. Then use that value to set the column size. The following statements in a program find the longest name and use the byte count to format a report.

```
firstcol = LARGEST(NUMBYTES(name.product))+1
LIMIT month TO FIRST 3
FOR product
   DO
     ROW WIDTH FIRSTCOL name.product WIDTH 6 ACROSS month -
     FIRST 3: units
   DOEND
```

When the program is run, it produces the following output.

```
3-Person Tents      200    203    269
Aluminum Canoes     347    400    482
Tennis Racquets     992  1,076  1,114
Warm-up Suits     1,096  1,214  1,294
Running Shoes     2,532  2,405  2,775
```

NUMCHARS

The NUMCHARS function counts the number of characters in a text expression. When the value is a multiline text value, NUMCHARS returns the total number of characters in all the lines. The result returned by NUMCHARS has the same dimensions as the specified expression.

Return Value

INTEGER

Syntax

NUMCHARS(*text-expression*)

Arguments

text-expression
The text expression whose characters are to be counted.

Notes

multibyte Characters

When you are using a multibyte character set, you can use the NULLIF function instead of the NUMCHARS function.

TEXT and NTEXT

NUMCHARS accepts either a TEXT or NTEXT argument. It does not perform an automatic conversion to either data type. It returns the information that is correct for the data type of the specified argument.

Examples

Example 18–40   Counting the Characters in the Longest Name

You would like to know the length of the names of your products so you can specify the appropriate width for the label column in a report. You can use the NUMCHARS function in combination with the LARGEST function to find the length of the longest label. Then use that value to set the column size. The following statements in a program find the longest name and use the character count to format a report.

```
firstcol = LARGEST(NUMCHARS(name.product))+1
LIMIT month TO FIRST 3
FOR product
   DO
     ROW WIDTH FIRSTCOL name.product WIDTH 6 ACROSS month -
     FIRST 3: units
   DOEND
```

When the program is run, it produces the following output.

```
3-Person Tents      200    203    269
Aluminum Canoes     347    400    482
Tennis Racquets     992  1,076  1,114
Warm-up Suits     1,096  1,214  1,294
Running Shoes     2,532  2,405  2,775
```

NUMLINES

The NUMLINES function counts the number of lines in each value of a text expression. The result returned by NUMLINES has the same dimensions as the specified expression.

NUMLINES accepts either a TEXT or NTEXT argument. It does not perform an automatic conversion to either data type.

Return Value

INTEGER

Syntax

NUMLINES(*text-expression*)

Arguments

text-expression
The text expression whose lines are to be counted.

Examples

Example 18–41 Counting the Number of Lines

In this example, you want to determine the number of lines in the multiline text variable LASTNAMES. The LASTNAMES variable has the following values.

```
Adamson
Jones
Smith
Taylor
```

The statement

```
SHOW NUMLINES(lastnames)
```

produces the following output.

```
4
```

NVL

The NVL function replaces a `NA` value with a string.

To evaluate a specified expression and replace a non-`NA` value with one value and a `NA` value with another, use NVL2.

Return Value

The specified replacement value when the value of the base expression is `NA`, or the base expression when the value of the base expression is not `NA`. The data type of the return value is always the same as the data type of the base expression.

Syntax

NVL (*exp* , *replacement-exp*)

Arguments

expr
The expression that you want to replace when it has a `NA` value.

replacement-exp
The value with which you want to replace a `NA` value.

NVL2

The NVL2 function returns one value when the value of a specified expression is not `NA`, or another value when the value of the specified expression is `NA`.

To merely replace a `NA` value with a string, use NVL.

Return Value

The data type of the return value is always the data type of *expr2* (that is, the expression whose value is returned when the value of *expr1* is not `NA`).

Syntax

NVL2 (*expr1* , *expr2* , *expr3*)

Arguments

expr1
The expression whose value this function evaluates.

expr2
An expression whose value is returned when the value of *expr1* is not `NA`.

expr3
An expression whose value is returned when the value of *expr1* is `NA`.

Notes

Comparing Values of Different Data Types

When the data types of *expr2* and *expr3* are different, then the function converts *expr3* to the data type of *expr2* before comparing them.

19

OBJ to QUAL

This chapter contains the following OLAP DML statements:

- PERMIT_WRITE
- PERMITERROR
- PERMITRESET
- POP
- POPLEVEL
- POUTFILEUNIT
- PRGTRACE
- PROGRAM
- PROPERTY
- PUSH
- PUSHLEVEL
- QUAL

OBJ

The OBJ function returns information about a workspace object.

Return Value

The return value depends on the *choice* keyword.

Syntax

OBJ(*choice* [*object-name*])

where *choice* is one of the following keywords that indicates the type of information you want:

AGGMAP
AGGMAPLIST
ALIASLIST
ALIASOF
AW
AWLIST
BTREE
BTREE SHARED
CACHECOUNT *var-name*
CACHEEMPTY *var-name*
CLASS
DATA
DFNDIMS
DIMMAX
DIMS
DIMTYPE
DISKSIZE
FORMULA
HASCACHE *var-name*
HASH
HASPROPERTY *prop-name*
HIDDEN *program-name*
INORDER
ISBY [RECURSIVE] *dimension-name* [*object-name*]
ISCOMPILED
KVSIZE

LD
MODEL
NAPAGES
NOHASH
NUMDELS
NUMDFNDIMS
NUMDIMS
NUMSEGS
NUMVALS
OWNSPACE
PARTBY
PARTDIMS *partitions*
PARTITION *partitions*
PARTMETH
PARTNAMES
PARTRANGE *partitions*
PERIOD
PHYSVALS
PMTMAINTAIN
PMTPERMIT
PMTREAD
PMTWRITE
PRECISION
PROGRAM
PROPERTY *prop-name*
PROPERTYLIST
PROPERTYTYPE *prop-name*
REFERS *text-expression*
SCALE
SEGWIDTH {*dimension-name* | ALL}
SPARSE
SURROGATELIST {*surrogate* | *dimension*}
TRIGGER [*triggering-event]*
TYPE
VALSIZE
VNF
WIDTH

Arguments

AGGMAP
Returns a `TEXT` value which is the specification of the aggmap that you specify.

AGGMAPLIST
Returns a `TEXT` value which is the aggmap objects in the formula that you specify.

ALIASLIST
Returns a `TEXT` value which is the alias dimensions for the dimension that you specify.

ALIASOF
Returns a `TEXT` value which is the base dimension for the alias dimension that you specify.

AW
Returns a `TEXT` value which is the name of an attached workspace that contains an object with the specified name. When the specified object is in only one attached workspace, AW returns the name of the workspace. When the specified object is in more than one attached workspace, AW still returns only one workspace name. You must use the AWLIST keyword to get all the relevant workspace names. When the object is not in any attached workspace, AW returns `NA`.

AWLIST
Returns a multiline `TEXT` value each line of which is the name of an attached workspace that contains an object with the specified name. When you specify a qualified object name for the object, AWLIST returns only the relevant workspace name. When no workspace contains the specified object, AW returns `NA`.

BTREE
Returns a BOOLEAN value that indicates whether a conjoint dimension or a composite is using the BTREE index algorithm to load and access data. For other types of objects, it returns `NA`.

CACHEEMPTY *var-name*
Returns a BOOLEAN value that indicates whether a session cache has been emptied of data for *var-name* which is a text expression that specifies the name of the variable. A cache can be emptied by using the CLEAR command with the CACHE keyword. When *var-name* is not a variable or when it has no session cache, then CACHEEMPTY returns `NA`. (For more information on the session cache, see "What is an Oracle OLAP Session Cache?" on page 21-54.)

CACHECOUNT *var-name*

Returns a LONG INTEGER value which is the number of non-`NA` cells in the session cache for *var-name* which is a text expression that specifies the name of the variable. When *var-name* is not a variable or when it does not have a no session cache, then CACHECOUNT returns `NA`. (For more information on the session cache, see "What is an Oracle OLAP Session Cache?" on page 21-54.)

CLASS

Returns a `TEXT` value which is the storage class of an object. Possible return values are:

- **TEMPORARY** — An object whose values are not saved in the workspace; applicable to valuesets, variables, relations, and worksheets.
- **null** — A permanent object whose values, when modified, are stored in a new place in the workspace until you update and are then included in the update; applicable to all object types.

DATA

Returns a `TEXT` value which is the data type of an object.

- For dimensions, variables, and formulas, possible values are `INTEGER`, `SHORTINTEGER`, `LONGINTEGER`, `DECIMAL`, `NUMBER`, `SHORT` (for SHORTDECIMAL), `BOOLEAN`, `ID`, `TEXT`, `NTEXT`, `DATE` or `DATETIME`.

  Tip: To find out the type of time period represented by dimensions of type DAY, WEEK, MONTH, QUARTER, and YEAR, use the PERIOD choice.

- For a relation, DATA returns the name of the related dimension.
- For a concat or conjoint dimension or a composite, it returns the names of the base dimensions of an object as a multiline text value.
- For a program defined with a data type, it returns the name of the data type.
- For a valueset, it returns the name of the dimension for which the valueset was defined.
- For other types of objects, it returns `NA`.

DFNDIMS

Returns a `TEXT` value which is a multiline text value that contains the names of the dimensions and composites in the dimension list that is used to define an object. An object defined with a dimension list could be a variable, relation, formula, valueset, concat or conjoint dimension, dimension surrogate, or composite.

- The name returned for an unnamed composite is the form used in the object definition: `SPARSE<dim1 dim2 ...>`.
- For a dimension surrogate, DFNDIMS returns the name of the dimension for which the surrogate was defined.
- When no dimension list was used when an object was defined, DFNDIMS returns `NA`.

DIMMAX

Returns an INTEGER value which is the number of values in a dimension. For other object types, DIMMAX returns `0` (zero). When you use the DIMMAX choice with a dimension that has a read permission that restricts access to the dimension values, the result that OBJ returns depends on whether the dimension has previously been loaded. Permissions are evaluated when an object is loaded. Generally, the first time you refer to an object in your session, Oracle OLAP loads the object and evaluates its permissions. However, the OBJ function does not load objects, since it is just providing information about them. When you use DIMMAX with a dimension that has not yet been loaded, the result reflects the entire number of values in the dimension, regardless of whether the dimension has read permissions. When a dimension with permissions has already been loaded, then the DIMMAX choice reflects the permitted size. To ensure that the DIMMAX choice returns the permitted size, you can execute a LOAD command before using the OBJ function.

DIMS

Returns a multiline `TEXT` value that contains the names of the dimensions of an object:

- For dimensions, simple, concat, or conjoint, DIMS returns the name of the dimension itself. To find out the base dimensions of a concat or conjoint dimension, use the DATA choice.
- For composites, it returns a multiline text value listing the base dimensions of the composite.
- For a dimension surrogate, it returns the name of the dimension for which the surrogate was defined.
- For other objects, it returns a multiline text value that contains the names of the dimensions of the object.
- When an object has no dimensions, it returns `NA`.

DIMTYPE
Returns one of the following `TEXT` values:

- For a concat dimension, returns `CONCAT`.
- For a conjoint dimension, returns, `CONJOINT`.
- For a composite, returns, `COMPOSITE`.
- For a simple dimension, returns the data type of the dimension.
- For a partition template object, returns `PARTITION TEMPLATE`.
- For all other objects, returns `NA`.

FORMULA
Returns a `TEXT` value which is the expression in the definition of a formula. When the object is not a formula, FORMULA returns `NA`.

HASCACHE *variable-name*
Returns a BOOLEAN value that indicates whether a session cache that is local to the session has been established to store data for *variable-name* which is a text expression that specifies the name of the variable. When *variable-name* is not a variable, HASCACHE returns `NA`. (For more information on the session cache, see "What is an Oracle OLAP Session Cache?" on page 21-54.)

HASH
Returns a BOOLEAN value that indicates whether a conjoint dimension or a composite is using the HASH index algorithm to load and access data. For other types of objects, it returns `NA`.

HASPROPERTY *prop-name*
Returns a BOOLEAN value that indicates whether the property specified by *prop-name* exists for an object. (Abbreviated HASPRP)

HIDDEN *program-name*
Returns a BOOLEAN value that indicates whether the text of the program specified by program-name has been hidden. (See the entries for the HIDE and UNHIDE commands.) For other types of objects, it returns `NA`.

ISBY [RECURSIVE] *dimension-name* [*object-name*]
When you supply both arguments returns a BOOLEAN value that answers the question: Is the specified object (*object-name*) dimensioned by or related to or a surrogate for the specified dimension (*dimension-name*)? Returns a BOOLEAN value that indicates whether an object is dimensioned by the dimension you specify in

dimension-name; or when the object is an aggmap, whether the specified dimension is a dimension of any relations or models in the aggmap.

- **RECURSIVE** specifies that Oracle OLAP should search for *dimension-name* in the base dimensions of the specified object, at any level. the. See Example 19–1, "OBJ With ISBY" on page 19-17.
- ***dimension-name*** is a text expression that is the name of a dimension. (Oracle OLAP automatically converts the name to uppercase.) When *dimension-name* is a composite, the value returned by ISBY indicates whether or not an object was defined with the composite.
- ***object-name*** is a text expression that is a dimension surrogate, variable, relation, or valueset name to learn if that object is dimensioned by or related to or a surrogate for the specified dimension. You can omit *object-name* when you are looping through the list of workspace objects to obtain information about more than one object, or when you are using OBJ to limit the NAME dimension.

ISCOMPILED
Returns a BOOLEAN value that indicates information about the compilation status of a compilable object (such as a program, model, or formula). The value returned depends on the type of object and on whether a compilation error was found in that object. For example:

- For programs, returns `YES` when the program has been processed by the compiler since the last time it was modified. A return value of `YES` does not necessarily indicate that all lines of the program are compiled. See COMPILE for more information.
- For formulas, returns `YES` only when the formula was compiled without finding a single error and when the formula can be saved. When the formula contains ampersand substitution, it cannot be saved. When the formula is empty, the ISCOMPILED choice returns `NO`.
- For models, returns `YES` only when the model was compiled without a single error found or when the model is empty.
- For programs, formulas, and models, returns `NO` when you delete an object that the program, formula, or model references.

For non-compilable objects, ISCOMPILED returns `NA`.

KVSIZE
Returns an INTEGER value which is the number of pages currently allocated to hash and BTREE indexes.

LD
Returns a `TEXT` value which is LD (long description) of an object. When the object does not have an LD, it returns `NA`.

MODEL
Returns a `TEXT` value which is the specification of a model. For other types of objects, it returns `NA`.

NOHASH
Returns a BOOLEAN value that indicates whether a conjoint dimension uses the NOHASH index algorithm to load and access data. For other types of objects, it returns `NA`.

NUMDFNDIMS
Returns an INTEGER value which is the number of dimensions or composites in the dimension list used to define an object. For this count, each composite counts as one, and the dimensions within the dimension list of the composite are not counted. An object defined with a dimension list could be a variable, relation, formula, valueset, concat or conjoint dimension, dimension surrogate, or composite. When no dimension list was used when defining the object (as for single-cell variables, programs, and so on.), it returns `0` (zero).

NUMDIMS
Returns an INTEGER value which is one of the following depending on the type of object:

- For a dimensioned object, the number of dimensions.
- For all types of dimensions and dimension surrogates, NUMDIMS returns `1`.
- For a composite, it returns the number of base dimensions.
- For objects with no dimensions, it returns `0` (zero).

NUMVALS
Returns an `INTEGER` value that is the number of values or cells in the object. For a compressed composite or a variable dimensioned by a compressed composite, returns an `INTEGER` value that is the number of logical values in the object (that is, the value that would be returned if the composite was a b-tree composite). To retrieve the number of physical cells, use the PHYSVALS keyword.

OWNSPACE
Returns a BOOLEAN value that indicates whether a conjoint dimension or a composite is using private page space to store BTREE nodes, when that object is

using the BTREE index algorithm. In addition, whether the data that is associated with a composite, a conjoint dimension, a variable-width text dimension, a relation, or a variable-width text variable is stored in one or more private page spaces that are associated with that object.

PARTBY
When *name* is the name of a partitioned variable or a partition template object, returns the names of the partition dimensions as a multiline text value (one line for each dimension). For all other object types, returns `NA`.

PARTDIMS *partitions*
When *name* is the name of a partition template or a partitioned variable, returns the names of the dimensions of the specified partitions as a multiline text value (one line for each dimension). For all other object types, returns `NA`.

partitions is a multiline text value (one line for each partition name) that specifies which partitions you're asking about. When you specify a partition name that is not a valid partition in *partitions*, an error occurs.

PARTITION *partitions*
When *name* is the name of a partitioned variable or a partition template object, returns a textual description of the specified partitions. When called on a partition template, the returned description is similar to the DEFINE PARTITION TEMPLATE statement. When called on a partitioned variable, the returned description is similar to the DEFINE VARIABLE statement. For all other object types, returns `NA`.

partitions is a multiline text value (one line for partition name) that specifies which partitions you're asking about.When you specify a partition name that is not a valid partition in *partitions*, an error occurs.

PARTMETH
When *name* is the name of a partition template or a partitioned variable, returns a text value that is the method (`RANGE`, `LIST`, or `CONCAT`) by which it is partitioned. For all other object types, returns `NA`.

PARTNAMES
When *name* is the name of a partition template, returns a multiline `VARCHAR` containing the names of all the defined partitions. When *name* is the name of a partitioned variable, returns a multiline `VARCHAR` containing the names of all the partitions of the variable. For all other object types, returns `NA`.

Note: Not all of the partitions defined by a partition template necessarily exist in each partitioned variable. Calling `OBJ(PARTNAMES)` on a partitioned variable returns only those partitions that actually exist.

PARTRANGE *partitions* [*name*]
When *name* is the name of a a RANGE partition template or a RANGE partitioned variable, returns the values of the `LESS THAN` clause for each of the specified partitions. The return value is a multiline text value (one line for each partition). For all other kinds of partition templates and partitioned variables and all other object types, returns `NA`.

partitions is a multiline text value (one line for each partition name) that specifies which partitions you're asking about. When you specify a partition name that is not a valid partition in *partitions*, an error occurs.

PERIOD
For dimensions of type DAY, WEEK, MONTH, QUARTER, and YEAR, returns a `TEXT` value which is the type of the dimension plus an indication of multiple periods or phasing, if any. For objects other than DAY, WEEK, MONTH, QUARTER, or YEAR dimensions, it returns `NA`.

PHYSVALS
For a compressed composite or a variable dimensioned by a compressed composite, returns an `INTEGER` value that is the number of physical cells in the object. To retrieve the number of logical values, use the NUMVALS keyword.

PMTMAINTAIN
Returns a `TEXT` value which is the permission condition for the maintain permission associated with a dimension. When there is no maintain permission for the dimension, it returns `NA`.

PMTPERMIT
Returns a `TEXT` value which is the permission condition for the permit permission associated with an object. When there is no permit permission for the object, it returns `NA`.

PMTREAD
Returns a TEXT value which is the permission condition for the read permission associated with an object. When there is no read permission for the object, it returns NA.

PMTWRITE
Returns a TEXT value which is the permission condition for the write permission associated with an object. When there is no write permission for the object, it returns NA.

PRECISION
Returns an INTEGER value which is the precision of a NUMBER dimension or variable. The precision is the total number of digits. When the variable was defined without a precision specification, then OBJ returns NA.

PROGRAM
Returns a TEXT value which is the text of a program. For other types of objects, it returns NA.

PROPERTY *prop-name*
The value of the property specified by *prop-name* which is a text expression that specifies the name of the property. The data type of the return value is determined at runtime. (See "Converting Values Returned by OBJ (PROPERTY)" on page 19-16 for more information.) When the named property does not exist, it returns NA. See the PROPERTYTYPE argument. (Abbreviated PRP)

PROPERTYLIST
Returns a TEXT value which is a multiline text value that lists the properties associated with an object, one property on a line. The names are in uppercase letters and are stored in the collating sequence for ASCII characters. For objects without properties, it returns NA. (Abbreviated PRPLIST)

PROPERTYTYPE *prop-name*
The data type of *prop-name* which is a text expression that specifies the name of the property The type is derived from the expression used in the PROPERTY command. Possible return values are BOOLEAN, TEXT, ID, DATE, DATETIME, NUMBER, INTEGER, LONGINTEGER, DECIMAL, and SHORT.When the named property does not exist or has a value of NA, it returns NA. (Abbreviated PRPTYPE)

REFERS *text-expression*
Returns a multiline TEXT value which is the words found in a compilable object (for example, a program) that match the ones you specify in *text-expression*. REFERS

returns `NA` when it does not find any of the specified words, when the specified object is not a compilable object, or when the workspace does not contain any compilable objects. When you supply both arguments, REFERS searches only the specified object for the listed words. When you omit *object-name,* REFERS searches all the compilable objects in the current workspace.

- ***text-expression*** is a multiline `TEXT` expression that is the words for which it should search. Each line in the text value is considered a separate word to be searched for.

Tip: When you omit *object-name*, use REPORT, rather than SHOW, to produce the output. Because the return value of `OBJ(REFERS)` is dimensioned by the NAME dimension, the REPORT command will return output for each object in the workspace.

The search is not case-sensitive; REFERS treats `TEXTVAR` and `Textvar` as the same word. REFERS ignores all text that is included in a comment or enclosed in single quotes.

When, for *text-expression*, you specify a list of words that is the result of the OBJLIST function, you can produce a cross-reference for compilable objects in the current workspace.

SCALE
Returns an INTEGER value which is the scale of a NUMBER dimension or variable. A positive scale indicates the number of digits to the right of the decimal point. A negative scale indicates the number of rounded digits to the left of the decimal point. When the variable was defined without a scale specification, then OBJ returns `NA`.

SEGWIDTH {*dimension-name* | ALL}
Returns a single or multiline `TEXT` value which is default or user-specified segment size of a variable that has more than one dimension and that is associated with either a particular dimension or all dimensions. Each line begins with a segment-size (up to 11 digits) followed by the name of the associated dimension or composite. The dimension name is not included in the line when you specify a dimension and its dimensioned object. In that case only the segment value is returned. When the segment size is reported as zero, it means the default segment size is in effect, and therefore you may need to use CHGDFN to set an appropriate size for the variable's segments. When applied to an object other than a variable, this choice returns `NA`.

- ***dimension-name*** is a text expression that is the name of a dimension.
- **ALL** specifies all dimensions.

SPARSE
Returns a `TEXT` value which is a multiline text value that lists the composites used in the definition of an object.

SURROGATELIST *surrogate|dimension*
Returns a `TEXT` value which is a multiline text value that lists the surrogates defined for a dimension. The object-name can be the name of a surrogate or a dimension. When no surrogates are defined for the dimension, then OBJ returns `NA`.

TRIGGER [*triggering-event*]
Specify the *triggering-event* using one of the following keywords:

MAINTAIN
DELETE
PROPERTY
ASSIGN
BEFORE_UPDATE
AFTER_UPDATE

TRIGGER without a *triggering-event* keyword returns a `TEXT` value which consists of all the *triggering-event* keywords and trigger programs names associated with the object; or `NA` when the object does not have any trigger programs associated with it. TRIGGER with a *triggering-event* keyword returns a `TEXT` value that is the names of the trigger programs associated with the object event.

TYPE
Returns a `TEXT` value which is the object type of an object. Possible values include `AGGMAP`, `COMPOSITE`, `DIMENSION`, `FORMULA`, `MODEL`, `OPTION`, `PARTITION TEMPLATE`, `PROGRAM`, `RELATION`, `SURROGATE`, `VALUESET`, `VARIABLE` and `WORKSHEET`.

VNF
Returns a `TEXT` value which is the VNF (value name format) of a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR. For other types of objects, and for DAY, WEEK, MONTH, QUARTER, and YEAR dimensions with no VNF, it returns `NA`.

WIDTH
Returns an INTEGER value which is the width, in bytes, of the storage area of each value of an object:

- For dimensioned INTEGER and BOOLEAN variables that you defined with a width, it returns `1`.
- For dimensioned text variables and text dimensions that you defined with a width, it returns an integer between `1` and `4000`, which identifies the defined width.
- For all other objects, it returns `NA`.

object-name
A text expression that contains the name of the object in which you are interested. The object can be in any attached workspace. When you specify *object-name* as a text literal, you must enclose it in single quotes. (Oracle OLAP automatically converts the name to uppercase.) When you specify the name of a program as the *object-name* and you omit the quotes, Oracle OLAP runs the program and uses its return value as the name of the object to be supplied as *object-name*.

You can omit *object-name* when you are using the OBJ function as part of a statement, such as the LIMIT command, that loops through the NAME dimension. In this case, the return value is dimensioned by the NAME dimension in the current workspace.

Notes

Matching the Case of the Return Value

When you are checking the value returned by OBJ in a Boolean expression, you must be careful to check for the exact value returned by the OBJ function. This means you must match uppercase and lowercase exactly. For example, OBJ(DATA) returns the data type in uppercase.

```
OBJ(DATA) EQ 'INTEGER'
```

This is particularly important when you are limiting NAME to a particular group of objects. See Example 19–4, "Using OBJ to Select Objects" on page 19-19.

Converting Values Returned by OBJ (PROPERTY)

The return value of OBJ(PROPERTY) is like a worksheet value whose data type is determined at runtime. In some cases, the statement calling OBJ(PROPERTY) makes assumptions about its return value. In the following example, `market` is a dimension of TEXT values and `max` is a property with an INTEGER value.

```
LIMIT market TO OBJ(PROPERTY 'max' 'market')
```

Because the data type of the property is not the same data type as the dimension, the LIMIT command produces an error message stating that the return value is not a valid dimension value. To solve this problem, use the CONVERT function.

```
LIMIT market TO CONVERT(OBJ(PROPERTY 'max' 'market') INTEGER)
```

Examples

Example 19–1 OBJ With ISBY

For example, the following statement limits NAME to all the objects dimensioned by `month`.

```
LIMIT NAME TO OBJ(ISBY 'month')
```

You can use ISBY to find out if a dimension is a base dimension of a concat or conjoint dimension or a composite. For example, assume that you had a conjoint dimension named `proddist` whose base dimensions were `product` and `district`. In this case, the following statement returns `YES`.

```
SHOW OBJ(ISBY 'district' 'proddist')
```

You can use ISBY to find out if a dimension is a dimension of a relation or a model used in an aggmap. For example, assume that you had an aggmap called `myaggmap` and you wanted to find out if a dimension named `mydimension` was used in any relations or models within `myaggmap`. In this case, you could issue the following statement.

```
SHOW OBJ(ISBY 'mydimension' 'myaggmap')
```

To determine whether a specified dimension is a base dimension at any level, you must use ISBY with the RECURSIVE keyword. For example, assume that you had a conjoint dimension named `proddist.mon` whose base dimensions were `proddist` and `month` and a variable `proddist.sales` dimensioned by `proddist`. In this case, each of the following statements would return `NO`.

```
SHOW OBJ(ISBY 'district' 'proddist.mon')
SHOW OBJ(ISBY 'district' 'proddist.sales')
```

However, when you use ISBY with the RECURSIVE keyword, each of the following statements would return `YES`.

```
SHOW OBJ(ISBY RECURSIVE 'district' 'proddist.mon')
SHOW OBJ(ISBY RECURSIVE 'district' 'proddist.sales')
```

Example 19–2 Getting Information about a Variable

This example illustrates the use of several choices of the OBJ function to obtain information about the variable `sales`. The definition of `sales` is as follows.

```
DEFINE sales VARIABLE DECIMAL <month product district>
LD Sales Revenue
```

- The statement

  ```
  SHOW OBJ(TYPE 'sales')
  ```

  produces the following output.

  ```
  VARIABLE
  ```

- The statement

  ```
  SHOW OBJ(DATA 'sales')
  ```

  produces the following output.

  ```
  DECIMAL
  ```

- The statement

  ```
  SHOW OBJ(DIMS 'sales')
  ```

  produces the following output.

  ```
  MONTH
  PRODUCT
  DISTRICT
  ```

- The statement

  ```
  SHOW OBJ(ISBY 'product' 'sales')
  ```

  produces the following output.

  ```
  YES
  ```

- The statement

  ```
  SHOW OBJ(LD 'sales')
  ```

  produces the following output.

  ```
  Sales Revenue
  ```

Example 19–3 Returning the Name of the Object or the Type of the Object

Suppose `textvar` is a variable whose value is `geog`, which is the name of a dimension. Whether you enclose the word `textvar` in quotation marks determines whether the following `OBJ` function calls return the word `VARIABLE` (the type of object `textvar` is) or `DIMENSION` (the type of object `geog` is).

```
SHOW OBJ(TYPE 'textvar')
VARIABLE

SHOW OBJ(TYPE textvar)
DIMENSION
```

Example 19–4 Using OBJ to Select Objects

This example uses OBJ and DESCRIBE to look at the definitions of all the relations in a workspace. The Oracle OLAP statements

```
LIMIT NAME TO OBJ(TYPE) EQ 'RELATION'
DESCRIBE
```

produce the following output.

```
DEFINE REGION.DISTRICT RELATION REGION <DISTRICT>
LD REGION for each DISTRICT

DEFINE DIVISION.PRODUCT RELATION DIVISION <PRODUCT>
LD DIVISION for each PRODUCT

DEFINE MLV.MARKET RELATION MARKETLEVEL <MARKET>

DEFINE MARKET.MARKET RELATION MARKET <MARKET>
LD Self-relation for the Market Dimension
```

Example 19–5 Counting Compiled Objects

The following statements count how many compilable objects in your workspace are compiled and how many are not compiled. Each statement loops over the objects in the current workspace. The OBJ function returns `YES` for each object that is compiled, NO for each compilable object that is not compiled, and NA for objects that are not compilable. When NASKIP is YES (the default), the COUNT function in the first statement counts the number of YES values that are returned by OBJ, and in the second statement it counts the number of NO values that are returned.

```
SHOW COUNT(OBJ(ISCOMPILED))
SHOW COUNT(NOT OBJ(ISCOMPILED))
```

Example 19–6 OBJ with REFERS

The following statement searches the compilable objects in the current workspace for references to the objects in all the attached workspaces. The output lists the non-compilable objects in the current workspace too, but the return value for them is NA.

```
REPORT OBJ(REFERS OBJLIST(AW(LIST)))
```

In the following example, OBJ(REFERS) tells you whether var1, var2, or var3 appears in the myprog program. The return value of OBJ(REFERS) is a multiline text value that contains the references it finds. When only var1 and var3 appear in the program, then the return value contains those two names, each on a separate line. The statement

```
SHOW OBJ(REFERS 'var1\nvar2\nvar3' 'myprog')
```

produces the following output.

```
VAR1
VAR3
```

When you do not specify the name of a program or formula to be searched, OBJ(REFERS) returns a single-line or multiline text value for *each object* in the

NAME dimension of the current workspace. For objects that are not programs or formulas, NA is returned. The statement

```
REPORT OBJ(REFERS 'var1\nvar2\nvar3')
```

produces the following output.

```
               OBJ(REFERS
                 'var1
                  var2
NAME           var3' )
-------------- ----------
PRODUCT        NA
DISTRICT       NA
DIVISION       NA
LINE           NA
QUARTER        NA
REGION         NA
YEAR           NA
MONTH          NA
   ...
MYPROG         VAR1
               VAR3
VAR1           NA
VAR2           NA
VAR3           NA
```

Example 19–7 OBJ with PROPERTY

In the following example, OBJ(PROPERTY) returns information about the `decplace` property of the `actual` variable. (See PROPERTY.) The user created this property to store the number of decimal places and now wants to obtain that value to produce a report of the `actual` variable.

The statements

```
CONSIDER actual
PROPERTY 'decplace' 4
LIMIT month TO FIRST 1
LIMIT division TO 'Camping'
REPORT ACROSS month W 20 DECIMAL OBJ(PROPERTY 'decplace' -
'actual') actual
```

produce the following output.

```
DIVISION: CAMPING
               -------ACTUAL-------
               -------MONTH--------
LINE                  JAN 95
-------------- --------------------
Revenue                533,362.8800
Cogs                   360,810.6600
Gross.Margin           172,552.2200
Marketing               37,369.5000
Selling                 89,007.3800
R.D                     24,307.5000
Opr.Income              21,867.8400
Taxes                   15,970.3900
Net.Income               5,897.4500
```

Example 19–8 OBJ with SEGWIDTH

The following statements show how to change and display segment size values for all of a variable's dimensions.

```
CHGDFN sales SEGWIDTH 150 5000 50
SHOW OBJ(SEGWIDTH ALL 'sales')
```

These statements produce the following output.

```
 150 MONTH
5000 PRODUCT
  50 DISTRICT
```

The following statement shows how to obtain the segment size value for a specific dimension.

```
SHOW OBJ(SEGWIDTH 'product' 'sales')
```

This statement produces the following output.

```
5000
```

The following statement shows how to obtain a list of segment sizes for every multidimensional variable or relation associated with the dimension.

When *object-name* is not specified, you need to use REPORT rather than SHOW to obtain a value for each object in the NAME dimension.

```
REPORT OBJ(SEGWIDTH 'product')
```

This statement produces the following output.

```
NAME           OBJ(SEGWIDTH 'product')
-------------- -----------------------
SALES          5000
SALES.FORECAST 5000
SALES.PLAN     5000
SHARE          5000
UNITS          5000
UNITS.M        0
    ...
```

The following statement shows how to produce a list of segment sizes for all dimensions in the current workspace.

```
REPORT OBJ(SEGWIDTH ALL)
```

This statement produces the following output.

```
NAME           OBJ(SEGWIDTH ALL)
-------------- -----------------
SALES          150 MONTH
               5000 PRODUCT
               50 DISTRICT

SALES.FORECAST 150 MONTH
               5000 PRODUCT
               50 DISTRICT
     ...
```

OBJLIST

The OBJLIST function provides a list of the objects that are contained in one or more workspaces that you specify. The specified workspaces must be currently attached when you use the function.

The result, a multiline TEXT value, can be used as an argument to the OBJ function with the REFERS keyword. This usage helps in producing a cross-reference list for compilable objects, such as programs and models, in the current workspace.

Return Value

TEXT

Syntax

OBJLIST[(*text-expression*)]

Arguments

text-expression
A text expression that contains a single name or several names of currently attached workspaces. Each workspace name must be on a separate line of a multiline TEXT value. When you do not supply this argument, OBJLIST uses the current workspace name.

Notes

Output from OBJLIST

The list of workspace objects returned by OBJLIST has duplicates removed and it is sorted in ascending order.

Listing All Workspace Objects

OBJLIST always returns the names of all the objects in a given workspace, even when you have limited its NAME dimension.

When a Workspace Is Not Attached

When *text-expression* includes the name of a workspace that is not attached, OBJLIST does not return a value. Instead, it signals an error.

Examples

Example 19–9 Listing Objects in Three Workspaces

In the following example, OBJLIST returns a multiline TEXT value that includes all the objects in the three workspaces specified: `mycode`, `mydata`, and `mytools`. The statement

```
SHOW OBJLIST('mycode\nmydata\nmytools')
```

produces the following output.

```
ACTUAL
ADDFIVE
ADVERTISING
BUDGET
CITYLIST
CITYREPINIT
CITYREPS
    ...
YEAR
```

Example 19–10 Listing Referenced Objects

In the following example, OBJ(REFERS) returns a multiline TEXT value that contains every object from the `mycode`, `mydata`, and `mytools` workspaces that is referenced in the `myprog` program. The statement

```
SHOW OBJ(REFERS OBJLIST('mycode\nmydata\nmytools') 'myprog')
```

produces the following output.

```
ACTUAL
BUDGET
 ...
YEAR
```

OBSCURE

The OBSCURE function provides two mechanisms for encrypting a single-line text expression. Depending on the mechanism you use, OBSCURE can also restore the encrypted value to its original form.

Return Value

TEXT

Note: The return value of the OBSCURE function always has a text data type. However, unless you specify the TEXT keyword, the actual value returned by `OBSCURE(HASH)` and `OBSCURE(HIDE)` is binary. When you want to be able to manage these encrypted values as text (for example, when you want to be able to store them in a text file), you must specify the TEXT keyword. See Example 19–13, "Generating Text Data" on page 19-29.

Syntax

OBSCURE({HASH|HIDE|UNHIDE} [TEXT] *seed-exp input-exp*)

Arguments

HASH

Specifies that Oracle OLAP encrypts the input text expression according to the seed expression that you specify. With the HASH keyword:

- Encrypted values *cannot be restored* to their original form.
- The same seed expression and input text always produce the *same* result.

A typical application would be a local password validation scheme. You can use OBSCURE with the HASH keyword to encrypt passwords, store them, and then validate the passwords presented by users against the stored encrypted values. See Example 19–11, "Using HASH" on page 19-28.

HIDE
Specifies that Oracle OLAP encrypts the input text expression according to the seed expression that you specify. With the HIDE keyword:

- Encrypted values *can be restored* to their original form with UNHIDE.
- The same seed expression and input text always produce *different* results.

The HIDE keyword provides a mechanism for storing values in encrypted form while actually comparing their *unencrypted* values. A typical application would be a remote password validation scheme. You could use OBSCURE with the HIDE keyword to store passwords in encrypted form on a local system. You could then pass them in encrypted form to a remote system for validation against unencrypted criteria on the host. See Example 19–12, "Using HIDE" on page 19-28.

UNHIDE
When specified with the original seed expression, restores values encrypted with the HIDE keyword to their original form. See "Restoring Text" on page 19-27.

TEXT
The TEXT keyword causes OBSCURE to convert binary data to text, such that the return value consists only of text data. When you do not specify the TEXT keyword, the output of OBSCURE is binary data. See "Restoring Text" on page 19-27, and "Generating Text Data" on page 19-29.

seed-exp
A single-line text expression that is used as a seed value in the encryption of the input text expression.

input-exp
A single-line text expression to be encrypted or restored by OBSCURE.

Notes

Restoring Text
When you have used OBSCURE(HIDE) with the TEXT keyword to encrypt a text expression, you must also specify the TEXT keyword with `OBSCURE(UNHIDE)` to restore the encrypted expression to its original form.

OBSCURE and C2 Security
The OBSCURE function does *not* conform to the C2 security level specified by the Department of Defense.

Case Sensitivity

Both the seed expression and the text expression that you provide as input to OBSCURE are case-sensitive.

Examples

Example 19–11 Using HASH

The following example shows how you could use the HASH keyword to store a password in encrypted form in the variable `first_user`. When a new user attempts to log in, his password is encrypted with the HASH keyword and compared to the value stored in `first_user`. When the values are the same, the program `validate_user`, which allows the new user to log in, is invoked.

```
passvar = 'JoeSmith'
first_user = OBSCURE(HASH 'lxyz' passvar)
 ...
'Run a login procedure that assigns a password
'presented by a user to the variable NEW_USER
'and checks it against the stored encrypted value
 ...
IF OBSCURE(HASH 'xyz' new_user) EQ first_user
   THEN validate_user
   ELSE deny_access
```

Example 19–12 Using HIDE

You can encrypt the name `JSmith` with the seed expression'`abc` and restore it to its original form, using the following statements.

```
DEFINE pswobsc VARIABLE TEXT
pswobsc = OBSCURE(HIDE 'abc' 'JSmith')
SHOW OBSCURE(UNHIDE 'abc' pswobsc)
```

This SHOW statement generates the following output.

```
jsmith
```

Example 19–13 Generating Text Data

The following statements illustrate the use of the TEXT keyword.

```
DEFINE encrypted_text VARIABLE TEXT
DEFINE unencrypted_text VARIABLE TEXT

unencrypted_text = 'max'
encrypted_text = OBSCURE(HIDE TEXT 'XXXX' unencrypted_text)
SHOW encrypted_text
```

This SHOW statement generates the following output.

```
c5WF/XfABuY
```

The same statements without the TEXT keyword would produce binary output from the SHOW statement.

OKFORLIMIT

The OKFORLIMIT option controls whether you can limit the dimension you are looping over within an explicit FOR loop.

Data type

BOOLEAN

Syntax

```
OKFORLIMIT = {NO|YES}
```

Arguments

NO
You cannot limit the dimension you are looping over within an explicit FOR loop. (Default)

YES
You can limit the dimension you are looping over within an explicit FOR loop.

Notes

Related Statements

See the TEMPSTAT command to set the status of the dimension you are looping over in a loop that is generated by a REPORT command.

Examples

Example 19–14 Allowing Limits Within a Loop

The following program excerpt sets OKFORLIMIT to `YES`, thereby allowing the user to limit `market` within a FOR loop.

```
 ...
OKFORLIMIT = YES
FOR market
    DO
      LIMIT market TO CHILDREN USING market.market
      REPORT market
    DOEND
 ...
```

OKNULLSTATUS

The OKNULLSTATUS option determines whether Oracle OLAP allows a dimension status list to be set to null. The default is to not allow an empty status list. When null status lists are not allowed, Oracle OLAP produces an error message when you execute a LIMIT command that would result in a null status list.

Data type

BOOLEAN

Syntax

OKNULLSTATUS = {YES|<u>NO</u>}

Arguments

YES
Indicates that null status lists are allowed. With this setting, when you execute a LIMIT command (without the IFNONE argument) that results in a dimension status list being null, the status list *is* set to null, and no error message is produced.

NO
Indicates that null status lists are not allowed. With this setting, when you execute a LIMIT command (without the IFNONE argument and without the NULL keyword) that would result in a dimension status list being null, the status list is *not* changed and an error message is produced. (Default)

Notes

Conditions When OKNULLSTATUS Has No Effect

The value of OKNULLSTATUS has no effect in the following situations.

- When a LIMIT command includes an IFNONE argument. An IFNONE argument indicates that program execution should not take its normal course when a dimension status list or valueset were to be set to null. Therefore, when IFNONE is present, Oracle OLAP branches to the IFNONE label and does not set the status list or valueset to null, even when OKNULLSTATUS is `YES`.

- When a LIMIT command uses the NULL keyword to set a dimension status list to null. The status list is set to null, and no error message is produced, even when OKNULLSTATUS is `NO`.
- When a LIMIT command sets a valueset to null (unless the IFNONE argument is used). The valueset is set to null, and no error message is produced, even when OKNULLSTATUS is `NO`.
- When a LIMIT function is specified to return a null dimension status list. The value returned is `NA`, and no error message is produced, even when OKNULLSTATUS is `NO`.

NULL Status

See the LIMIT command for more information about using null status in dimensions and valuesets.

Examples

Example 19–15 Using OKNULLSTATUS

The following statement turns off error messages about the null status of dimensions and allows dimension status lists to be set to null.

```
OKNULLSTATUS = YES
```

ONATTACH

A program that you write and that Oracle OLAP checks for by name when an AW ATTACH statement executes. Depending on the value returned by the program, Oracle OLAP executes the code within the program immediately after attaching the analytic workspace.

Note: Oracle OLAP checks for other programs when a user attaches a workspace. See "Programs Executed When Attaching Analytic Workspaces" on page 8-39 for more information.

Returns

`BOOLEAN`

`TRUE` when Oracle OLAP has successfully set up and attached the analytic workspace; or `FALSE` when it has not or when the `onattach` program has thrown an exception.

Note: You are encouraged to use the normal return values rather than relying on exceptions to create a return value of `FALSE`.

Syntax

To define a program with the name `ONATTACH` use the syntax shown in DEFINE PROGRAM. Code the actual program as a user-defined function with the following argument.

ONATTACH ({READ|WRITE|EXCLUSIVE|MULTI} *password*)

Arguments

See AW ATTACH for explanations of the attachment modes (that is, READ, WRITE, EXCLUSIVE, and MULTI) and *password*.

Notes

Creating an ONATTACH program

A program with the name of `onattach` does not exist within an analytic workspace unless you define and write one. You write a `onattach` as a user-defined functions that returns a BOOLEAN value. You can use the return value to indicate to Oracle OLAP whether or not the user has the right to attach the workspace.

Depending on the statements in the `onattach` program, the user is granted or denied access to specific objects or sets of object values.For multiwriter attachment, you can use ACQUIRE statements to provide access to individual workspace objects. For read-only and read/write attachment, you can use PERMIT commands that grant or restrict access to individual workspace objects

Note: All of the objects referred to in a given `onattach` program must exist in the same analytic workspace.

Examples

For examples of how attachment programs behave, see Example 8–14, "Startup Programs" on page 8-42.

OUTFILE

The OUTFILE command lets you redirect the text output of statements to a file.

Syntax

OUTFILE [APPEND|EOF] *file-name* [NOCACHE] [NLS_CHARSET *charset-exp*]

Arguments

APPEND
Specifies that the output should be added to the end of an existing disk file. When you omit this argument, the new output replaces the current contents of the file. APPEND has no effect when the file does not already exist or when you specify EOF.

file-name
A text expression that is the name of the file to which output should be written. Unless the file is in the current directory, you must include the name of the directory object in the name of the file.

Note: Directory objects are defined in the database, and they control access to directories and file in those directories. You can use the CDA command to identify and specify a current directory object. Contact your Oracle DBA for access rights to a directory object where your database user name can read and write files.

EOF
The current outfile is closed and output is redirected to the default outfile.

NOCACHE
Specifies that Oracle OLAP should write lines to the outfile as they are generated. Without this keyword, Oracle OLAP reduces file I/O activity by saving text and writing it periodically to the file. The NOCACHE keyword slows performance significantly, but it ensures that every line is immediately recorded in the outfile. This argument must be specified after*file-name*

NLS_CHARSET *charset-exp*
Specifies the character set that Oracle OLAP will use when writing data to the file specified by *file-name*. This allows Oracle OLAP to convert the data accurately into that character set. This argument must be specified after *file-name*. When this argument is omitted, then Oracle OLAP writes the data to the file in the database character set, which is recorded in the NLS_LANG option.

Notes

File Reading and Writing Options

A number of options are important during file read and write operations. These options are listed in Table 15–1, " File Reading and Writing Options" on page 15-6.

Permission Programs: Copying to and from Analytic Workspaces When you export PERMIT_READ or PERMIT_WRITE programs which are hidden, they are empty when imported. Additionally, when you outfile PERMIT_READ or PERMIT_WRITE programs which are hidden, then they are empty when infiled.

> **Tip:** Rename PERMIT_READ and PERMIT_WRITE programs before using EXPORT (to EIF) or OUTFILE. After copying the programs to an analytic workspace using IMPORT (from EIF) or INFILE.

Current Outfile Identifier

As a first step, every OUTFILE command closes the current outfile. When OUTFILE opens a new outfile on disk, it automatically assigns to it an arbitrary integer as its file unit number. The current file unit number is held in the OUTFILEUNIT option.

Appending to an Outfile

When you send output to a file and then send output to a second file, the first file does not remain open. To resume sending output to the first file, you must execute another OUTFILE command and include the APPEND *file-name* phrase.

Automatic Closing of Outfile

When you use OUTFILE *file-name* to direct output to a disk file, OUTFILE closes any outfile currently open. This happens even when the new file is not actually opened (as when you specify an invalid *file-name* in the OUTFILE command).

Paging Options and Redirected Output

The paging options control the organization of text output in pages. Examples are: BMARGIN, LINENUM, LINESLEFT, PAGESIZE, PAGENUM, PAGEPRG, PAGING, TMARGIN, and LSIZE. The paging options have a separate value for each separate outfile. When you set one of the paging options to control output to a disk file, the new value remains in effect until you use the OUTFILE command again to redirect output. At this point, the paging option returns to its default value. Therefore, when you want a paging option to have a particular value for a disk file, you generally have set it after you execute the OUTFILE command.

Line Length

The maximum line length in Oracle OLAP is 4000 characters.

Current and Default Outfiles

The current outfile is the destination for the output of statements, such as REPORT and DESCRIBE, that produce text. When you have not used the OUTFILE command to send output to a file, Oracle OLAP uses your default outfile.

Examples

Example 19–16   Sending a Report to an Output File

In this example, you want to send the output of a REPORT command to an output file.

```
OUTFILE 'budget.rpt'
REPORT budget
OUTFILE EOF
```

Example 19–17   Directing Output to a File

Suppose you have a program called `year.end.sales`, and you want to save the report it creates in a file. Type the following commands to write a file of the report. In this example, `userfiles` is a directory object and `yearend.txt` is the name of the file.

```
OUTFILE 'userfiles/yearend.txt'
year.end.sales
OUTFILE EOF
```

Now the file contains the `year.end.sales` report. You can add more reports to the same file with the `APPEND` keyword for OUTFILE. Suppose you have another program called `year.end.expenses`. Add its report to the file with the following commands. Remember that without `APPEND`, the OUTFILE command overwrites the expense report.

```
OUTFILE APPEND 'userfiles/yearend.txt'
year.end.expenses
OUTFILE EOF
```

OUTFILEUNIT

(Read-only) The OUTFILEUNIT option holds the file unit number of the current OUTFILE destination, set by the last OUTFILE command. The first time you redirect output to a given file, OUTFILE assigns that file an arbitrary integer as a file unit number.

Data type

INTEGER

Syntax

OUTFILEUNIT

Notes

OUTFILE and OUTFILEUNIT

You automatically change the setting of OUTFILEUNIT whenever you specify a different file with the OUTFILE command. For example, after the statement `OUTFILE` *`myfilename`*, the value of OUTFILEUNIT is the file unit number assigned to *myfilename.*

Examples

Example 19–18 Using OUTFILEUNIT with FILEQUERY

Suppose you have saved the file unit number for a file in a variable called `filenum`. Your current outfile is another disk file. You want to set the value of PAGEPRG for the first file to the value that it has for the current outfile. Because the file unit number for the current outfile is contained in the OUTFILEUNIT option, you can use FILEQUERY with the OUTFILEUNIT number to get the PAGEPRG setting for the current outfile.

```
FILESET filenum PAGEPRG FILEQUERY(OUTFILEUNIT PAGEPRG)
```

PAGE

The PAGE command forces a page break in output when PAGING is set to `YES`. An optional argument to PAGE specifies a conditional page break based on how many lines are left on the page.

The PAGE command is commonly used in report programs. It is meaningful only when PAGING is set to `YES` and only for output from statements such as REPORT and LISTNAMES.

Syntax

PAGE [*n*]

Arguments

n

A positive integer expression that indicates that a page break should occur only when there are fewer than *n* lines left on the current page. When the number of lines left equals or exceeds *n*, no page break occurs. See Example 19–19, "Keeping Lines Together" on page 19-42.

Notes

Top of Page

No page break occurs when you are already at the top of a page when the PAGE command is executed.

Producing the Header

The PAGE command signals that further output should be produced on a new page, but it does *not* produce a header on the new page unless there is further output. When there is further output, Oracle OLAP produces the heading that is defined by the current PAGEPRG program and then starts producing the output.

Examples

Example 19–19 Keeping Lines Together

Suppose you have 12 lines of data that would be hard to read when interrupted by a page break, so you want to prevent such an interruption. Use the `PAGE 12` statement immediately before the statements that produce the 12 lines of data. A page break will occur before the 12 lines of data only when there are less than 12 lines left on the page. When there are 12 lines or more left at that point, output will continue on the same page.

Example 19–20 Forcing a Page Break

The following lines from a report program force a page break at the start of each loop for `district`. This makes the report for each district start at the top of a page. (The report program uses a heading program called `report.head` to create a customized heading. See PAGEPRG for information on customized heading programs.)

```
PUSH PAGING PAGEPRG
PAGING = YES
PAGEPRG = 'report.head'
FOR district
   DO
   PAGE
   ROW district
   BLANK
   FOR month
      ROW WIDTH 8 month sales sales.plan
   DOEND
PAGE
POP PAGING PAGEPRG
```

PAGENUM

The PAGENUM option holds the current page number of output. You can use PAGENUM with PAGEPRG to produce the page number on each page of a report. The PAGENUM option is meaningful only when PAGING is set to `YES` and only for output from statements such as REPORT and LISTNAMES.

Data type

INTEGER

Syntax

PAGENUM = *n*

Arguments

n
An integer expression that specifies the page number to use for the next page of output. The default is `1`.

Notes

Starting with Page 1

When you are sending output to the default outfile, set both PAGENUM and LINENUM to 1 whenever you want to produce a report starting on page 1. You can set these options in the initialization section of your report program. When you use an OUTFILE command to send output to a file, PAGENUM is automatically set to `1`.

Setting PAGENUM in Mid-Page

The value of PAGENUM is incremented automatically when the last line of output has been generated on a page. When you set PAGENUM when an output page is only partially full, the value of PAGENUM will be incremented by 1 before the next page is produced. This means you usually have to set PAGENUM to a value of one less than the number you want to show on the following page.

The Effect of PAGING

When you set PAGING to NO, PAGENUM stops counting and keeps its last value. When you reset PAGING to YES, PAGENUM resumes counting at the page number where it left off.

The Effect of OUTFILE

When you use the OUTFILE command to direct output to a file, PAGENUM is set to 1 for the file. When you use the OUTFILE command with the EOF keyword to redirect output to the default outfile, PAGENUM will contain the number that it last held for the default outfile.

Examples

Example 19–21 Changing the Heading for Page 2

Suppose you want each page of a report to have a standard running page heading and a custom title, and pages after the first page to also have the heading "(Continued)". You can define a page heading program called report.head that uses the PAGENUM value to determine when to add the "(Continued)" heading.

```
DEFINE report.head PROGRAM
PROGRAM
STDHDR
BLANK
PAGING = YES
HEADING WIDTH LSIZE CENTER 'Annual Sales Report'
BLANK
IF PAGENUM GT 1
   THEN HEADING WIDTH LSIZE CENTER '(Continued)'
BLANK
END
```

In your report program, set the PAGEPRG option to use the report.head program.

```
PAGEPRG = 'report.head'
```

When you run the report program, each page after the first page starts with a heading such as the following.

```
15JAN95 15:05:16                                     Page  2
                       Annual Sales Report

                           (Continued)
```

PAGEPRG

The PAGEPRG option holds the name of a program or the text of a statement to be executed at the beginning of each page of output. You can use this program or statement to create titles and column headings on multiple pages of a report. A program can also contain other statements appropriate for execution at the start of every page. Normally, you set the value of PAGEPRG in the initialization section of a report program.

The PAGEPRG option is meaningful only when PAGING is set to `YES` and only for output from statements such as REPORT and LISTNAMES.

Data type

TEXT

Syntax

PAGEPRG = {'*program*'|'*statement*'|'NONE'|<u>'STDHDR'</u>}

Arguments

program
The name of a program to be executed after every page break. When you specify the program name as a text expression, you can omit the single quotes.

statement
The text of a statement to be executed after every page break. When you specify the statement as a text expression, you can omit the single quotes.

NONE
Indicates that no statement or program is executed automatically after a page break.

STDHDR
Makes STDHDR the program name that PAGEPRG stores. You can also set PAGEPRG to `'DEFAULT'` to make STDHDR the program name that PAGEPRG stores. STDHDR produces a heading with the date and time on the left and the page number on the right. (Default)

Notes

Using STDHDR in a Header Program

When you create a PAGEPRG program, you can include the STDHDR program as a line in the program. Generally, you place STDHDR before the other statements that will produce the custom heading. See Example 19–22, "Creating a Custom Heading" on page 19-47.

Keeping Header Information Current

You can use Oracle OLAP features such as TODAY, TOD, and PAGENUM in a program that is specified by the PAGEPRG option. You can also have a header program that accepts arguments, such as the title for a particular report. In this case you would set the PAGEPRG option to a text expression that invokes the report header program with arguments. See Example 19–23, "Using Program Arguments" on page 19-47.

Output to the Default Outfile

When you set PAGEPRG for the default outfile, the new value remains in effect until you reset it, regardless of intervening OUTFILE commands that send output to a file. That is, the value of PAGEPRG is automatically saved for the default outfile.

Output to a File

To set PAGEPRG for a file, first make the file your current outfile by specifying its name in an OUTFILE command, then set PAGEPRG to the desired value. The new value remains in effect until you reset it or until you use an OUTFILE command to direct output to a different outfile. When you direct output to a different outfile, PAGEPRG returns to its default value of `'STDHDR'` for the file.

Examples

Example 19–22 Creating a Custom Heading

Suppose you want each page of a report to include both the standard running page heading and the title "Annual Sales Report." To accomplish this, create a program called `report.head`.

```
DEFINE report.head PROGRAM
PROGRAM
STDHDR
BLANK
HEADING WIDTH LSIZE CENTER 'Annual Sales Report'
BLANK
IF PAGENUM GT 1
   THEN HEADING WIDTH LSIZE CENTER '(Continued)'
BLANK
END
```

Specify this program to execute after every page break by setting the PAGEPRG option in the report program. You can include PUSH and POP commands to save the PAGEPRG setting that is active.

```
PUSH PAGEPRG PAGING
PAGEPRG = 'report.head'
PAGING = YES
      ... (body of report program)
  POP PAGEPRG PAGING
```

When you run the report, each page will contain the following heading.

```
15JAN98  15:05:16                                    Page 1

                       Annual Sales Report
```

Each page after the first page will also contain the subheading "(Continued)" because of the PAGENUM test in the IF statement.

Example 19–23 Using Program Arguments

As an alternative to specifying the report name in the `report.head` program, you can pass the report name to the `report.head` program from your report program. You can do this by setting the PAGEPRG option to a text expression that invokes the report.head program with the report name as an argument. Suppose your report program contains the following statement.

```
PAGEPRG = 'CALL report.head(\'Annual Sales Report\')'
```

Then you can change the first few lines of the report.head program to the following.

```
ARGUMENT titlevar TEXT
STDHDR
BLANK
HEADING WIDTH LSIZE CENTER titlevar
```

PAGESIZE

The PAGESIZE option specifies the size of a page of output. The value of PAGESIZE is the number of output lines to be produced on each page. PAGESIZE is usually used in the initialization section of report programs. The PAGESIZE option is meaningful only when PAGING is set to `YES` and only for output from statements such as REPORT and LISTNAMES.

Data type

INTEGER

Syntax

PAGESIZE = *n*

Arguments

n

An integer expression that specifies the number of output lines on a page; *n* includes the top and bottom margins (controlled by the TMARGIN and BMARGIN options). The default is 66 lines, which is suitable for printing report output on 8 1/2" by 11" paper.

Notes

Usable Output Lines

When you use the standard heading and the default settings for the PAGESIZE, TMARGIN, and BMARGIN options, the total number of usable output lines is 61.

```
                              Output Lines
Lines from PAGESIZE                     66
Lines for TMARGIN                      - 2
Lines for the standard heading         - 2
Lines for BMARGIN                      - 1
Lines available for output              61
```

Eliminating Headings and Page Breaks

You can produce pages with no headings by using the statement `PAGEPRG='NONE'` or suppress page breaks entirely by using the statement `PAGING = NO`.

Forcing a Page Break

When PAGING is set to `YES`, you can force a page break at any point in a page of output by using a PAGE command.

The Effect of PAGESIZE on LINESLEFT

PAGESIZE also controls the LINESLEFT option. When PAGESIZE is changed, Oracle OLAP adjusts LINESLEFT accordingly.

Output to the Default Outfile

When you set PAGESIZE for the default outfile, the new value remains in effect until you reset it, regardless of intervening OUTFILE commands that send output to a file. That is, the value of PAGESIZE is automatically saved for the default outfile.

Output to a File

To set PAGESIZE for a file, first make the file your current outfile by specifying its name in an OUTFILE command, then set PAGESIZE to the desired value. The new value remains in effect until you reset it or until you use an OUTFILE command to direct output to a different outfile. When you direct output to a different outfile, PAGESIZE returns to its default value of `66` for the file.

Examples

Example 19–24 Printing on Legal Paper

In this example, you want to produce a report that you will print on legal-size paper (8 1/2" by 14"). Include the following statement in the initialization section of your report program.

```
PAGESIZE = 84
```

PAGING

The PAGING option controls the production of paged output in Oracle OLAP. When you set PAGING to `YES`, output from statements such as DESCRIBE, REPORT, ROW command, HEADING, SHOW, and LISTNAMES is produced in a page-oriented format. Output is produced in page-size segments with standard top and bottom margins and headings. You can use a variety of paging-related options to change the size of the page, the size of the margins, and the headings on each page.

Paging is useful primarily for making output more attractive when you plan to print output that you send to a file. However, you can also send paged output to the default outfile. Normally you would set the PAGING option in the initialization section of a report program to turn paging on for your report.

Data type

BOOLEAN

Syntax

PAGING = {YES|NO}

Arguments

YES
Produces output with page breaks, top and bottom margins, and page headings.

NO
Produces output that contains no page breaks, top and bottom margins, or page headings. Output is continuous, one line after another. (Default)

Notes

Output to the Default Outfile

When you set PAGING for the default outfile, the new value remains in effect until you reset it, regardless of intervening OUTFILE commands that send output to a file. That is, the value of PAGING is automatically saved for the default outfile.

Output to a Different File

To set PAGING for a file, first make the file your current outfile by specifying its name in an OUTFILE command, then set PAGING to the desired value. The new value remains in effect until you reset it or until you use an OUTFILE command to direct output to a different outfile. When you direct output to a different outfile, PAGING returns to its default value of `NO` for the file.

Paging-Related Options

Oracle OLAP uses default values for page length, page headings, and top and bottom margins. You can change these values by setting the PAGESIZE, PAGEPRG, TMARGIN, and BMARGIN options. Other paging options that become meaningful when PAGING is turned on are LINENUM, LINESLEFT, and PAGENUM.

Paging for the Current Outfile

The value of PAGING for the current outfile determines whether the paging-related options will be used. You must set PAGING to `YES` for the current outfile in order to make the paging options take effect.

Changing Outfiles

When you use the OUTFILE command to direct output to a file, all the paging-related options are set to their default values for the file. When you use the OUTFILE command with the EOF keyword to redirect output to the default outfile, the paging-related options will contain the values that they last held for the default outfile.

The LINENUM Option

When you set PAGING to `NO`, the value of the LINENUM option continues to increment as more output lines are produced. When you set PAGING to `YES`, LINENUM is set to `1` and it begins counting lines on the current page.

The LINESLEFT Option

When you set PAGING to `NO`, the LINESLEFT option is set to PAGESIZE, and it keeps this value until PAGING is set to `YES`. When you set PAGING to `YES`, LINESLEFT begins counting the lines left on the current page.

The PAGENUM Option

When you set PAGING to `NO`, the PAGENUM option stops counting and retains its current value. When you set PAGING to `YES`, PAGENUM resumes counting at the page number where it left off.

Examples

Example 19–25 Setting Paging Options

Suppose you are writing a report program and you want to control page breaks and the top margin. You can include the following lines in the initialization section of your program. These lines send output to a file named `repfile.txt`, turn the PAGING option on, and change the page size and top margin.

```
OUTFILE 'repfile.txt'
PAGING = YES
PAGESIZE = 84
TMARGIN = 6
```

PARENS

The PARENS option controls whether negative numbers are represented in output with parentheses or a minus sign.

Data type

BOOLEAN

Syntax

PARENS = {YES|<u>NO</u>}

Arguments

YES
Encloses negative values in parentheses, instead of using a minus sign.

NO
Uses a minus sign to represent negative values. (Default)

Notes

Overriding PARENS
The setting of the PARENS option is overridden by a PAREN or NOPAREN attribute in a HEADING, REPORT, or ROW command. The PAREN attribute specifies the use of parentheses; the NOPAREN attribute specifies the use of a minus sign.

Allowing Space for Parentheses
When you use parentheses to represent negative values in a report, Oracle OLAP lines up the positive and negative values in the column. To do this, it reserves the right-most character in each numeric column for the closing parenthesis. The column is always reserved, even when there are no negative values in the output. Consequently, each value requires more space than when you use the minus sign, and you might need to increase your column width to accommodate your data.

Examples

Example 19–26 Showing Negative Values in Parentheses

In a report, you would like to show negative values in parentheses, so you first set PARENS to `YES`.

```
LIMIT line TO 'Cogs'
LIMIT division TO 'Sporting'
LIMIT month TO 'Jan96' TO 'Jun96'
PARENS = YES
DECIMALS = 0
REPORT DOWN month budget actual budget-actual
```

These statements produce the following output.

```
DIVISION: SPORTING
               --------------LINE--------------
               --------------COGS--------------
                                     BUDGET-ACT
MONTH            BUDGET     ACTUAL      UAL
-------------- ---------- ---------- ----------
Jan96            279,773    287,558     (7,785)
Feb96            323,982    315,299      8,683
Mar96            302,178    326,185    (24,007)
Apr96            386,101    394,544     (8,443)
May96            433,998    449,862    (15,864)
Jun96            448,042    457,348     (9,305)
```

PARSE

Use the PARSE command to parse a specified group of expressions. When the argument can be parsed, PARSE determines the number of expressions in it and their text, object type, data type, and the number and names of their dimensions. This information is stored internally by Oracle OLAP, but can be obtained with the INFO function.

The PARSE command is especially useful when you want to accept expressions as arguments to a program.

Syntax

PARSE *text-expression*

Arguments

text-expression
A text expression that contains one or more smaller expressions to be parsed. When you are processing program arguments, you can specify ARGS as the text expression that contains all the arguments for the program.

Notes

Obtaining Information Produced by the PARSE Command

See INFO for an explanation of how to obtain the information produced by the PARSE command.

Examples

Example 19–27 Parsing the Arguments to a Program

In a simple report program, you want to specify the data to be reported as an argument to the program. You want to be able to specify an expression, as well as the name of a data variable.

Suppose you want to display each of the arguments with a different column width, which means you must process the arguments individually. The ARGS function can only process them together. So you use PARSE and INFO to parse the arguments and produce individual columns for each of them. Here is a sample program.

```
DEFINE report1 PROGRAM
PROGRAM
PUSH month product district DECIMALS
DECIMALS = 0
LIMIT month TO FIRST 2
LIMIT product TO ALL
LIMIT district TO 'Chicago'
PARSE ARGS
REPORT ACROSS month WIDTH 8 <&INFO(PARSE FORMULA 1) -
   WIDTH 13 &INFO(PARSE FORMULA 2)>
POP month product district DECIMALS
END
```

When you run the program, you can supply either the names of variables (such as `sales`) or expressions (such as `sales-expense`) or one of each as arguments. The following REPORT statement produces the illustrated report.

```
report1 sales sales-expense

DISTRICT: CHICAGO
             --------------------MONTH--------------------
             --------JAN95--------- --------FEB95---------
PRODUCT       SALES   SALES-EXPENSE  SALES   SALES-EXPENSE
------------ -------- ------------- -------- -------------
Tents          29,099         1,595   29,010         1,505
Canoes         45,278           292   50,596           477
Racquets       54,270         1,400   58,158         1,863
Sportswear     72,123         7,719   80,072         9,333
Footwear       90,288         8,117   96,539        13,847
```

PARTITIONCHECK

The PARTITIONCHECK function identifies whether an aggmap object is compatible with the partitioning specified by a partition template object.

Aggregation can cross partitions,; however the data flow must always be in one direction. The data cannot go both in and out of the same partition; this processing causes Oracle OLAP to produce an error during the aggregation.

Return Value

`BOOLEAN`.

`YES` when Oracle OLAP would not issue an error when aggregating a variable partitioned using the specified partition template using the specified aggmap; or `NO` when an error would occur.

Syntax

PARITITONCHECK (*aggmap parttition-template*)

Arguments

aggmap
A text expression that is the name of an aggmap object.

partition-template
A text expression that is the name of the partition template object that you want to check for compatibility with aggregation.

Examples

Assume that you have the following objects defined in your analytic workspace.

```
DEFINE YEAR_2003 DIMENSION TEXT
DEFINE YEAR_2002 DIMENSION TEXT
DEFINE PRODUCT DIMENSION TEXT
DEFINE SALES_2003 VARIABLE DECIMAL <YEAR_2003 PRODUCT>
DEFINE SALES_2002 VARIABLE DECIMAL <YEAR_2002 PRODUCT>
DEFINE TIME DIMENSION CONCAT (YEAR_2003 YEAR_2002) UNIQUE
DEFINE TIME_PARENTREL RELATION TIME <TIME>
DEFINE PART_TEMP_SALES_BY_YEAR PARTITION TEMPLATE <TIME PRODUCT> -
   PARTITION BY CONCAT (TIME) -
    (PARTITION PARTITION_2002 <YEAR_2002 PRODUCT> -
     PARTITION PARTITION_2003 <YEAR_2003 PRODUCT>)
DEFINE SALES VARIABLE DECIMAL <PART_TEMP_SALES_BY_YEAR <TIME PRODUCT>> -
    (PARTITION PARTITION_2002 EXTERNAL SALES_2002 -
     PARTITION PARTITION_2003 EXTERNAL SALES_2003)
DEFINE AGG_SALES AGGMAP
  AGGMAP
  RELATION time_parentrel OPERATOR SUM
  END
```

To determine if `sales` is partitioned in such a way that you can use `agg_sales` to aggregate it, issue the following statement. Since the statement returns a value of `YES`, you can safely use `agg_sales` to aggregate `sales`.

```
SHOW PARTITIONCHECK (agg_sales part_temp_sales_by_year)
yes
```

PERCENTAGE

The PERCENTAGE function computes the percent of total for each value in a numeric expression.

Return Value

DECIMAL

Syntax

PERCENTAGE(*expression* [BASEDON *dimension-list*])

Arguments

expression
The numeric expression for which percent figures are to be computed.

BASEDON *dimension-list*
An optional list of one or more of the dimensions of *expression* on which to base the percentage for each value. When you do not specify the dimensions, then PERCENTAGE bases the percentage on the total of all of the values of all of the dimensions of *expression*.

Notes

The Effect of NASKIP

PERCENTAGE is affected by the NASKIP option. When NASKIP is set to `YES` (the default), then PERCENTAGE ignores `NA` values. When NASKIP is set to `NO`, then PERCENTAGE returns `NA` for any cell in *expression* whose value is `NA`.

Examples

Example 19–28 Calculating the Percentage

The following statements s limit the `month` and `district` dimensions, and report the data values, with subtotals, for the `units` variable.

```
LIMIT month TO 'Jul96' TO 'Sep96'
LIMIT district TO 'Denver'
REPORT SUBTOTALS W 8 units
```

The preceding statement produces the following output.

```
DISTRICT: DENVER
               ----------UNITS-----------
               ----------MONTH-----------
PRODUCT         Jul96    Aug96    Sep96
-------------- -------- -------- --------
Tents               608      517      441
Canoes              467      363      411
Racquets          3,006    2,836    2,838
Sportswear        2,395    2,039    2,138
Footwear          1,581    1,532    1,667
-------------- -------- -------- --------
TOTAL DENVER      8,057    7,287    7,495
```

This statement reports the percentage that each `month` value represents of the total `month` values for each of the `product` values that are in status. The total of the values that PERCENTAGE returns for each `product` value is `1`.

```
REPORT SUBTOTALS W 8 DOWN month PERCENTAGE(units BASEDON month)
```

The preceding statement produces the following output.

```
DISTRICT: DENVER
         -----------PERCENTAGE(UNITS BASEDON MONTH)------------
         -----------------------PRODUCT------------------------
MONTH      Tents      Canoes    Racquets  Sportswear  Footwear
-------- ---------- ---------- ---------- ---------- ----------
Jul96          0.39       0.38       0.35       0.36       0.33
Aug96          0.33       0.29       0.33       0.31       0.32
Sep96          0.28       0.33       0.33       0.33       0.35
-------- ---------- ---------- ---------- ---------- ----------
TOTAL          1.00       1.00       1.00       1.00       1.00
DENVER
```

This statement reports the percentage that each `product` value represents of the total `product` values for each of the `month` values that are in status.

```
REPORT SUBTOTALS W 8 PERCENTAGE(units BASEDON product)
```

The preceding statement produces the following output.

```
DISTRICT: DENVER
               -PERCENTAGE(UNITS BASEDON-
               ---------PRODUCT)---------
               ----------MONTH-----------
PRODUCT         Jul96    Aug96    Sep96
-------------- -------- -------- --------
Tents              0.08     0.07     0.06
Canoes             0.06     0.05     0.05
Racquets           0.37     0.39     0.38
Sportswear         0.30     0.28     0.29
Footwear           0.20     0.21     0.22
-------------- -------- -------- --------
TOTAL DENVER       1.00     1.00     1.00
```

This statement reports the percentage based on all of the dimensions of the `units` variable. The total of all of the values that PERCENTAGE returns is `1`.

```
REPORT SUBTOTALS W 8 PERCENTAGE(units)
```

The preceding statement produces the following output.

```
DISTRICT: DENVER
               ----PERCENTAGE(UNITS)-----
               ----------MONTH-----------
PRODUCT         Jul96    Aug96    Sep96
-------------- -------- -------- --------
Tents              0.03     0.02     0.02
Canoes             0.02     0.02     0.02
Racquets           0.13     0.12     0.12
Sportswear         0.10     0.09     0.09
Footwear           0.07     0.07     0.07
-------------- -------- -------- --------
TOTAL DENVER       0.35     0.32     0.33
```

The total for all of the values for both the `product` and `month` dimensions is `1.00`.

PERMIT

The PERMIT command lets you control access to analytic workspace objects. You can use PERMIT commands in Oracle OLAP security applications that specify workspace access rights for many users. You can also use PERMIT as a general scoping tool in other types of applications. Scoping restricts the view of workspace objects.

With the PERMIT command, you can grant or deny read-only and read/write access permission for workspace objects and for specific values of dimensions and dimensioned objects. You can also use PERMIT to grant or deny permission to maintain dimensions and to change permission for workspace objects.

The PERMIT command assigns permission to the object most recently defined or considered. When the definition of the object is not the current one, first use a CONSIDER command before issuing PERMIT commands for the object.

> **Note:** When using PERMIT, it is important that you not lock out the DBA user, which must have access to everything in the workspace at all times.

Syntax

PERMIT {READ|WRITE|MNT|PERMIT} [WHEN *permission-condition*...]

Arguments

READ
Grants permission to read an object or values in a dimension or dimensioned object, depending on the permission conditions. You can specify read permission either with a single-cell permission condition or with dimensioned permission conditions.

When you grant read permission for an object, write permission is also allowed for the values you can read, unless you deny it with an explicit `PERMIT WRITE` statement.

To completely deny access to an object, you can specify `PERMIT READ` with a single-cell permission condition that evaluates to `NO`. To restrict access to a subset of values in a dimension or dimensioned object, you can specify `PERMIT READ` with dimensioned permission conditions. To restore full access to an object, issue a

`PERMIT READ` command with no WHEN clause or with a single-cell permission condition that evaluates to `YES`.

WRITE
Grants permission to modify an object or values of a dimensioned object, depending on the permission conditions. Write permission is not meaningful for dimensions, except to provide write access to objects dimensioned by the dimension. You can specify write permission either with a single-cell permission condition or with dimensioned permission conditions.

When you do not specify a `PERMIT READ` command in addition to the `PERMIT WRITE`, then read permission is provided by default for the object. In this case, when the object is dimensioned and write permission only applies to some of its values, the values with write permission are available for read/write access and the values without write permission are available for read-only access.

MNT
Grants permission to maintain a dimension. Maintain permission always applies to the entire dimension, and is based on a single-cell permission condition. Maintain permission is automatically denied when there is restricted read permission for the dimension, even when you specify maintain permission.

PERMIT
Grants permission to use the PERMIT command to change the read, write, maintain, or permit permission for the object. Permit permission always applies to the entire object, and is based on a single-cell permission condition. Whether or not there is read, write, or maintain permission for an object, permit permission is always allowed unless explicitly denied with a PERMIT PERMIT statement with a permission condition that evaluates to `NO`.

WHEN *permission-condition*...
The conditions for granting read, write, maintain, or permit permission consist of one or more Boolean expressions. When you omit the WHEN clause and execute a `PERMIT READ`, `PERMIT WRITE`, or `PERMIT MNT` statement, Oracle OLAP will restore full read, write, or maintain permission.

When permission applies to an object without dimensionality or to *all* the values of a dimensioned object, or when you are specifying permit or maintain permission, the permission condition consists of a single Boolean value. When you specify a dimensioned Boolean expression in this case, PERMIT uses the first value in status.

When permission applies to individual cells within a dimensioned object, the permission condition consists of a Boolean variable dimensioned by some or all of the dimensions of the object.

When read or write permission applies to dimension values or slices of a dimensioned object, the permission conditions consist of dimensioned Boolean expressions with the following format.

WHEN *dimensioned_permission_condition1*

[BY *dimensioned_permission_condition2*

BY *dimensioned_permission_condition3*...]

Each dimensioned permission condition consists of a Boolean expression dimensioned by one of the dimensions of the object. When a Boolean expression has any extra dimensions in addition to one of the object dimensions, PERMIT takes the first value in status to determine which column of Boolean values to use. The intersection of the `YES` values for each dimension (a logical `AND` of the conditions) is the subset of values within the object to which the permission applies. When any of the object dimensions are not represented by a dimensioned permission condition, then Oracle OLAP assumes `YES` for all those dimension values.

Notes

PERMIT Commands and Objects

You can apply up to four PERMIT commands to an object, one for read, write, maintain, and permit permission. PERMIT commands must exist within the same workspace as the objects for which they control permission.

Resetting Permission

When you want to keep the existing PERMIT commands for an object, but you want Oracle OLAP to recalculate the permission conditions associated with them, issue a PERMITRESET command. The new permission conditions will be evaluated upon next reference to the object. See "Reevaluating Single-Cell Permission Conditions" on page 19-66 and "Permission and the OBJ Function" on page 19-66.

Changing Permission

Provided you have permit permission for an object, you can change its permission by issuing new PERMIT commands for it. The new permission will be evaluated upon next reference to the object. See "Permission and the OBJ Function" on page 19-66.

Reevaluating Single-Cell Permission Conditions

When you are targeting any object but a dimension for permission, and the permission condition consists of a single Boolean variable, any changes to that variable affect the permission immediately. You do not need to execute a PERMITRESET in this case.

Permission and the OBJ Function

In general, Oracle OLAP evaluates permission upon next reference to the object. However, the OBJ function is an exception to this rule. The OBJ function provides information about a workspace object that you specify. Since OBJ does not load the object into memory, it does not reflect any changes to the object permission since the last time it was loaded. When you want OBJ to provide information based on new permission criteria, execute a LOAD command before the OBJ.

Workspace Permission Programs

You can specify values for the variables of permission conditions in the workspace permission programs, PERMIT_READ and PERMIT_WRITE. These programs are user-defined functions which cause the AW ATTACH command to either attach or not attach the workspace, depending on the return value of the program.

When a user attaches the workspace RO (read-only), Oracle OLAP runs PERMIT_READ, if it exists. When a user attaches the workspace RW (read/write), Oracle OLAP runs PERMIT_WRITE, if it exists.

When you specify a password with the AW command, it is passed as an argument to the workspace permission program. The workspace permission programs run before AUTOGO. Permission specified in the workspace permission programs only pertains to objects in the workspace being attached.

Workspace Permission Programs: Evaluating Permission

Within the workspace permission programs, permission is not evaluated upon first reference to an object, as it is in every other context. Permission is only evaluated within a workspace permission program when you issue an explicit PERMIT or PERMITRESET command and then reference the targeted object. AW ATTACH executes a PERMITRESET immediately after executing a workspace permission program. This causes the workspace to be attached with all permission implemented.

Workspace Permission Programs: In More Than One Workspace

When you have workspace permission programs defined in workspaces that are currently attached, Oracle OLAP executes the one in the workspace that you are attaching. However, when you have workspace permission programs in more than one currently attached workspace, you need to take special care when you edit them or use them in any other way, to ensure that you access the appropriate version.

Read/Write Permission

When the only PERMIT command for an object is a `PERMIT WRITE`, then read permission is provided by default for the object. The default read permission is provided independent of the value of the permission condition(s) for the `PERMIT WRITE` statement. This means that a `PERMIT WRITE` with a single-cell permission condition which evaluates to `NO` provides read-only access to an undimensioned object or to all the values of a dimensioned object. When the only PERMIT command for an object is a `PERMIT WRITE` with dimensioned permission conditions, it designates some values for read/write access and the remaining values for read-only access. See Example 19–29, "Variable Permission" on page 19-70.

Write But Not Read

Oracle OLAP does not prevent you from establishing write permission for values that you cannot read within a dimensioned object. When you have both a `PERMIT READ` and a `PERMIT WRITE` statement for a dimensioned object, and some of the values which satisfy the permission conditions for write do not fall within the subset of values which satisfy the permission conditions for read, then those values may be modified but not seen.

Default Status

The dimension values that satisfy the permission condition for `PERMIT READ` constitute the default status for the dimension. When Oracle OLAP loops over the dimension, it only includes those values with read permission. For example, a `LIMIT ALL` statement provides only those values. A reference to integer position means the position within the set of values with read permission. The same principle also applies to QDRs, LAG and LEAD references, and UNRAVEL.

Note: Dimensions with an INTEGER data type have values identified by their numeric position. PERMIT renumbers INTEGER dimensions to keep the normal sequence of integers (1, 2, 3, ...). When this behavior is not desirable, you should use a text or time-period data type.

All dimensioned data is affected by the read permission on its dimensions. The values of dimensioned objects that correspond to dimension values without read permission are inaccessible.

Dimension Permission

Write permission is only meaningful for dimensions in providing write access to objects dimensioned by the dimension. In order for write permission associated with a dimension to apply to objects dimensioned by it, there must be at least one PERMIT command associated with the dimensioned object. When you want a dimensioned object to inherit write permission from its dimensions but you do not want it to have permission of its own, which could interact with the dimension permission, you can simply use a `PERMIT READ` with a single-cell permission condition that evaluates to `YES`. Dimension permission interacts with permission for objects dimensioned by it in the following ways:

- When there is read or write permission associated with a dimension, but no permission restriction associated with an object dimensioned by that dimension, then the permission for the dimensioned object is the same as the dimension permission.
- When there is read permission associated with both the dimension and the dimensioned object, Oracle OLAP determines the values with read permission in the object by taking the intersection of the values with read permission in the dimension and the values with read permission in the object.
- When there is write permission associated with both the dimension and the dimensioned object, Oracle OLAP determines the values with write permission in the object by taking the intersection of the values with read permission in the dimension, the values with write permission in the dimension, and the values with write permission in the object.

Assigning Access Permissions to a Concat Dimension

Use the PERMIT command to grant or deny access to dimension values. Access restrictions that you apply to the concat dimension are added to any restrictions that already exist on the component dimensions.

Relations, Valuesets, and Worksheets

You can specify permission based on a single-cell permission condition for relations, valuesets, and worksheets. When there is restricted write permission for a dimension of a relation, it does not affect the relation. Restricted write permission on the dimension from which a valueset derives does not affect permission on the valueset.

Programs, Models, and Formulas

You can specify read and write permission for programs, models, and formulas with a single-cell permission condition. When you have read/write permission for a program, model, or formula, you can both edit and run it. When you have read-only permission, you can run it but not change it.

Change Permission Authority

You should avoid specifying a `PERMIT PERMIT` statement with a Boolean value as a permission condition (for example, `YES` or `NO`). Instead specify the permission condition as a Boolean variable, a function that returns a Boolean result, or a Boolean value calculated by comparison operators. In this way, when permit permission has been denied, you can restore it by setting the value of the Boolean and executing a PERMITRESET command. When you do lock up an object and are unable to modify its permission, you can specify permit permission for it in the workspace permission program for that workspace, then detach and reattach the workspace. For more information on workspace permission programs, see "Workspace Permission Programs" on page 19-66.

Determining Permission

The permission associated with an object is provided, like an LD, when you describe it using a DESCRIBE statement. The only exception is when you are denied permit permission for the object. In this case, no permission is provided when you describe it.

Scoping

As a tool for scoping within application programs, PERMIT has several advantages over the LIMIT command. To restrict the scope of a dimensioned object according to a Boolean expression, you have to use two LIMIT statements, a `LIMIT` and a `LIMIT KEEP`. You only need one PERMIT command to do the same thing. Moreover, application users cannot change the restricted scope set by PERMIT commands in application programs. Application users can easily change the scope set by LIMIT commands in application programs simply by executing more LIMIT commands.

Permission Violations

You can use the PERMITERROR option to control the way Oracle OLAP handles attempted violations of the permission established by PERMIT commands for variables. The default value of PERMITERROR is `YES`, meaning that Oracle OLAP will signal an error when a user attempts to access a value for which permission is denied. When you set PERMITERROR to `NO`, Oracle OLAP simply denies access without signaling an error condition. This is useful when you want to do a report of a dimensioned variable for which you have partial permission without limiting the dimensions to the permitted values up front. With PERMITERROR set to `NO`, values for which you do not have read permission appear as `NA` values in the report.

Permissions and Concat Dimensions

You can use the PERMIT command to assign permissions to a concat dimension. Any access restrictions on a concat dimension are in addition to the restrictions on its component dimensions. To have access to a value of the concat dimension, you must have permission to access the value in the concat itself and in all the components that contain the value.

Permissions and Dimension Surrogates

You cannot use the PERMIT command on a dimension surrogate. The access permissions of a dimension apply to all dimension surrogates defined for that dimension.

Examples

Example 19–29 Variable Permission

For a variable `sales` dimensioned by `month`, `product`, and `district`, you might have three dimensioned permission conditions in the form of three variables as illustrated in the following report.

```
MONTH.BOOL<MONTH>  PROD.BOOL<PRODUCT>  DISTRICT.BOOL<DISTRICT>
-----------------  ------------------  -----------------------
Jan95    NO        Tents     YES       Boston     NO
Feb95    YES       Canoes    YES       Atlanta    NO
Mar95    NO        Racquets   NO       Chicago    YES
...      ...       ...        ...      ...        ...
```

When the `YES` values shown in the preceding example are the only `YES` values in the permission conditions, the following PERMIT command provides read/write access to `sales` data for tents and canoes sold in Chicago in Feb95. In the absence of a `PERMIT READ` statement for `sales`, Oracle OLAP provides read-only permission for all the other values of `sales`.

```
PERMIT WRITE WHEN district.bool BY prod.bool BY month.bool
```

You can restore full write permission with the following PERMIT command.

```
PERMIT WRITE
```

When there is no restricted write permission for `sales`, the following PERMIT command provides read/write access to sales data for tents and canoes sold in Chicago in Feb95, and it causes all other values of sales to be invisible.

```
PERMIT READ WHEN district.bool BY prod.bool BY month.bool
```

Example 19–30 Dimensioned Permission Condition

To restrict access to the `product` dimension you need a permission condition dimensioned by `product`. However, when the permission condition has a second dimension, say `authority`, PERMIT selects the BOOLEAN values that pertain to `product` based on the first value in status of `authority`. When you restrict read permission on the `authority` dimension to one value, PERMIT uses that value to determine the BOOLEAN values of the permission condition for `product`. The REPORT commands produce the output that follows them.

```
DEFINE authority DIMENSION TEXT
MAINTAIN authority ADD OTHER DBA
DEFINE prod_authority VARIABLE BOOLEAN <product authority>
...
" Assign values to the variable
...
REPORT prod_authority

              ----------------PROD_AUTHORITY-----------------
              -------------------PRODUCT---------------------
AUTHORITY     Tents   Canoes   Racquets   Sportswear   Footwear
---------     -----   ------   --------   ----------   --------
Other            NO       NO        YES          YES        YES
Dba             YES      YES        YES          YES        YES

CONSIDER product
PERMIT READ WHEN prod_authority
PERMITERROR = NO
RPEPORT product

PRODUCT
-------------
Racquets
Sportswear
Footwear

CONSIDER authority
PERMIT READ WHEN AUTHORITY EQ 'dba'
PERMITRESET
Report product

PRODUCT
-------------
Tents
Canoes
Racquets
Sportswear
Footwear
```

Example 19–31 User-Defined Boolean Function

In the following example, `usercheck` is a user-defined Boolean function that checks the current value of the variable `thisuser` against a list of user IDs. `usercheck` returns `NO` when the current value of `thisuser` is not in the list. The following PERMIT command applied to the `sales` variable provides read-only access to all values of `sales` when `usercheck` returns `NO`. It provides read/write access to all values of `sales` when `usercheck` returns `YES`.

```
PERMIT WRITE WHEN usercheck(thisuser)
```

The following PERMIT command, applied to the variable `price`, provides full access to all values of `price` when `usercheck` returns `YES`. When it returns `NO`, it denies all access to the `price` variable.

```
PERMIT READ WHEN usercheck(thisuser)
```

Example 19–32 Individual Cells

When you want to prevent access to one particular sales figure, say for racquets in Boston in March of 1997, you can create a Boolean variable and use it in a PERMIT command as illustrated in the following statements.

```
DEFINE sales.bool VARIABLE BOOLEAN <month product district>
sales.bool = yes
LIMIT month TO 'Mar97'
LIMIT product TO 'Racquets'
LIMIT district TO 'Boston'
sales.bool = no
CONSIDER sales
PERMIT READ WHEN sales.bool
```

Example 19–33 Individual Dimension Values

The following PERMIT commands applied to the `district` dimension prevent access to all dimension values except `Boston` and `Atlanta`. They provide read/write access for all data related to `Boston` and read-only access for all data related to `Atlanta`. They also prevent anyone with a user ID not allowed by the function `usercheck` (see Example 19–31, "User-Defined Boolean Function" on page 19-73) from modifying the permission for `district`.

```
PERMIT READ WHEN district EQ 'Boston' OR district EQ 'Atlanta'
PERMIT WRITE WHEN district EQ 'Boston'
PERMIT PERMIT WHEN usercheck(thisuser)
```

PERMIT_READ

A program that you write and that Oracle OLAP checks for by name when an AW ATTACH read-only statement executes. Depending on the value returned by the program, Oracle OLAP executes the code within the program after attaching the analytic workspace.

Note: Oracle OLAP checks for other programs when a user attaches a workspace. See "Startup Programs" on page 1-11 for more information.

Returns

`BOOLEAN`

`TRUE` when Oracle OLAP has successfully set up and attached the analytic workspace; or `FALSE` when it has not or when the `permit_read` program has thrown an exception

Note: You are encouraged to use the normal return values rather than relying on exceptions to create a return value of `FALSE`.

Syntax

To define a program with the name `PERMIT_READ` use the syntax shown in DEFINE PROGRAM. Code the actual program as a user-defined function with the the following argument.

PERMIT_READ (*password*)

Arguments

See AW ATTACH for an explanation of *password*.

Note: When a user specifies a password when attaching the analytic workspace, then the password is passed as an argument to the program for processing.

Notes

Creating a PERMIT_READ program

A program with the name of `permit_read` does not exist within an analytic workspace unless you define and write one. You write a `permit_read` as a user-defined functions that returns a BOOLEAN value. You can use the return value to indicate to Oracle OLAP whether or not the user has the right to attach the workspace.

Depending on the statements in the `permit_read` program the user is granted or denied access to specific objects or sets of object values. Within `permit_read` program, you can specify PERMIT commands that grant or restrict access to individual workspace objects.

Note: All of the objects referred to in a given `permit_read` must exist in the same analytic workspace.

Examples

For examples of how attachment programs, see Example 8–14, "Startup Programs" on page 8-42.

PERMIT_WRITE

A program that you write and that Oracle OLAP checks for by name when an AW ATTACH read/write statement executes. Depending on the value returned by the program, Oracle OLAP executes the code within the program after attaching the analytic workspace.

Note: Oracle OLAP checks for other programs when a user attaches a workspace. See "Startup Programs" on page 1-11 for more information.

Returns

`BOOLEAN`

`TRUE` when Oracle OLAP has successfully set up and attached the analytic workspace; or `FALSE` when it has not or when the `permit_write` program has thrown an exception

Note: You are encouraged to use the normal return values rather than relying on exceptions to create a return value of `FALSE`.

Syntax

To define a program with the name `PERMIT_WRITE` use the syntax shown in DEFINE PROGRAM. Code the actual program as a user-defined function with the following argument.

PERMIT_WRITE (*password*)

Arguments

See AW ATTACH for an explanation of *password*.

Note: When a user specifies a password when attaching the analytic workspace, then the password is passed as an argument to the program for processing.

Notes

Creating a permit_write program

A program with the name of `permit_write` does not exist within an analytic workspace unless you define and write one. You write a `permit_write` as a user-defined functions that returns a BOOLEAN value. You can use the return value to indicate to Oracle OLAP whether or not the user has the right to attach the workspace.

Depending on the statements in the permit_write program, the user is granted or denied access to specific objects or sets of object values. Within permit_write program, you can specify PERMIT commands that grant or restrict access to individual workspace objects.

Note: All of the objects referred to in a given `permit_write` must exist in the same analytic workspace.

Examples

For examples of how attachment programs, see Example 8–14, "Startup Programs" on page 8-42.

PERMITERROR

The PERMITERROR option controls whether or not an error is signaled on attempted access of a variable for which read or write permission is denied by a PERMIT command.

See Also: PERMIT and PERMITRESET.

Data type

BOOLEAN

Syntax

PERMITERROR = NO|YES

Arguments

NO

When you set PERMITERROR to `NO`, an error condition is not created on attempted access of a variable for which read or write permission is denied with a PERMIT command. Values for which you do not have read permission are displayed as `NA`'s. When you try to change a value for which you do not have write permission, the request is ignored.

YES

When PERMITERROR is `YES` (the default), an error is signaled upon attempted access of a variable for which read or write permission is denied with a PERMIT command. The error, which can be trapped, terminates the Oracle OLAP operation that initiated the illegal access.

Notes

Programs, Models, and Valuesets

The setting of PERMITERROR is ignored for violations of permission for objects without dimensionality such as programs, models, and valuesets. Attempted access of variables and relations with permission, whether or not they have dimensionality, is always affected by the setting of PERMITERROR.

Maintaining Dimensions

The setting of PERMITERROR is ignored for violations of maintain and permit permission. Attempted violations of permission to maintain dimensions and to change permission are always treated as errors. Attempted violations of read or write permission for dimensions are, similarly, always treated as errors.

Obtaining Data Without Full Permission

When PERMITERROR is `YES` and you attempt to fetch a dimensioned variable that contains values that do not have read permission, an error condition is created when the first of those values is encountered. You can avoid creating an error condition by limiting the dimensions in advance so that only permissible values are in status, or by setting PERMITERROR to `NO`, before doing the report.

Examples

Example 19–34 Report Without Full Permission

In the following example, the read permission on the `price` variable prevents you from seeing price data for any values of `product` other than `Tents`. However, when you set PERMITERROR to `NO`, you can still do a report of the `price` variable for Dec. 1996 without creating an error condition.

```
PERMITERROR = no
DESCRIBE price
```

The output of this statement is

```
DEFINE PRICE VARIABLE DECIMAL <MONTH PRODUCT>
LD Wholesale Unit Selling Price
PERMIT READ WHEN product eq 'Tents'
```

The statements

```
LIMIT month TO 'Dec96'
REPORT price
```

produce the following output.

```
                 ----PRICE----
                 ----MONTH----
PRODUCT              DEC96
----------------  -------------
Tents                 165.64
Canoes                    NA
Racquets                  NA
Sportswear                NA
Footwear                  NA
```

The statements

```
PERMITERROR = yes
REPORT price
```

produce the following error,

```
ERROR: You do not have permission to read this value of PRICE
```

and the following output.

```
                ---PRICE---
                ---MONTH---
PRODUCT            DEC96
---------------  -----------
Tents                165.64
```

PERMITRESET

The PERMITRESET command causes the values of permission conditions to be reevaluated. Permission conditions consist of one or more Boolean expressions that designate the criteria used by PERMIT commands associated with an object.

When permission conditions are dimensioned, they indicate which values of a dimensioned object PERMIT will target for permission. A single-cell permission condition can indicate any Boolean criterion, such as whether or not a particular user may access the object.

When you want to keep the existing PERMIT commands for an object, but you want the permission conditions associated with them to be recalculated, issue a PERMITRESET command. The permission for that object will be based on the new values of the permission conditions the next time you use the object in an OLAP DML statement.

See also: PERMIT and PERMITERROR

Syntax

PERMITRESET [*object_name*] [READ|WRITE]

Arguments

object_name
Specifies the name of an object for which permission conditions should be reevaluated. When you do not specify an object name, the permission conditions for all objects are reevaluated.

READ
Causes reevaluation of the permission conditions for `PERMIT READ` commands only, or for the `PERMIT READ` command for the specified object.

WRITE
Causes reevaluation of the permission conditions for `PERMIT WRITE` commands only, or for the `PERMIT WRITE` command for the specified object.

Notes

Changing Permission

Provided you have permit permission for an object, you can change its permission by issuing new PERMIT commands for it. The new permission will be evaluated upon next reference to the object. See "Permission and the OBJ Function" on page 19-82.

Reevaluating Single-Cell Permission Conditions

When you are targeting any object but a dimension for permission, and the permission condition consists of a single Boolean variable, any changes to that variable affect the permission immediately. You do not need to execute a PERMITRESET in this case.

Permission and the OBJ Function

In general, permission are evaluated upon next reference to the object. However, the OBJ function is an exception to this rule. The OBJ function provides information about an analytic workspace object that you specify. Because OBJ does not load the object into memory, it does not reflect any changes to the object's permission since the last time it was loaded. When you want OBJ to provide information based on new permission criteria, execute a LOAD command before the OBJ.

Examples

Example 19–35 Resetting Permission

In the following example, the user-defined Boolean function `usercheck` checks the current value of the variable `thisuser` and returns `YES` only when it is greater than `100`. Access to the variable `uservar` is only allowed when `thisuser` is greater than `100`. However, when you change the value of `thisuser` to a value less than or equal to `100` without resetting the permission for `uservar`, access is still permitted.

The statement

```
DESCRIBE uservar
```

produces the following output.

```
DEFINE USERVAR VARIABLE INTEGER
PERMIT READ WHEN usercheck(thisuser)
```

The statement

```
SHOW uservar
```

produces the following output.

```
5
```

The statement

```
DESCRIBE usercheck
```

produces the following output.

```
DEFINE USERCHECK PROGRAM BOOLEAN
PROGRAM
  ARG thisuser INT
  TRAP ON errorexit NOPRINT
  IF thisuser GT 100
          THEN RETURN YES
  ELSE RETURN NO
  errorexit:
       RETURN NO
END
```

The statement

```
DESCRIBE thisuser
```

produces the following output.

```
DEFINE THISUSER VARIABLE INTEGER
```

The statement

```
SHOW thisuser
```

produces the following output.

```
101
```

The statements

```
thisuser = 100
SHOW uservar
```

produces the following output.

```
5
```

The statements

```
PERMITRESET luservar READ
SHOW uservar
```

produce the following error.

```
ERROR: You do not have permission to read this value of USERVAR
```

POP

The POP command restores the status of a dimension, the status of a valueset, or the value of an option or single-cell variable that was saved with a previous PUSH command.

PUSH and POP are commonly used within a program to make changes to options and dimension status that apply *only* during the program's execution. Afterward, the options and status are the same as they were before the execution of the program.

Syntax

POP *name1* [*nameN*]

Arguments

name
The name of a dimension, valueset, variable, or option that was specified in a previous PUSH command, whose saved value you want to restore.

Notes

NAME Dimension

PUSH and POP also work for the NAME dimension.

Effect of the MAINTAIN Command

Using the MAINTAIN command with a dimension clears that dimension's pushed status lists. For example, suppose you have pushed the dimension `month` several times, with different limits each time. When you then use the MAINTAIN command to perform any maintenance activity on the `month` dimension, Oracle OLAP resets the status of `month` to `ALL` (the default), and popping that dimension will have no effect.

Related Statements

POPLEVEL, PUSH, PUSHLEVEL, and CONTEXT command.

Examples

For an example of using POP, see Example 19–44, "Saving and Restoring Values" on page 19-101.

POPLEVEL

The POPLEVEL command restores all values saved with PUSH commands that were executed since the last POPLEVEL command specifying the same marker.

You must use PUSHLEVEL to mark a starting point for a series of PUSH commands before you can use POPLEVEL to restore the saved values. POPLEVEL itself marks the end of the series. You can use POPLEVEL only within programs. (Abbreviated PPL.)

Syntax

POPLEVEL *marker-expression* [DISCARD]

Arguments

marker-expression
A text value used as a marker. This must be exactly the same as the value used in the corresponding PUSHLEVEL command to mark the start of a series of saved values being popped.

DISCARD
Specifies that the pushed values for that level are discarded when you issue the POPLEVEL command. When you do not specify DISCARD, the values that were pushed are used to reset the pushed objects.

Notes

POPLEVEL and PUSH

Two possible uses for the POPLEVEL command are:

- After a series of increasingly broadening or narrowing LIMIT commands, each with a corresponding PUSH.
- After a single extremely long and complicated PUSH command, or a series of short ones given throughout a program, that may need a lot of editing. PUSHLEVEL and POPLEVEL allow you to edit the arguments for a long and complicated PUSH command without also having to edit a corresponding long and complicated POP command.

Nesting

You can nest PUSHLEVEL and POPLEVEL pairs, as long as you specify a different marker for each pair.

Duplicate Markers

When you specify the same marker for two or more PUSHLEVEL commands, a POPLEVEL command specifying that same marker will restore values that were saved only since the most recent PUSHLEVEL command.

Multiple PUSHLEVEL Commands

When you specify a different marker for two or more PUSHLEVEL commands, a POPLEVEL command that specifies the marker of any PUSHLEVEL command restores all the values that were saved since that command, including values that were saved after later PUSHLEVEL commands.

Related Statements

POP, PUSH, PUSHLEVEL, and CONTEXT command.

Examples

To see a sample program using POPLEVEL, see the PUSHLEVEL example.

POUTFILEUNIT

The POUTFILEUNIT option identifies a destination for status information about the execution of many OLAP DML statements, including:

- AGGREGATE command
- AGGREGATE function
- ALLOCATE
- CLEAR
- AW ATTACH
- AW DETACH
- UPDATE
- IMPORT
- EXPORT

When an OLAP DML statement is executed through SQL, using the POUTFILEUNIT option enables you to see the work that the statement is doing as it progresses instead of waiting until the execution of the SQL call is complete.

Note: Because POUTFILEUNIT does not have a default setting, this information is not collected unless you set POUTFILEUNIT in your session.

Syntax

POUTFILEUNIT = *fileunit*

Arguments

fileunit

Specifies a destination, such as an open disk file, to which Oracle OLAP sends information on the progress of an operation. The *fileunit* can be the value of the OUTFILEUNIT option or the results of the FILEOPEN function.

Notes

Information Sent by ALLOCATE and AGGREGATE

ALLOCATE, the AGGREGATE command, and AGGREGATE function send the following types of information:

- Progress of the verification of the hierarchy of a dimension
- Progress of the building of intermediate computation structures
- The execution of the allocation or aggregation
- Errors or anomalous behavior that Oracle OLAP encounters in allocating or aggregating the source data to the target variable cells, such as skipped deadlocks in an allocation.

Examples

Example 19–36 Using FILEOPEN to Open a File

The FILEOPEN command opens a file named `progress.txt` in the `userfiles` directory object and returns the file handle to the POUTFILEUNIT option. The file receives status messages from the AGGREGATE command. When the aggregation is complete, the FILECLOSE command closes the file.

```
POUTFILEUNIT=FILEOPEN('userfiles/progress.txt' WRITE)
AGGREGATE sales units USING gpct.aggmap
FILECLOSE POUTFILEUNIT
```

Example 19–37 Viewing Progress in OLAP Worksheet

The following statement sets POUTFILEUNIT to the current outfile destination. When the current outfile destination is the OLAP Worksheet window, remember that you cannot do other work in your session until the operation completes.

```
POUTFILEUNIT=OUTFILEUNIT
```

PRGTRACE

The PRGTRACE option controls whether each line of a program is recorded in the current outfile or in a debugging file during execution of the program. PRGTRACE is primarily used as a debugging tool to uncover problems by tracing the execution of a program.

System DML programs are not traced unless EXPTRACE is set to `YES`.

When you have used the DBGOUTFILE command to specify a debugging file, Oracle OLAP sends PRGTRACE output to the debugging file instead of the current outfile.

Data type

BOOLEAN

Syntax

PRGTRACE = {YES|NO}

Arguments

YES
Oracle OLAP records each line in a program before it is executed.

NO
Oracle OLAP does not record each line in a program. (Default)

Notes

PRGTRACE Output
PRGTRACE records the name of the current program at the beginning of each program line. It includes an equals sign to indicate a compiled line.

```
(PRG= SALESREP) . . .
```

It includes a colon to indicate an uncompiled line.

```
(PRG: SALESREP) . . .
```

A compiled line is a line that has been translated into an efficient internal form, whereas an uncompiled line has not. Oracle OLAP ordinarily stores lines in compiled form to make programs work more efficiently, especially programs that contain loops.

Uncompiled Program Lines

Oracle OLAP compiles a program before running it. Therefore, the only lines that will be marked as uncompiled in the PRGTRACE output are lines that cannot be compiled, such as lines that include ampersand substitution.

Examples

Example 19–38 Tracing Program Execution

Suppose you have a program called `salesrep` that produces a simple budget report.

```
DEFINE salesrep PROGRAM
PROGRAM
PUSH month division line
TRAP ON cleanup
LIMIT month TO &ARGS
LIMIT division TO ALL
LIMIT line TO FIRST 1
REPORT DOWN division across month: dec 0 budget

cleanup:
POP month division line
END
```

When you want to debug this program, you can trace the execution of each of its lines by turning on PRGTRACE and executing the program.

```
PRGTRACE = yes
salesrep FIRST 3
```

PRGTRACE produces the following output in the current outfile or debugging file.

```
(PRG= SALESREP) push month division line
(PRG= SALESREP) trap on cleanup
(PRG: SALESREP) limit month to &args
(PRG= SALESREP) limit division to all
(PRG= SALESREP) limit line to first 1
(PRG= SALESREP) report down division across month: dec0 budget
LINE: REVENUE
               -------------BUDGET-------------
               -------------MONTH--------------
DIVISION         JAN95      FEB95      MAR95
-------------- ---------- ---------- ----------
CAMPING          679,149    707,945    780,994
SPORTING         482,771    517,387    525,368
CLOTHING         983,888  1,016,528    992,331
(PRG= SALESREP) cleanup:
(PRG= SALESREP) pop month division line
```

PROGRAM

The PROGRAM command enters completely new contents into a new or existing program. When the program already has lines of code, Oracle OLAP overwrites them.

PROGRAM assigns contents to the most recently defined or considered program object (see the DEFINE PROGRAM and CONSIDER commands).

> **See also:** For a discussion of writing, compiling, and debugging OLAP DML programs, see Chapter 5, "OLAP DML Programs".

Syntax

PROGRAM [*contents*]

Arguments

contents

A text expression that is the OLAP DML statements that are the lines of the program. You can use most OLAP DML statements within a program. For brief descriptions of statements that manage program flow-of-control and other traditional programming tasks, see Table A–23, " Statements Used Only in OLAP DML Programs" on page A-23 and Table A–24, "Statements Used Primarily in OLAP DML Programs" on page A-25. For a discussion of writing, compiling, and debugging OLAP DML programs, see Chapter 5, "OLAP DML Programs".

The maximum number of lines you can have in a program is 4,000. Separate program lines with newline delimiters (`\n`), or use the JOINLINES function as shown in "Program On the Fly" on page 19-96.

Examples

Example 19–39 User-Defined Function with Arguments

Suppose your analytic workspace contains a variable called `units.plan`, which is dimensioned by the `product`, `district`, and `month` dimensions. The variable holds INTEGER data that indicates the number of product units that are expected to be sold.

Suppose also that you define a program named`units_goals_met`. This program is a user-defined function. It accepts three dimension-value arguments that specify a given cell of the `units.plan` variable, and it accepts a fourth argument that specifies the number of units that were actually sold for that cell. The program returns a Boolean value to the calling program. It returns `YES` when the actual figure comes up to within 10 percent of the planned figure; it returns `NO` when the actual figure does not.

The definition of the `units_goals_met` program is follows.

```
DEFINE units_goal_met PROGRAM BOOLEAN
LD Tests whether actual units met the planned estimate
"Program Initialization
ARGUMENT userprod  TEXT
ARGUMENT userdist  TEXT
ARGUMENT usermonth TEXT
ARGUMENT userunits INTEGER
VARIABLE answer boolean
TRAP ON errorlabel
PUSH product district month
"Program Body
LIMIT product TO userprod
LIMIT district TO userdist
LIMIT month TO usermonth
IF (units.plan - userunits) / units.plan GT .10
   THEN answer = NO
   ELSE answer = YES
"Normal Exit
POP product district month
RETURN answer
"Abnormal Exit
errorlabel:
POP product district month
SIGNAL ERRORNAME ERRORTEXT
END
```

To execute the `units_goal_met` program and store the return value in a variable called `success`, you can use an assignment statement (SET).

```
success = units_goal_met('TENTS' 'BOSTON' 'JUN96' 2000)
```

Example 19–40 Program On the Fly

This example creates a flexible report program "on the fly" to avoid the inefficiencies of a more conventional program using *ampersand substitution.* The conventional program would contain the following loop.

```
FOR &dimname
   ROW &dimname &varname
```

To avoid ampersand substitution, define a program, for example, STANDARDREP, and leave it without any code in it, or with code that can be discarded. Then in your report program, insert lines such as the following.

```
DEFINE myreport PROGRAM
LD Program to produce my report
PROGRAM
ARGUMENT dimname TEXT
ARGUMENT varname TEXT
...
CONSIDER standardrep
PROGRAM JOINLINES(JOINCHARS('FOR ', dimname) -
   JOINCHARS('   ROW ', dimname, ' ', varname) )
COMPILE standardrep
standardrep
...
```

Example 19–41 Program from an Input File

This example presents the text of a simple program that is in an ASCII disk file called `salesrep.inf`. The first line in the file defines the program, the second line contains the PROGRAM command, and the subsequent lines provide the lines of the program.

```
DEFINE salesrep PROGRAM
PROGRAM
PUSH month product district
TRAP ON haderror
LIMIT month TO FIRST 3
LIMIT product TO FIRST 3
LIMIT district TO ALL
REPORT grandtotals sales
haderror:
POP month product district
END
```

To include the `salesrep` program in your analytic workspace, you can execute the following statement.

```
INFILE 'salesrep.inf'
```

You can create an input file from an existing program using the OUTFILE command

Example 19–42 Using OLAP Worksheet Instead of the PROGRAM Command

When you use OLAP Worksheet to create a new program, you can use the EDIT command to display an Edit window where you can enter the contents. For example, use the following statements to define a new program named `salesrep` and display it in an Edit window.

```
DEFINE salesrep PROGRAM
EDIT salesrep
```

PROPERTY

The PROPERTY command adds or deletes properties to the most recently defined or considered object (see the DEFINE PROGRAM and CONSIDER commands). A property is a named value that is associated with a given definition. You can assign one or more properties to any type of definition. For example, you can assign a property to an object so you know how many decimal places to use when preparing a report on the object.

Syntax

PROPERTY { *name value* | DELETE {ALL | *name*} }

Arguments

name
A text expression that contains the name of the property. The property name can be from 1 to 256 bytes long.

Important: Do not create your own properties with names that begin with a `$` (dollar sign). Properties with names beginning with a `$` (dollar sign) are reserved for Oracle OLAP to use as "system" properties that Oracle OLAP interprets in predetermined ways.

Property names have the TEXT data type, unless you include a Unicode escape sequence in the value you specify for the name, or unless you explicitly convert the value you specify to NTEXT (using the CONVERT or TO_NCHAR functions).

value
An expression that contains the value of the property. The property value can have one of the following data types: NUMBER, INTEGER, LONGINTEGER, DECIMAL, SHORTDECIMAL, TEXT, NTEXT, ID, BOOLEAN, DATE, or DATETIME. Oracle OLAP determines the data type based on the value that you specify. For example, when you specify `YES`, then Oracle OLAP assumes a type of BOOLEAN. When you specify a date value that is stored in a variable of type DATE, then Oracle OLAP assumes a type of DATE for the property.

DELETE ALL
DELETE *name*
Deletes either all of the properties of the object or only the property you specify for *name*. You can specify only one *name* at a time.

Notes

Triggering Program Execution When PROPERTY Executes

Using the TRIGGER command, you can make the PROPERTY command an event that automatically executes an OLAP DML program. See "Trigger Programs" on page 1-14 for more information

Changing a Property Value

When you execute a PROPERTY command that assigns a new value to an existing property name, then the new value overwrites the previous one.

Determining Property Values with OBJ

To use properties with OLAP DML statements, you must obtain the values by using the property-related keywords of the OBJ function. For example, suppose a property called `decplace` stores the number of decimal places to use when reporting an object. When you execute the REPORT command, you can use the OBJ function with the PROPERTY keyword to obtain the value of the `decplace` property and use that value with the REPORT command's DECIMAL attribute.

Listing Property Values with FULLDSC

You can list the properties of an object by using the FULLDSC command. You can use the output from FULLDSC to create new objects. See FULLDSC for more information.

Examples

Example 19–43 Adding Properties to a Variable

The following statements add the properties `decplace` and `prgname` to the `actual` variable and assign the `decimal 4` as the value for the `decplace` property and the text `repprg` as the value for the `prgname` property.

```
CONSIDER actual
PROPERTY 'decplace' 4
PROPERTY 'prgname' 'repprg'
```

PUSH

The PUSH command saves the current status of a dimension, the status of a valueset, or the value of an option or single-cell variable. You can then restore these saved values and status at a later time with a POP command.

PUSH and POP are commonly used within a program to make changes to options and dimension status that apply *only* during the program's execution. Afterward, the options and status are the same as they were before the execution of the program.

See also: the following related commands:

- POPLEVEL and PUSHLEVEL
- CONTEXT command

Syntax

PUSH *name1* [*name*]

Arguments

name

The name of a dimension, valueset, option, or variable whose status or value you want to save.

Notes

NAME Dimension

PUSH and POP also work for the NAME dimension.

Effect of the MAINTAIN Command

Using the MAINTAIN command with a dimension clears that dimension's pushed status lists. For example, suppose you have pushed the dimension `month` several times, with different limits each time. When you then use the MAINTAIN command to perform any maintenance activity on the `month` dimension, Oracle OLAP resets the status of `month` to `ALL` (the default), and popping that dimension will have no effect.

Examples

Example 19–44 Saving and Restoring Values

The following program uses PUSH and POP to produce sales figures without decimal places for a specific selection of products, districts, and months, and then restores the status settings and the value of DECIMALS to what they were before the program was run.

```
DEFINE report1 PROGRAM
PROGRAM
TRAP ON cleanup
PUSH DECIMALS product district month

DECIMALS = 0
LIMIT product TO 'Sportswear' 'Footwear'
LIMIT district TO 'Atlanta' 'Dallas'
LIMIT month TO 'Jan96' TO 'Jun96'
REPORT sales

cleanup:
POP DECIMALS product district month
END
```

PUSHLEVEL

The PUSHLEVEL command marks the start of a series of PUSH commands. You can then use a corresponding POPLEVEL command to restore all the values saved by PUSHcommands that are executed after PUSHLEVEL. POPLEVEL must specify the same marker as the PUSHLEVEL command that starts the series. You can use PUSHLEVEL only within programs.

Syntax

PUSHLEVEL *marker-expression*

Arguments

marker-expression
A text value to mark the start of a series of PUSH commands all of whose saved values are to be popped at once. A POPLEVEL command that specifies the exact same *marker-expression* restores the whole series of saved values.

Notes

Nesting

You can nest PUSHLEVEL/POPLEVEL pairs, as long as you specify a different marker for each pair, as illustrated in the following code.

```
PUSHLEVEL 'firstlevel'
PUSH PAGESIZE DECIMALS       < saves values in FIRSTLEVEL
...
PUSHLEVEL 'secondlevel'
PUSH month product           < Saves values in SECONDLEVEL
...
POPLEVEL 'secondlevel'       < Restores values in SECONDLEVEL
...
POPLEVEL 'firstlevel'        < Restores values in FIRSTLEVEL
```

You do not normally need more than one level in a single program. However, Oracle OLAP automatically creates nested levels when one program calls another program and each program contains a set of PUSHLEVEL and POPLEVEL commands.

Duplicate Markers

When you specify the same marker for two or more PUSHLEVEL commands, a POPLEVEL command specifying that same marker will restore values that were saved only since the most recent PUSHLEVEL command.

Multiple PUSHLEVEL Commands

When you specify a different marker for two or more PUSHLEVEL commands, a POPLEVEL command that specifies the marker of any PUSHLEVEL command restores all the values that were saved since that command, including values that were saved after later PUSHLEVEL commands.

Related Statements

POP PUSH, POPLEVEL, and CONVERT.

Examples

Example 19–45 Creating Level Markers

You can use the PUSHLEVEL command to establish a level marker called `firstlevel`, and then use PUSH to save the current values.

```
PUSHLEVEL 'firstlevel'
PUSH month DECIMALS ZSPELL
```

The level marker can be any text that is enclosed in single quotation marks. It can also be the name of a single-cell `ID` or `TEXT` variable, whose value becomes the name of the level marker. In the exit sections of the program, you can then use the POPLEVEL command to restore all the values you saved since establishing the `firstlevel` marker.

```
POPLEVEL 'firstlevel'
```

Example 19–46 Nesting PUSHLEVEL and POPLEVEL Commands

You can nest PUSHLEVEL and POPLEVELcommands to save certain groups of values in one place in a program and other groups of values in another place in a program. The next example shows two sets of nested PUSHLEVEL and POPLEVEL commands.

```
PUSHLEVEL 'firstlevel'
PUSH PAGESIZE DECIMALS "Saves values in FIRSTLEVEL
        ...
PUSHLEVEL 'secondlevel'
PUSH month product     "Saves values in SECONDLEVEL
      ...
POPLEVEL 'secondlevel' "Restores values in SECONDLEVEL
       ...
POPLEVEL 'firstlevel'  "Restores values in FIRSTLEVEL
```

Normally, you do not use more than one set of PUSHLEVEL and POPLEVEL commands in a single program. However, the nesting feature comes into play automatically when one program calls another program, and each program contains a set of PUSHLEVEL and POPLEVEL commands.

Example 19–47 One-Step Restoration and Nested Levels

The following program uses `PUSHLEVEL 'rpt1'` to mark for one-step restoration the original value of DECIMALS and the original status of `month`, `product`, and `district`, even though these are pushed separately in the program.

To demonstrate nesting, the program includes a nested PUSHLEVEL-POPLEVEL pair with `'rpt2'` as its marker and some STATUS commands at various points.

You can compare the program's output with the program to see how the status is affected.

```
DEFINE sales.RPT PROGRAM
PROGRAM
STATUS month product district

PUSHLEVEL 'rpt1'
PUSH DECIMALS month
DECIMALS = 0
LIMIT month TO 'Jan96'
REPORT WIDTH 8 DOWN district WIDTH 9 ACROSS product: expense
PUSH product
LIMIT product TO 'Racquets' 'Sportswear'
REPORT DOWN district ACROSS product: advertising

PUSHLEVEL 'rpt2'
PUSH district
LIMIT district TO 'Atlanta' 'Dallas' 'Chicago'
REPORT DOWN district ACROSS product: sales
BLANK
STATUS month product district
BLANK

POPLEVEL 'rpt2'
STATUS month product district
BLANK
POPLEVEL 'rpt1'

STATUS month product district
END
```

The `sales.rpt` program produces the following output.

```
The current status of MONTH is:
ALL
The current status of PRODUCT is:
ALL
The current status of DISTRICT is:
ALL
MONTH: JAN96
         ---------------------EXPENSE---------------------
         ---------------------PRODUCT---------------------
DISTRICT   Tents    Canoes   Racquets  Sportswear  Footwear
-------- --------- --------- --------- ---------- ----------
Boston      31,299    67,527    52,942     49,668    80,565
Atlanta     41,139    53,186    57,159    108,047    99,758
Chicago     27,768    45,621    53,756     65,055    81,639
Dallas      47,063    34,072   118,807    113,629    19,785
Denver      33,177    42,975    89,144     63,380    36,960
Seattle     41,043    64,009    26,719     38,970    46,900
Month: Jan96
               -----ADVERTISING-----
               -------PRODUCT-------
DISTRICT        RAcquets  Sportswear
-------------- ---------- ----------
Boston             3,784      3,352
Atlanta            4,384      9,509
Chicago            3,351      5,283
Dallas             8,700      8,340
Denver             6,215      4,654
Seattle            2,344      3,726
MONTH: Jan96
               --------SALES--------
               -------PRODUCT-------
DISTRICT        Racquets  Sportswear
-------------- ---------- ----------
Atlanta           61,895    129,616
Dallas           125,880    128,115
Chicago           58,649     77,490
The current status of MONTH is:
JAN96
The current status of PRODUCT is:
RACQUETS, SPORTSWEAR
The current status of DISTRICT is:
ATLANTA, DALLAS, CHICAGO
```

```
The current status of MONTH is:
JAN96
The current status of PRODUCT is:
RACQUETS, SPORTSWEAR
The current status of DISTRICT is:
ALL

The current status of MONTH is:
ALL
The current status of PRODUCT is:
ALL
The current status of DISTRICT is:
ALL
```

QUAL

The QUAL function lets you explicitly specify a qualified data reference (QDR). You should use QUAL in cases where the syntax of a QDR is ambiguous and could either be misinterpreted by Oracle OLAP or cause a syntax error.

QDRs provide a mechanism for limiting one or more dimensions of an expression to a single value. QDRs are useful when you want to temporarily reference a value that is not in the current status.

Return Value

The value that is returned has the same data type as the expression being qualified.

Syntax

QUAL(*expression*, *dimname1 dimexp1* [, *dimnameN dimexpN*])

Arguments

expression
The expression being qualified. You should use QUAL to qualify complex expressions that contain computation, function calls, or ampersand substitution. You can also use QUAL when the expression is a simple variable name. However, QUAL is not required for simple expressions, and you can use the following standard QDR syntax.

expression(*dimname1 dimexp1* [, *dimname2 dimexp2* ...])

dimname
The dimension to be limited. You can specify one or more of the dimensions of the expression. Each dimension must be paired with a *dimexp*. You can specify a dimension surrogate instead of the dimension.

dimexp
An expression that represents the value to which the dimension should be limited. The expression can be a value of the dimension, a text expression whose result is a value of the dimension, a numeric expression whose result is the logical position of a value of the dimension, or a relation of the dimension.

When the dimension being limited is a conjoint dimension, then *dimexp* must be enclosed in angle brackets and must include a value for each of its base dimensions.

When the dimension being limited is a concat dimension, then *dimname* and *dimexp* can be one of the combinations listed in Table 19–1, "Valid dimname and dimexp Combinations for Concat Dimensions".

Table 19–1 Valid dimname and dimexp Combinations for Concat Dimensions

dimname	***dimexp***
The name of the concat dimension	A value of the concat dimension
The name of the concat dimension	The name of a base dimension
The name of a base dimension of the concat dimension	A value of the base dimension
The name of a base dimension of the concat dimension	The name of the concat dimension

Examples

Example 19–48 Using QUAL with MAX

The following example first shows how you might view your data by limiting its dimensions, and then how you might view it by using QUAL.

Assume that you issue the following OLAP DML statements to limit the view of the Cogs line data in the Sporting division to January 1996 through June 1996, and, then, report by month on the maximum value of actual costs or budgeted costs or `MAX(actual,budget)`, actual costs, and budgeted costs for each month.

```
LIMIT month TO 'Jan96' TO 'Jun96'
LIMIT line TO 'Cogs'
LIMIT division TO 'Sporting'
REPORT DOWN month W 11 MAX(actual,budget) W 11 actual W 11 budget
```

The preceding statements produce the following report.

```
DIVISION: SPORTING
               ---------------LINE----------------
               ---------------COGS----------------
               MAX(ACTUAL,
MONTH            BUDGET)     ACTUAL      BUDGET
-------------- ----------- ----------- -----------
Jan96           287,557.87  287,557.87  279,773.01
Feb96           323,981.56  315,298.82  323,981.56
Mar96           326,184.87  326,184.87  302,177.88
Apr96           394,544.27  394,544.27  386,100.82
May96           449,862.25  449,862.25  433,997.89
Jun96           457,347.55  457,347.55  448,042.45
```

Now consider how you might view the same figures for `MAX(actual,budget)` without changing the status of `line` or `division`.

```
ALLSTAT
LIMIT month TO 'Jan96' TO 'Jun96'
REPORT HEADING 'For Cogs in Sporting Division' DOWN month -
   W 11 HEADING 'MAX(actual,budget)'-
   QUAL(MAX(actual,budget), line 'Cogs', division 'Sporting')

For Cogs in
Sporting       MAX(actual,
Division         budget)
-------------- -----------
Jan96           287,557.87
Feb96           323,981.56
Mar96           326,184.87
Apr96           394,544.27
May96           449,862.25
Jun96           457,347.55
```

When you attempt to produce the same report with standard QDR syntax, Oracle OLAP signals an error.

```
REPORT HEADING 'For Cogs in Sporting Division' DOWN month -
   W 11 HEADING 'MAX(actual,budget)'-
   MAX(actual,budget) (line cogs, division sporting)
```

The following error message is produced.

```
ERROR: A right parenthesis or an operator is expected after LINE.
```

Example 19–49 Using QUAL with a Concat Dimension

The following example shows two ways of limiting the values of a concat dimension in a QUAL function. The `reg.dist.ccdim` concat dimension has `region` and `district` as its base dimensions. The `rdsales` variable is dimensioned by `month`, `product`, and `reg.dist.ccdim`.

```
LIMIT month TO 'Jan96' TO 'Jun96'
LIMIT product TO 'Tents' 'Canoes'

" Limit the concat by specifying one of its component dimensions
REPORT W 30 QUAL(rdsales * 2, month 'Feb96', district 'Boston')
```

These statements produce the following report.

```
                 QUAL(RDSALES * 2, MONTH
PRODUCT          'Feb96', DISTRICT 'Boston')
-------------- ------------------------------
Tents                             69,283.18
Canoes                           164,475.36

" Limit the concat by specifying one of its values
REPORT W 30 QUAL(rdsales * 2, month 'Mar96', reg.dist.ccdim
'<district: Boston>')

                  QUAL(RDSALES * 2, MONTH
                  'Mar96', REG.DIST.CCDIM
PRODUCT            '<district: Boston>')
-------------- ------------------------------
TENTS                             91,484.42
CANOES                           195,244.56
```

20

RANDOM to REPORT

This chapter contains the following OLAP DML statements:

- RANDOM
- RANDOM.SEED.1 and RANDOM.SEED.2
- RANK
- RECAP
- RECNO
- RECURSIVE
- REDO
- REEDIT
- REGRESS
- REGRESS.REPORT
- RELEASE
- REM
- REMBYTES
- REMCHARS
- REMCOLS
- REMLINES
- RENAME
- REPLBYTES
- REPLCHARS

- REPLCOLS
- REPLLINES
- REPORT

RANDOM

The RANDOM function produces a number that is randomly distributed between specified low and high boundaries. Randomly generated numbers are useful when building and duplicating tests of applications. They are especially useful for simulation and forecasting applications.

Return Value

DECIMAL

Syntax

RANDOM([*lowbound*] [*highbound*])

Arguments

lowbound
A numeric expression that specifies the lower boundary for the random number series. The default is `0`.

highbound
A numeric expression that specifies the upper boundary for the random number series. The default is `1`.

Notes

NA Values

When either *lowbound* or *highbound* is `NA`, the RANDOM function produces `NA`.

DECIMAL-to-INTEGER Conversion

When you use the RANDOM function to assign values to an INTEGER variable, RANDOM produces decimal values that are rounded when assigned to the variable.

Reproducing a Random Sequence

To compute the number, RANDOM uses the values of the options RANDOM.SEED.1 and RANDOM.SEED.2, and then changes the values for the next time.

When you want to reproduce the same sequence of random numbers when you are developing and debugging your application programs, set RANDOM.SEED.1 and RANDOM.SEED.2 to some specific values *just before* using RANDOM. To duplicate the sequence, set these options to the same values *just before* using RANDOM again. Then changes in the behavior of your programs will be caused by your changes to the programs and not by differing sequences of random numbers.

Creating Seed Values

When you create your own seeds, set both RANDOM.SEED.1 and RANDOM.SEED.2 to odd numbers. This practice enhances the randomness of the numbers that are produced.

Examples

Example 20–1 Producing Random Numbers

This example assigns random numbers between 100 and 200 to a variable called `test`, which is dimensioned by `product`.

```
test = RANDOM(100 200)
REPORT test
```

These statements produce a report such as the following.

```
PRODUCT            TEST
-------------- ----------
Tents              122.93
Canoes             176.69
Racquets           168.32
Sportswear         150.92
Footwear           187.46
```

RANDOM.SEED.1 and RANDOM.SEED.2

The RANDOM.SEED.1 and RANDOM.SEED.2 options specify values used by RANDOM when computing random numbers. To compute the number, RANDOM uses the values of the options RANDOM.SEED.1 and RANDOM.SEED.2, and then changes the values for the next time.

When you want to reproduce the same sequence of random numbers when you are developing and debugging your application programs set RANDOM.SEED.1 and RANDOM.SEED.2 to some specific values *just before* using RANDOM.

Data type

INTEGER

Syntax

RANDOM.SEED.1|RANDOM.SEED.2 = *n*

Arguments

n
An integer expression that specifies the value to use when generating random numbers. The default is for RANDOM.SEED.1 is 12345 and RANDOM.SEED.2 is 1073.

Notes

Reproducing a Random Sequence

As illustrated in Example 20–1, "Producing Random Numbers" on page 20-4, when you want to reproduce the same sequence of random numbers when you are developing and debugging your application programs, set RANDOM.SEED.1 and RANDOM.SEED.2 to some specific values *just before* using RANDOM. To duplicate the sequence, set these options to the same values *just before* using RANDOM again. Then changes in the behavior of your programs will be caused by your changes to the programs and not by differing sequences of random numbers.

Creating Seed Values

When you create your own seeds, set both RANDOM.SEED.1 and RANDOM.SEED.2 to odd numbers. This practice enhances the randomness of the numbers that are produced.

Examples

Example 20–2 Explicitly Seeding RANDOM for a Test

Assume that you have the following dimension and variable in your analytic workspace

```
DEFINE id DIMENSION TEXT
DEFINE myvar VARIABLE INTEGER <id>
```

As shown in the following code, when you use RANDOM to populate `myvar` without seeding it first. Oracle OLAP populates `myvar` with different values each time the RANDOM executes.

```
myvar = 0
myvar = RANDOM (10, 20)
REPORT myvar

ID                 MYVAR
-------------- ----------
a1                     11
a2                     19
a3                     14

myvar = 0
myvar = RANDOM (10, 20)
REPORT myvar

ID                 MYVAR
-------------- ----------
a1                     16
a2                     13
a3                     12
```

Now, assume that you want to write a test that uses RANDOM to create predictable values for `myvar`. As the following code illustrates, to ensure that the results of

RANDOM are the same from time to time, you must set the values of RANDOM.SEED.1 and RANDOM.SEED.2 right before the execution of RANDOM.

```
myvar = 0
RANDOM.SEED.1 = 5
RANDOM.SEED.2 = 3
myvar = RANDOM (10, 20)
REPORT myvar

ID                MYVAR
-------------- ----------
a1                     10
a2                     16
a3                     13

myvar = 0
RANDOM.SEED.1 = 5
RANDOM.SEED.2 = 3
myvar = RANDOM (10, 20)
REPORT myvar

ID                MYVAR
-------------- ----------
a1                     10
a2                     16
a3                     13
```

The values that you set for RANDOM.SEED.1 and RANDOM.SEED.2 do not stay the same throughout a session. As the following code illustrates, when you do *not* reseed with the same values before each execution, the values produced by RANDOM are *not* the same.

```
myvar = 0
RANDOM.SEED.1 = 5
RANDOM.SEED.2 = 3
myvar = RANDOM (10, 20)
REPORT myvar

ID                 MYVAR
-------------- ----------
a1                     10
a2                     16
a3                     13

myvar = 0
myvar = RANDOM (10, 20)
REPORT myvar

ID                 MYVAR
-------------- ----------
a1                     11
a2                     16
a3                     20
```

RANK

The RANK function computes the rank of values in a numeric expression.

Return Value

DECIMAL

Syntax

RANK(*expression method* [BASEDON *dimension-list*])

Arguments

expression
The numeric expression for which rankings are to be computed.

method
The method to use in computing the rank of the values in *expression*. The *method* argument can be one of the following keywords. See also "Results of Method Values" on page 20-10.

Table 20–1 Methods for Computing RANK

Method	Description
MIN	Identical values get the same minimum rank.
MAX	Identical values get the same maximum rank.
AVERAGE	Identical values get the same average rank.
PACKED	Identical values get the same rank but the results are packed into consecutive integers.
UNIQUE	All values get a unique rank; for identical values the rank is arbitrary.
PERCENTILE	Values are ranked from 1 to 100, based on the relative frequency of their occurrence in the expression.
DECILE	Values are ranked from 1 to 10, based on the relative frequency of their occurrence in the expression.
QUARTILE	Values are ranked from 1 to 4, based on the relative frequency of their occurrence in the expression.

BASEDON *dimension-list*

An optional list of one or more of the dimensions of *expression* to include in the ranking. When you do not specify the dimensions, then RANK bases the ranking on all of the dimensions of *expression*.

Notes

Results of Method Values

This note describes the results of the different methods of ranking values. The results are based on the `sales2` variable, which is described in "Ranking Values" on page 20-11, with the `geography` dimension limited to `G2` as the following statements demonstrate.

```
LIMIT geography TO 'G2'
SORT items D sales2
REPORT DOWN geography sales2
```

The preceding statements produce the following output.

```
               ------------------------SALES2------------------------
               ------------------------ITEMS-------------------------
GEOGRAPHY        ITEM4      ITEM2      ITEM3      ITEM1      ITEM5
-------------- ---------- ---------- ---------- ---------- ----------
G2                  25.00      20.00      20.00      15.00       7.00
```

Table 20–2, " Results of Different Methods of Ranking" on page 20-10 shows the results of the different methods of ranking that are produced by a statement of the form

```
REPORT DOWN geography RANK(sales2 MIN BASEDON items)
```

with the different *method* keywords substituted for MIN.

Table 20–2 Results of Different Methods of Ranking

Methods	(ITEM4, G2) = 25	(ITEM2, G2) = 20	(ITEM3, G2) = 20	(ITEM1,G2) = 15	(ITEM5,G2) = 7
MIN	1	2	2	4	5
MAX	1	3	3	4	5
AVERAGE	1	2.5	2.5	4	5
PACKED	1	2	2	3	4
UNIQUE	1	2	3	4	5

Table 20–2 (Cont.) Results of Different Methods of Ranking

Methods	(ITEM4, G2) = 25	(ITEM2, G2) = 20	(ITEM3, G2) = 20	(ITEM1,G2) = 15	(ITEM5,G2) = 7
PERCENTILE	100	62	62	25	1
DECILE	10	7	7	3	1
QUARTILE	4	3	3	1	1

Note that the value that is returned by the UNIQUE method for `Item2` and `Item3` can be either `2` or `3`, since the RANK function randomly assigns a unique rank for identical values in the expression.

Examples

Example 20–3 Ranking Values

These examples use the following `geography` and `items` dimensions and `sales2` variable.

```
DEFINE geography DIMENSION TEXT
MAINTAIN geography ADD 'g1' 'g2' 'g3'
DEFINE items DIMENSION TEXT
MAINTAIN items ADD 'Item1' 'Item2' 'Item3' 'Item4' 'Item5'
DEFINE sales2 DECIMAL <geography items>
```

Assume the SALES2 variable has the following data values.

```
               -------------SALES2-------------
               -----------GEOGRAPHY------------
ITEMS              g1         g2         g3
-------------- ---------- ---------- ----------
Item1               30.00      15.00      12.00
Item2               10.00      20.00      18.00
Item3               15.00      20.00      24.00
Item4               30.00      25.00      25.00
Item5                  NA       7.00      21.00
```

This statement reports the results of using the MIN method to rank the `sales2` values based on the `items` dimension.

```
report rank(sales2 min basedon items)
```

The preceding statement produces the following output.

```
               -RANK(SALES2 MIN BASEDON ITEMS)-
               -----------GEOGRAPHY------------
ITEMS              g1         g2         g3
-------------- ---------- ---------- ----------
Item1                1.00       4.00       5.00
Item2                4.00       2.00       4.00
Item3                3.00       2.00       2.00
Item4                1.00       1.00       1.00
Item5                  NA       5.00       3.00
```

This statement reports the results of using the MIN method to rank the `sales2` values based on the geography dimension.

```
REPORT RANK(sales2 MIN BASEDON geography)
```

The preceding statement produces the following output.

```
               ----RANK(SALES2 MIN BASEDON-----
               -----------GEOGRAPHY)-----------
               -----------GEOGRAPHY------------
ITEMS              g1         g2         g3
-------------- ---------- ---------- ----------
Item1                1.00       2.00       3.00
Item2                3.00       1.00       2.00
Item3                3.00       2.00       1.00
Item4                1.00       2.00       2.00
Item5                  NA       2.00       1.00
```

This statement reports the results of using the MIN method to rank the `sales2` values based on all of its dimensions.

```
REPORT RANK(sales2, MIN)
```

The preceding statement produces the following output.

```
               -------RANK(SALES2, MIN)--------
               -----------GEOGRAPHY------------
ITEMS              g1         g2         g3
-------------- ---------- ---------- ----------
Item1                1.00      10.00      12.00
Item2               13.00       7.00       9.00
Item3               10.00       7.00       5.00
Item4                1.00       3.00       3.00
Item5                  NA      14.00       6.00
```

RECAP

The RECAP command sends statements that were previously entered during the current session to the current outfile or to a file that you specify. The statements are copied from the command log, which is a list of up to 256 of the most recently entered statements.

Syntax

```
RECAP [number|ALL] [ 'search-text' ] [ FILE file-id ]
```

Arguments

number
A positive integer that indicates the number of statements to be provided. When you specify *search-text,* RECAP provides this number of statements from the subset that contains the *search-text* string. When you do not specify *search-text,* RECAP provides this number of statements from the most recently executed portion of the command log. The default number is 10.

ALL
When you specify *search-text,* ALL requests every statement that meets the search requirements. When you do not specify *search-text,* ALL requests every statement in the command log.

search-text
A quoted text literal. When you specify this argument, RECAP searches the statements in the command log for the ones that contain *search-text.* The search is *not* case-sensitive. These statements will then compose the subset from which RECAP provides *number* or ALL statements.

FILE *file-id*
Writes the output of the RECAP command to the specified file. The *file-id* is a text expression that represents the name of the file. The name must be in a standard format for a file identifier.

Notes

Order of *search-text* Argument

When you use both the *search-text* and the ALL or *number* arguments, you must specify *search-text* second.

RECAP with No Argument

When you specify RECAP without an argument, the ten most recent statements are provided.

Command Log

The command log is a list maintained internally by Oracle OLAP. It contains the statements that were executed most recently in your session. The maximum number of statements in the command log is 256. When you start a new session, the list is empty.

Re-Executing Statements

You can use the output of RECAP to edit a previously executed statement with REEDIT, or reexecute a previously executed statement with REDO.

Statements Not in Command Log

RECAP, REDO, and REEDIT commands are not included in the command log. But the statements re-executed by REDO and re-edited by REEDIT are included.

Identifying Files and Directories

When specifying files and directories in OLAP DML statements, it is good practice to always enclose them in single quotes.

Examples

Example 20–4 Obtaining the Last Three Statements Containing "actual"

The following RECAP command requests the three most recent statements that included the text literal "actual."

```
RECAP 3 'actual'
```

This statement could produce the following output.

```
       COMMAND LOG
    3: dsc actual
    5: report total(actual)
    8: report average(actual)
```

RECNO

The RECNO function reports the current record number of a file opened for reading. It returns `NA` when Oracle OLAP has reached the end of the file.

Return Value

INTEGER

Syntax

RECNO(*fileunit*)

Arguments

fileunit
A file unit number assigned to a file opened for reading in a previous call to the FILEOPEN function.

Notes

Opening Files

Before you can use the RECNO function, you must open the file for reading. When the file unit number is not associated with an open file or the file has been opened for writing, RECNO returns an error.

Using RECNO with FILEGET

RECNO is usually used with FILEREAD or FILENEXT, which read whole records. When you are reading data from a file with the FILEGET function, which can read partial records, RECNO returns the number of times you have read data from the file, not the number of actual records.

LINENUM Option

See also the LINENUM option, which holds the current line number of output.

Records in Text Files

When the file is a text file, a record is delimited by a newline character. When the file is a binary file, you must set the file's LSIZE attribute to the record length with the FILESET command. TEXT is the default file type.

Examples

Example 20–5 Using RECNO with FILEREAD

In the following example code, the FILEREAD command maintains the integer dimension, adding each record number associated with `filename.` The text associated with each record number becomes each value of the variable `textvar`.

```
DEFINE dim1 INTEGER DIMENSION
DEFINE textvar TEXT <dim1>
x = FILEOPEN('filename' R)
FILEREAD x APPEND dim1 = RECNO(x) W 8 TEXTVAR
```

RECURSIVE

The RECURSIVE option controls the ability of a formula or $NATRIGGER expression to call itself.

Syntax

RECURSIVE = {YES|<u>NO</u>}

Arguments

YES
Specifying `YES` allows a formula or $NATRIGGER expression to call itself. Set this option to `YES` when you define a formula or an expression for the $NATRIGGER property that uses a recursive method of computation.

NO
Specifying `NO` prevents a formula or $NATRIGGER expression from calling itself. (Default) When you attempt to evaluate a recursive formula or $NATRIGGER expression, then Oracle OLAP displays an error message, which states that the RECURSIVE option is currently set to `NO`. Until the workspace contains a recursive formula or $NATRIGGER expression, keep this option set to `NO` in order to detect errors that could result in infinite looping behavior.

Notes

For Formulas and $NATRIGGER Expressions Only

When you set RECURSIVE to `YES`, only formulas and $NATRIGGER property expressions are affected. This option does not affect programs; that is, a program can be recursive regardless of the setting of the RECURSIVE option unless the program is an $NATRIGGER expression. An $NATRIGGER expression cannot call itself unless the RECURSIVE option is `YES`.

Limiting $NATRIGGER Recursion

You can limit the depth of recursion for $NATRIGGER property expressions with the TRIGGERMAXDEPTH option, which sets the maximum number of $NATRIGGER expressions that Oracle OLAP executes simultaneously.

REDO

The REDO command re-executes a statement that you entered earlier in your session. The statement is retrieved from the command log, which is a list of up to 256 statements that you have entered most recently during the current session. REDO enables you to changes in the statement before it is re-executed.

Syntax

REDO [*number*|*index*] '*original*' '*replacement*' [*specifier*]

Arguments

number
A positive integer that indicates the number of the statement to be re-executed. You can display the statements, with their numbers, using the RECAP command.

index
A negative integer or zero that indicates the position of the statement to be re-executed relative to the end of the command log. The most recent statement is `0`, the one before that is `-1`, and so on. The default is `0`.

original
A text literal that is part of the statement to be re-executed.

replacement
A text literal that should replace *original* when the statement is re-executed.

specifier
One of the specifiers listed in Table 20–3, " Valid Values for REDO specifier". Each specifier indicates where text replacement should occur in the re-executed statement.

Table 20–3 Valid Values for REDO specifier

Specifier	Meaning
`FIRST`	Indicates that only the first occurrence of *original* should be changed to *replacement*.
`LAST`	Indicates that only the last occurrence of *original* should be changed to *replacement*.

Table 20–3 (Cont.) Valid Values for REDO specifier

Specifier	Meaning
n	A number indicating which occurrence of *original* should be changed to *replacement.* For example, `3` indicates the third occurrence.
`ALL`	Indicates that all occurrences of *original* should be changed to *replacement*
*	Indicates that all occurrences of *original* should be changed to *replacement*.

The default is ALL. When you do not provide a specifier, all occurrences of *original* will be changed to *replacement.*

Notes

REDO with No Argument

When you type REDO without an argument, the most recent statement will be re-executed.

Command Log

The command log is a list maintained internally by Oracle OLAP. It contains the statements executed most recently in your session. The maximum number of statements in the command log is 256. When you start a new session, the list is empty.

REDO and REEDIT

The REEDIT command is similar to REDO, except that the statement is not executed after you edit it. It is placed in the command log so that you can edit it again.

Statements Not in the Command Log

RECAP, REDO, and REEDIT commands are not included in the command log. But the statements re-executed by REDO and re-edited by REEDIT are included.

Case-Sensitivity

When matching *original* with the text of the statement to be re-executed, REDO ignores case differences. For example, assume you specify `AT` as *original,* REDO will match it with `at`, `At`, `aT`, or `AT` in the statement.

When replacing *original* with *replacement*, REDO retains the case of all characters in *replacement*. For example, assume you specify `ShOw` as *replacement,* that is exactly how it will appear in the re-executed statement.

Examples

Example 20–6 Redoing a Report

The following output is the result of `recap 2` statement.

```
   COMMAND LOG
6: fetch w 20 down division total(actual)
7: listnames
```

The following REDO statement re-executes the FETCH statement with a different variable.

```
REDO 6 'actual' 'budget'
```

REEDIT

The REEDIT command edits a statement that you entered earlier in your session. The statement is retrieved from the command log, which is a list of up to 256 statements that you have entered most recently during the current session. REEDIT enables you to change the statement without executing it, so you can edit it sequentially.

Syntax

REEDIT [*number*|index] '*original*' '*replacement*' [*specifier*]

Arguments

number
A positive integer that indicates the number of the statement to be edited. You can display the statements, with their numbers, using the RECAP command.

index
A negative integer (or zero) that indicates the position of the statement to be edited relative to the end of the command log. The most recent statement is `0`, the one before that is `-1`, and so on. The default is `0`.

original
A text literal that is part of the statement to be edited.

replacement
A text literal that should replace *original* when the statement is edited.

specifier
One of the specifiers listed in Table 20–4, " Valid Values for REEDIT specifier". Each specifier indicates where text replacement should occur in the edited statement.

Table 20–4 Valid Values for REEDIT specifier

Specifier	Meaning
`FIRST`	Indicates that only the first occurrence of *original* should be changed to *replacement.*
`LAST`	Indicates that only the last occurrence of *original* should be changed to *replacement.*

Table 20–4 (Cont.) Valid Values for REEDIT specifier

Specifier	Meaning
n	A number indicating which occurrence of *original* should be changed to *replacement.* For example, `3` indicates the third occurrence.
`ALL`	Indicates that all occurrences of *original* should be changed to *replacement*
*	Indicates that all occurrences of *original* should be changed to *replacement.*

The default is ALL. When you do not provide a specifier, all occurrences of *original* will be changed to *replacement.*

Notes

REEDIT with No *number* or *index* Argument

When you type REEDIT without *number* or *index,* the most recent statement will be edited.

Command Log

The command log is a list maintained internally by Oracle OLAP. It contains the statements executed most recently in your session. The maximum number of statements in the command log is 256. When you start a new session, the list is empty.

REDO and REEDIT

The REDO command is similar to REEDIT, except that the statement is executed after you edit it.

Statements Not in the Command Log

RECAP, REDO, and REEDIT commands are not included in the command log. But the statements re-executed by REDO and re-edited by REEDIT are included.

Case-Sensitivity

When matching *original* with the text of the statement to be edited, REEDIT ignores case differences. For example, assume you specify `AT` as *original,* REEDIT will match it with `at`, `At`, `aT`, or `AT` in the statement.

When replacing *original* with *replacement,* REEDIT retains the case of all characters in *replacement.* For example, assume you specify `ShOw` as *replacement,* that is exactly how it will appear in the edited statement.

Examples

Example 20–7 Editing Multiple Values in a LIMIT Command

The following example illustrates why it could be helpful to use the REEDIT command to edit a statement several times before executing it. With REEDIT commands, you can edit multiple values in a LIMIT command before executing it. When you enter a REDO command, the LIMIT command is executed.

The following output is the result of a `recap 1` statement.

```
   COMMAND LOG
6: limit mydim to 1 to 10, 15 to 20, 24 to 28, 33 to 40
```

The statement

```
REEDIT 6 '1' '2' FIRST
```

produces the following output.

```
7: limit mydim to 2 to 10 , 15 to 20, 24 to 28, 33 to 40
```

The statement

```
REEDIT 7 '15' '18'
```

produces the following output.

```
8: limit mydim to 2 to 10 , 18 to 20, 24 to 28, 33 to 40
```

The statement

```
REDO 8 '40' '41'
```

makes one more change and re-executes the LIMIT command with the new values. It also produces the following output.

```
9: limit mydim to 2 to 10 , 18 to 20, 24 to 28, 33 to 41
```

REGRESS

The REGRESS command calculates a simple multiple linear regression. The optional WEIGHTBY keyword lets you calculate a weighted regression when some of the data points are more reliable than others.

You can then execute REGRESS.REPORT to produce a standard report of the regression. You can also use the INFO function to obtain portions of the results for use in your own customized reports or for further analysis.

> **Tip:** To performing more complex regression analysis use a forecasting context as discussed in "Forecasting Programs" on page 1-16.

Syntax

REGRESS [NOINTERCEPT] *dependent independent*... [WEIGHTBY *weight*]

Arguments

NOINTERCEPT
Directs Oracle OLAP to suppress the constant term (intercept) in the regression equation. The default is to calculate a constant term.

dependent
An expression to be used as the dependent variable in the regression.

In calculating the results, REGRESS loops over all the dimensions of the dependent and independent variables.

independent
One or more expressions to be used as the independent variables (regressors) in the regression.

WEIGHTBY *weight*
Specifies a weighted regression. The numeric expression *weight* supplies the weights for each data point (observation). Giving higher weights to more reliable observations results in a higher quality regression. WEIGHTBY must come last in the REGRESS command.

When *weight* is less than zero for any observation, an error occurs. When *weight* is equal to zero for any observation, that observation is ignored in the calculation. When WEIGHTBY is omitted, an unweighted regression is calculated.

Notes

Using a Forecasting Context

Instead of calculating a simple regression using the REGRESS command, you can perform more complex regression analysis using a forecasting context that you manipulate with the following OLAP DML statements:

1. FCOPEN function -- Creates a forecasting context.
2. FCSET command -- Specifies the characteristics of a forecast.
3. FCEXEC command -- Executes a forecast and populates Oracle OLAP variables with forecasting data.
4. FCQUERY function -- Retrieves information about the characteristics of a forecast or a trial of a forecast.
5. FCCLOSE command -- Closes a forecasting context.

Ignoring NA Values

In performing its calculations, the REGRESS command ignores any observation that has an `NA` value.

Producing a Standard Report

The standard report for a regression shows the coefficient, standard error, and T-ratio for each independent variable; as well as the R-square, F-Statistic, number of observations, and standard error of estimate for the regression. To produce this report, type the following.

```
REGRESS.REPORT
```

Obtaining Results

For information on how to obtain portions of the results of REGRESS for your own reports or further analysis, use an INFO statement.

Further Reading

For an explanation of the uses and interpretation of regression models, we suggest the latest edition of the following book:

Draper, Norman, and Smith, Harry. *Applied Regression Analysis.* New York: John Wiley & Sons, Inc.

Examples

Example 20–8 Simple Regression

The following statements limit the `product` dimension to `Canoes`, then use regression to investigate the influence of advertising, price, and expense on the sales of canoes.

```
LIMIT product TO 'Canoes'
REGRESS NOINTERCEPT sales advertising price expense
```

You can now execute REGRESS.REPORT as illustrated in Example 20–10, "Report for a Simple Regression" on page 20-28 to see the results of the regression.

Example 20–9 Weighted Regression

The following statements use a weighted regression, in which districts are weighted using a variable called `reliability` that has the following definition and values.

```
DEFINE reliability VARIABLE DECIMAL <district>

DISTRICT       RELIABILITY
-------------- -----------
Boston                1.00
Atlanta               0.90
Chicago               1.00
Dallas                0.80
Denver                0.90
Seattle               0.60
```

The following statements perform the regression.

```
REGRESS NOINTERCEPT sales advertising price expense -
WEIGHTBY reliability
```

You can now execute REGRESS.REPORT as illustrated in Example 20–11, "Report for a Weighted Regression" on page 20-29 to see the results of the regression.

REGRESS.REPORT

The `REGRESS.REPORT` program produces a standard report of a regression performed using the REGRESS command.

Syntax

REGRESS.REPORT

Examples

Example 20–10 Report for a Simple Regression

Assume that you have performed the simple regression illustrated in Example 20–8, "Simple Regression" on page 20-27. You can now execute REGRESS.REPORT to see the results of the regression.

```
                              Regression Analysis
                              ===================

       Dependent Variable: SALES
        WEIGHTBY Variable: NONE

Regressor                 Coefficient      Std. Error     T-ratio
--------------------     ------------     ------------    --------
ADVERTISING                      0.36             0.16        2.24
PRICE                           -8.66             1.80       -4.82
EXPENSE                          1.05             0.01       79.69

      Corrected R-square                   1.00
      F-Statistic (2, 141)                   NA
      Number of observations                144
      Standard error of estimate       1,477.16
```

Example 20–11 Report for a Weighted Regression

Assume that you have performed the simple regression illustrated in Example 20–9, "Weighted Regression" on page 20-27. You can now execute REGRESS.REPORT to see the results of the regression.

```
                              Regression Analysis
                              ===================

       Dependent Variable: SALES
        WEIGHTBY Variable: RELIABILITY

Regressor                 Coefficient      Std. Error     T-ratio
--------------------    ------------    ------------    --------
ADVERTISING                      0.44            0.17        2.64
PRICE                           -8.03            1.92       -4.19
EXPENSE                          1.04            0.01       76.45

      Corrected R-square                      1.00
      F-Statistic (2, 141)                      NA
      Number of observations                   144
      Standard error of estimate          1,373.15
```

RELEASE

When an analytic workspace is attached in multiwriter mode, the RELEASE command changes the access mode of the specified variables, relations, valuesets, or dimensions from read/write (acquired) access to read-only access.

Syntax

```
RELEASE [objects] [analytic_workspaces]
```

Arguments

When no parameters are specified, all acquired variables in the current AW are released.

objects

A list of one or more variables, relations, valuesets, or dimension names, separated by commas, that you want to release.

analytic workspaces

A list of analytic workspace names, separated by commas. When you specify an analytic workspace in this list, all acquired objects in that analytic workspace are released after all pending changes are made to them. All changes made to the variables, relations, valuesets, or dimensions before the RELEASE command executes are preserved as private changes after the release command.

Notes

Releasing Non-Updated Objects

Similarly to using the DETACH command for analytic workspaces that has been updated. using RELASE for objects that have been updated does not others to acquire the object until you commit or roll back the transaction. It may still be useful to release an object that has been updated before a commit when one wants to make further what-if changes and later needs to use update command to update all acquired variables.

Releasing a Dimension Causes the Dimension to Revert

When you release an acquired dimension, the dimension is automatically reverted (see REVERT for an explanation of what it means to revert a dimension).

As the following code illustrates, releasing an acquired dimension causes an automatic revert.

User A issues the following OLAP DML statements.

```
AW ATTACH myworkspace MULTI
ACQUIRE RESYNC time WAIT
MAINTAIN time ADD 'Y2002'
actuals (time 'Y2002', ...) = 37
REPORT time --> ..., 'Y2002'
SHOW actuals (time 'Y2002', ...) --> 37
RELEASE time
REPORT time --> ... (no 'Y2002')
AW ATTACH myworkspace MULTI
ACQUIRE RESYNC actuals, time WAIT
MAINTAIN time ADD 'Y2002'
actuals (time 'Y2002', ...) = 37
REPORT time --> ..., 'Y2002'
SHOW actuals (time 'Y2002', ...) --> 37
REVERT time
REPORT time --> ... (no 'Y2002')
MAINTAIN time ADD 'Y2002'
REPORT time --> ..., 'Y2002'
SHOW actuals (time 'Y2002', ...) --> NA
```

Examples

Example 20–12 Acquiring, Updating and Releasing Objects

A classic use of multiwriter attachment mode is to allow two users to modify two different objects in the same analytic workspace. For example, assume that an analytic workspace has two variables: `actuals` and `budget`. Assume also that one user (user A) wants to modify `actuals`, while another user (user B) wants to modify `budget`. In this case, after attaching the analytic workspace in the multiwriter mode, each user acquires the desired variable, performs the desired modification, updates, commits the changes, and then, either detaches the workspace or releases the acquired variable.

User A executes the following statements.

```
AW ATTACH myworkspace MULTI
ACQUIRE actuals
... make modifications
UPDATE MULTI actuals
COMMIT
RELEASE actuals
AW DETACH myworkspace
```

While, at the same time, User B executes the following statements.

```
AW ATTACH myworkspace MULTI
ACQUIRE budget
…make modifications
UPDATE MULTI budget
COMMIT
RELEASE budget
AW DETACH myworkspace
```

Example 20–13 Using RELEASE After UPDATE But Before COMMIT

Using a RELEASE statement does not always allow other users to acquire the released variable. For example, when you have updated a variable but have not committed the changes, the execution of a RELEASE statement has no effect on other users until a commit occurs. However, when you use a simple UPDATE to update all acquired variables, it can be useful to release a variable after updating it but before committing it. When a variable is released after the first update, it is not be included in the list of updated variables for the second update. The following code illustrates situations where user B1 releases `budget` at different times.

Assume that User B1 issues the following statements

```
AW ATTACH myworkspace MULTI
ACQUIRE RESYNC budget WAIT
make changes C1
RELEASE budget
UPDATE
make changes C2
UPDATE
COMMIT
```

User B2 could issue the following statements

```
AW ATTACH myworkspace MULTI
ACQUIRE RESYNC budget WAIT
```

User B2 gets `budget` and sees no changes and issues the following statements.

```
...
AW ATTACH myworkspace MULTI
ACQUIRE RESYNC budget WAIT
make changes C1
UPDATE
RELEASE budget
make changes C2
UPDATE
COMMIT
...
AW ATTACH myworkspace MULTI
ACQUIRE RESYNC budget WAIT
```

Alternatively, User B2 gets `budget` and sees changes C1 and issues the following statements.

```
AW ATTACH myworkspace MULTI
ACQUIRE RESYNC budget WAIT
make changes C1
UPDATE
make changes C2
RELEASE budget
UPDATE
COMMIT
...
AW ATTACH myworkspace MULTI
ACQUIRE RESYNC budget WAIT
```

Or, as another alternative, User B2 gets `budget` and sees changes C1 and issues the following statements.

```
AW ATTACH myworkspace MULTI
ACQUIRE RESYNC budget WAIT
make changes C1
UPDATE
make changes C2
UPDATE
COMMIT
RELEASE budget
...
AW ATTACH myworkspace MULTI
ACQUIRE RESYNC budget WAIT
```

At this point, User B2 gets `budget` and sees changes C2

REM

The REM function returns the remainder after one numeric expression is divided by another.

Return Value

DECIMAL

Syntax

REM(*expression1 expression2*)

Arguments

expression

REM returns the remainder of *expression1* divided by *expression2.*

Examples

Example 20–14 Calculating a Remainder

This example illustrates the use of REM to find the remainder after `14` is divided by `5`. The statement

```
SHOW REM(14 5)
```

produces the following result.

```
4.00
```

REMBYTES

The REMBYTES function removes one or more bytes from a text expression and returns the value that remains.

Return Value

TEXT

Syntax

REMBYTES(*text-expression start* [*length*])

Arguments

text-expression

The expression from which REMBYTES removes bytes. When the characters to be removed from *text-expression* contain embedded line breaks, these breaks are also removed. Other line breaks are preserved. Removed line breaks are not counted toward the total number of characters removed.

start

An integer that represents the character position at which to begin removing characters. The position of the first character in *text-expression* is `1`. When the value of *start* is greater than the length of *text-expression*, REMBYTES simply returns *text-expression*.

length

An integer that represents the number of characters to be removed. When *length* is not specified, only the character at *start* is removed.

Notes

Single-Byte Characters

When you are using a single-byte character set, you can use the REMCHARS function instead of the REMBYTES function.

NTEXT Data Type

This function does not accept NTEXT arguments, because it is oriented toward byte-manipulation instead of character manipulation. It always returns values of type TEXT. When you must use this function on NTEXT values, use the CONVERT or TO_CHAR function to convert the NTEXT value to TEXT.

Examples

Example 20–15 Using REMBYTES to Remove a Substring

This example shows how to remove the substring `there` from the text value `hellotherejoe`.

The statement

```
SHOW REMBYTES('hellotherejoe', 6, 5)
```

produces the following output.

```
hellojoe
```

Example 20–16 Removing a Single Byte

This example shows how to remove the character `t` from the text value `hellotherejoe`.

```
SHOW REMBYTES('hellotherejoe', 6)
```

produces the following output.

```
helloherejoe
```

REMCHARS

The REMCHARS function removes one or more characters from a text expression and returns the value that remains.

Return Value

TEXT or NTEXT

Syntax

REMCHARS(*text-expression start* [*length*])

Arguments

text-expression
The expression from which REMCHARS removes characters. When the characters to remove from *text-expression* contain embedded line breaks, these breaks are also removed. Other line breaks are preserved. Removed line breaks are not counted toward the total number of characters removed.

When you specify a TEXT expression, the return value is TEXT. When you specify an NTEXT expression, the return value is NTEXT.

start
An integer that represents the character position at which to begin removing characters. The position of the first character in *text-expression* is `1`. When the value of *start* is greater than the length of *text-expression*, REMCHARS simply returns *text-expression*.

length
An integer that represents the number of characters to be removed. When *length* is not specified, only the character at *start* is removed.

Notes

multibyte Characters

When you are using a multibyte character set, you can use the REMBYTES function instead of the REMCHARS function.

Examples

Example 20–17 Using REMCHARS to Remove a Substring

This example shows how to remove the substring `there` from the text value `hellotherejoe`.

The statement

```
SHOW REMCHARS('hellotherejoe', 6, 5)
```

produces the following output.

```
hellojoe
```

Example 20–18 Removing a Single Character

This example shows how to remove the character `t` from the text value `hellotherejoe`.

```
SHOW REMCHARS('hellotherejoe', 6)
```

produces the following output.

```
helloherejoe
```

REMCOLS

The REMCOLS function removes specified columns from every line of a multiline TEXT value. The function returns a multiline text value that includes only the remaining columns.

Columns refer to the character positions in each line of a multiline TEXT value. The first character in each line is in column one, the second is in column two, and so on.

Return Value

TEXT or NTEXT

Syntax

REMCOLS(*text-expression start* [*length*])

Arguments

text-expression

The text expression from which the specified columns should be removed. When *text-expression* is a multiline TEXT value, the characters in the specified columns are removed from each one of its lines.

When you specify a TEXT expression, the return value is TEXT. When you specify an NTEXT expression, the return value is NTEXT.

start

An integer, between 1 and 4,000, representing the column position at which to begin removing columns. The column position of the first character in each line of *text-expression* is `1`.

length

An integer representing the number of columns to be removed. When you do not specify *length,* REMCOLS removes only the starting column.

Notes

Number of Lines Returned

REMCOLS always returns a TEXT value that has the same number of lines as *text-expression,* though some of the lines may be empty.

Start Column Beyond the End of a Line

When you specify a starting column that is to the right of the last character in a given line in *text expression,* the corresponding line in the return value will be identical to the given line.

Length That Goes Beyond the End of a Line

When you specify a length that exceeds the number of characters that follow the starting position in a given line in *text expression,* the corresponding line in the return value will include only the characters that precede the starting column.

Examples

Example 20–19 Removing Text Columns

In the following example, four columns are removed from each line of CITYLIST, starting from the second column.

```
DEFINE citylist VARIABLE TEXT
CITYLIST = 'Boston\nHouston\nChicago\nDenver'
```

The statement

```
SHOW citylist
```

produces the following output.

```
Boston
Houston
Chicago
Denver
```

The statement

```
SHOW REMCOLS(citylist 2 4)
```

produces the following output.

```
Bn
Hon
Cgo
Dr
```

REMLINES

The REMLINES function removes one or more lines from a multiline TEXT expression and returns the value that remains.

Return Value

TEXT or NTEXT

Syntax

REMLINES(*text-expression start* [*length*])

Arguments

text-expression
A multiline text expression from whose values REMLINES removes one or more lines. When you specify a TEXT expression, the return value is TEXT. When you specify an NTEXT expression, the return value is NTEXT.

start
An integer that represents the line number at which to begin removing lines. The position of the first line in *text-expression* is `1`.

length
An integer that represents the number of lines to be extracted. When you do not specify *length,* only the line at *start* is removed.

Examples

Example 20–20 Removing Text Lines

This example shows how to remove the second line from a multiline text value in a variable called `mktglist` with the following values.

```
Salespeople
Products
Services
```

The statement

```
SHOW REMLINES(mktglist, 2)
```

produces the following output.

```
Salespeople
Services
```

RENAME

The RENAME command changes the name of an analytic workspace or an object in an analytical workspace.

Syntax

RENAME *oldname newname* [AW *workspace*]

Arguments

oldname

The name of an existing analytic workspace or an existing object in an analytic workspace. You can specify a qualified object name to indicate the attached workspace in which the object resides. As an alternative, you can use the AW keyword to specify the workspace. Do not use both.

When you do not use a qualified object name or the AW keyword to specify a workspace, the object is renamed in the current workspace.

For an unnamed composite, use the same syntax that was used to create it. See "Naming an Unnamed Composite" on page 20-45.

newname

The new name.

- The new name of an analytic workspace cannot duplicate any other name of a workspace in the schema in which the workspace is defined. Choose a name according to the rules for naming analytic workspaces (see the AW command).
- The new name of an analytic workspace object cannot duplicate any other name in the workspace in which the object exists. Choose a name according to the rules for naming analytic workspace objects (see the DEFINE command). To change a named composite to an unnamed composite, use the SPARSE keyword as the *newname* argument. See "Unnaming a Named Composite" on page 20-45.

AW *workspace*

The name of an attached workspace in which you wish to rename the object. When you do not use a qualified object name or the AW keyword to specify a workspace, the object is renamed in the current workspace.

Notes

Updating Associated Objects

When you change the name of a variable, objects that use that variable, such as formulas, are *not* automatically updated.

When you change the name of a dimension, the definitions of any objects that are dimensioned by that dimension are automatically updated. Additionally, any valuesets for the renamed dimension are automatically updated for the new name.

RENAME and PERMIT

You may not rename an object when a PERMIT command denies you the right to change its permission. Renaming an object does not affect permission associated with it.

Naming an Unnamed Composite

You can name an unnamed composite with the RENAME command. The following example assigns the name `m.prod` to an unnamed composite that is dimensioned by `market` and `product`.

```
RENAME SPARSE <market product> m.prod
```

Unnaming a Named Composite

You can change a named composite to an unnamed composite when the composite has no properties or permission restrictions and when there is at least one object dimensioned by it. In addition, there cannot be an unnamed composite with the same dimensions in the same order as the named composite, and the named composite cannot be used in the dimension list of any unnamed composite. To change a named composite to an unnamed composite, use the SPARSE keyword as the *newname* argument. The following example changes the named composite `m.prod` to an unnamed composite.

```
RENAME m.prod SPARSE
```

Restrictions on Renaming Composites

You cannot rename a composite when it is a base dimension of an unnamed composite, or when one of its base dimensions is an unnamed composite.

Examples

Example 20–21 Renaming a Program

This statement changes the name of the program `testreport` to `sales.report`.

```
RENAME testreport sales.report
```

REPLBYTES

The REPLBYTES function replaces one or more bytes in a text expression.

Return Value

TEXT

Syntax

REPLBYTES(*text-expression replacement* [*start*])

Arguments

text-expression

The expression in which REPLBYTES replaces bytes. When the bytes to replace from *text-expression* contain embedded line breaks, these breaks are removed. Other line breaks are preserved. Removed line breaks are not counted toward the total number of bytes replaced. Line breaks in the replacement expression are retained in the output of REPLBYTES, but are likewise not counted.

replacement

A text expression that contains one or more bytes that will replace existing bytes in *text-expression.*

start

An integer that represents the byte position at which to begin replacing bytes. The position of the first byte in *text-expression* is 1. When you omit this argument, REPLBYTES starts with the first byte. REPLBYTES replaces as many bytes of *text-expression* as are required for the bytes specified by *replacement.* When the value of *start* is greater than the length of *text-expression*, REPLBYTES simply returns *text-expression*.

Notes

Single-Byte Characters

When you are using a single-byte character set, you can use the REPLCHARS function instead of the REPLBYTES function.

NTEXT Data Type

This function does not accept NTEXT arguments, because it is oriented toward byte-manipulation instead of character manipulation. It always returns values of type TEXT. When you must use this function on NTEXT values, use the CONVERT or TO_CHAR function to convert the NTEXT value to TEXT.

Changing Occurrences of a Specified String in a Text Value

You can use the CHANGECHARS function to change one or more occurrences of a specified string in a text value to another string.

Examples

Example 20–22 Replacing Text as Bytes

This example shows how to replace a portion of the text value `Hello there, Joe`.

The statement

```
SHOW REPLBYTES('Hello there, Joe', 'Jane', 14)
```

produces the following output.

```
Hello there, Jane
```

Example 20–23 How REPLBYTES Handles Line Breaks

This example shows how REPLBYTES preserves but ignores line breaks.

```
var1 = JOINLINES('Hello' 'there' 'Joe')
var2 = JOINLINES('Hi' 'Jane')
```

The statement

```
SHOW REPLBYTES(var1 var2)
```

produces the following output.

```
Hi
Janehere
Joe
```

REPLBYTES has replaced the first 6 bytes of `var1` (`Hellot` of `HellothereJoe`) with the 6 bytes of `var2` (`HiJane`). It has preserved the line breaks following `Hi` (from `var2`) and `there` (from `var1`).

To replace all 13 bytes in var1, you must specify 13 replacement bytes; for example, you can add 7 spaces after `Jane`.

```
var2 = JOINLINES('Hi' 'Jane       ')
```

The statement

```
SHOW REPLBYTES(var1 var2)
```

produces the following output.

```
Hi
Jane
```

REPLCHARS

The REPLCHARS function replaces one or more characters in a text expression.

Return Value

TEXT or NTEXT

This function accepts TEXT values and NTEXT values as arguments. The data type of the return value depends on the data type of the values specified for the arguments:

- When all arguments are TEXT values, the return value is TEXT.
- When all arguments are NTEXT values, the return value is NTEXT.
- When the arguments include both TEXT and NTEXT values, the function converts all TEXT values to NTEXT before performing the function operation, and the return value is NTEXT.

Syntax

REPLCHARS(*text-expression characters* [*start*])

Arguments

text-expression
The expression in which characters are to be replaced. When the characters to be replaced from *text-expression* contain embedded line breaks, these breaks are removed. Other line breaks are preserved. Removed line breaks are not counted toward the total number of characters replaced. Line breaks in the replacement expression are retained in the output of REPLCHARS, but are likewise not counted.

characters
A text expression that contains one or more characters that will replace existing characters in *text-expression.*

start
An integer that represents the character position at which to begin replacing characters. The position of the first character in *text-expression* is `1`. When you omit this argument, REPLCHARS starts with the first character. REPLCHARS replaces as many characters of *text-expression* as are required for the specified new *characters.*

When the value of *start* is greater than the length of *text-expression*, REPLCHARS simply returns *text-expression*.

Notes

multibyte Characters

When you are using a multibyte character set, you can use the REPLBYTES function instead of the REPLCHARS function.

Changing Occurrences of a Specified String in a Text Value

You can use the CHANGECHARS function to change one or more occurrences of a specified string in a text value to another string.

Examples

Example 20–24 Replacing Text Characters

This example shows how to replace a portion of the text value `Hello there, Joe`.

The statement

```
SHOW REPLCHARS('Hello there, Joe', 'Jane', 14)
```

produces the following output.

```
Hello there, Jane
```

Example 20–25 How REPLCHARS Handles Line Breaks

This example shows how REPLCHARS preserves but ignores line breaks.

```
var1 = JOINLINES('Hello' 'there' 'Joe')
var2 = JOINLINES('Hi' 'Jane')
```

The statement

```
show REPLCHARS(var1 var2)
```

produces the following output.

```
Hi
Janehere
Joe
```

REPLCHARS has replaced the first 6 characters of `var1` (`Hellot` of `HellothereJoe`) with the 6 characters of `var2` (`HiJane`). It has preserved the line breaks following `Hi` (from `var2`) and `there` (from `var1`).

To replace all 13 characters in `var1`, you must specify 13 replacement characters; for example, you can add 7 spaces after `Jane`.

```
var2 = JOINLINES('Hi' 'Jane       ')
```

The statement

```
SHOW REPLCHARS(var1 var2)
```

produces the following output.

```
Hi
Jane
```

REPLCOLS

The REPLCOLS function replaces some or all of the character columns in one multiline TEXT value with the columns of another. The function returns a multiline TEXT value composed of the resulting lines.

Columns refer to the character positions in each line of a multiline TEXT value. The first character in each line is in column one, the second is in column two, and so on.

Return Value

TEXT or NTEXT

This function accepts TEXT values and NTEXT values as arguments. The data type of the return value depends on the data type of the values specified for the arguments:

- When all arguments are TEXT values, the return value is TEXT.
- When all arguments are NTEXT values, the return value is NTEXT.
- When the arguments include both TEXT and NTEXT values, the function converts all TEXT values to NTEXT before performing the function operation, and the return value is NTEXT.

Syntax

REPLCOLS(*text-expression columns* [*start*])

Arguments

text-expression
The text expression in which you want to replace one or more columns.

columns
A text expression containing one or more lines. This expression provides the columns to replace some or all of the columns in *text-expression.*

start
An integer, between 1 and 4,000, representing the column position at which to begin replacing. The column position of the first character in each line of *text-expression* is 1. When you do not specify *start*, replacement begins with Column 1.

Notes

Number of Lines Returned

The number of lines in the return value is always the same as the number of lines in *text-expression*. When the *columns* text expression has fewer lines, REPLCOLS repeats its last line in each subsequent line of the return value.

Start Column Beyond the End of a Line

When you specify a starting column that is to the right of the last character in a given line in *text-expression*, the corresponding line in the return value will have spaces filling in the intervening columns. See Example 20–26, "Joining and Aligning Columns" on page 20-54.

Examples

Example 20–26 Joining and Aligning Columns

In the following example, the `citylist` and `cityreps` lines are joined so that the values are aligned, one under the other. The replacement begins at Column 11. When JOINCOLS were used instead of REPLCOLS, the `cityreps` list would be misaligned.

The statement

```
SHOW citylist
```

produces the following output.

```
Boston
Houston
Chicago
Denver
```

The statement

```
SHOW cityreps
```

produces the following output.

```
Brady
Lopez
Alfonso
Cody
```

The statement

```
SHOW REPLCOLS(citylist cityreps 11)
```

produces the following output.

```
Boston     Brady
Houston    Lopez
Chicago    Alfonso
Denver     Cody
```

REPLLINES

The REPLLINES function replaces one or more lines in a multiline TEXT expression.

Return Value

TEXT or NTEXT

This function accepts TEXT values and NTEXT values as arguments. The data type of the return value depends on the data type of the values specified for the arguments:

- When all arguments are TEXT values, the return value is TEXT.
- When all arguments are NTEXT values, the return value is NTEXT.
- When the arguments include both TEXT and NTEXT values, the function converts all TEXT values to NTEXT before performing the function operation, and the return value is NTEXT.

Syntax

REPLLINES(*text-expression lines* [*start*])

Arguments

text-expression
A multiline text expression in which you want to replace one or more lines.

lines
A text expression that contains one or more lines that will replace existing lines in *text-expression.*

start
An integer that represents the line number at which to begin replacing. The position of the first line in *text-expression* is `1`. When you omit this argument, REPLLINES starts with line `1`. REPLLINES replaces as many lines of *text-expression* as are required for the specified new *lines.*

Examples

Example 20–27 Replacing a Text Line

This example shows how to replace the second line in a multiline TEXT value in a variable called `mktglist`.

The statement

```
SHOW mktglist
```

produces the following output.

```
Salespeople
Products
Services
```

The statement

```
SHOW REPLLINES(mktglist, 'advertising', 2)
```

produces the following output.

```
Salespeople
Advertising
Services
```

REPORT

The REPORT command quickly produces output for one or more data expressions. REPORT automatically loops over the dimensions of the data and formats the output. Using this default layout, you can produce an attractive report with a single short command. In addition, you can use REPORT command options to modify the default format and produce a custom report. Output from the REPORT command is sent to the current outfile.

To produce a default report, use this simple form.

```
REPORT expression . . .
```

You can also customize the layout that REPORT produces by using various options. REPORT has an underlying format similar to ROW, and all of the options available with the ROW command are available with REPORT.

Syntax

REPORT [NOHEAD] [GRANDTOTALS] [[SUBTOTALS] GROUP *dimension*] -

[[SUBTOTALS] [*attributes*] DOWN *dimension*] -

[[ROWTOTALS] ACROSS *dimension* [*limit-clause*]:] -

[SUBTOTALS] [*attributes*] *expression*(s)

Arguments

NOHEAD
Specifies that the report should contain no initial blank line and no headings. NOHEAD must be the first argument to the REPORT command. It overrides any HEADING arguments you specify in the command, as well as suppressing all headings that the REPORT command normally generates automatically. The NOHEAD keyword is useful for creating files that do not contain data or headings.

GRANDTOTALS
Includes a grand total for each numeric column at the end of your report. Unless you include NOHEAD, GRANDTOTALS must be the first argument to the REPORT command. When you include NOHEAD, GRANDTOTALS must be the second argument to the REPORT command.

SUBTOTALS
Includes subtotals for numeric columns. A row of dashes precedes each row of subtotals. You can get subtotals for a specific set of data by specifying the keyword SUBTOTALS before the GROUP or DOWN keyword or before a data expression.

```
SUBTOTALS GROUP dimension
SUBTOTALS DOWN dimension
SUBTOTALS expression
```

When you specify SUBTOTALS for an expression or DOWN phrase, you get subtotals for each GROUP dimension (or composite). When you specify SUBTOTALS for a GROUP phrase, you get subtotals for the specified dimension and for any slower-varying GROUP dimensions. The subtotals for a group appear at the bottom of the last slice in the group.

GROUP *dimension*
Produces a separate *group*, or two-dimensional slice, of the data for each value of *dimension.* You can use the GROUP keyword more than once to specify more than one GROUP dimension (or composite). In this case, you produce a separate slice for each combination of the values of the GROUP dimensions. Any GROUP phrases must be specified before any DOWN or ACROSS phrases.

In place of *dimension,* you can specify a text expression in order to provide formatted labels. The expression must be dimensioned only by the desired GROUP dimension, and each value of the expression should be descriptive text that corresponds to its associated dimension value. For information on providing formatted labels for a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR, see "Formatting DAY, WEEK, MONTH, QUARTER, and YEAR Dimension Values" on page 20-70.

DOWN *dimension*
Produces a column of dimension values labeling the rows down the left side of your report. The default width of the label column is controlled by the LCOLWIDTH option, which has a default value of 14 characters. When the DOWN phrase specifies a composite or a conjoint dimension, Oracle OLAP creates a separate column for each base dimension. The default width of the base dimension columns is controlled by the COLWIDTH option, which has a default value of `10` characters. You can override the default of any label column by using the WIDTH attribute in REPORT (see Table 20–5, " Format Attributes for Data Values" on page 20-62). You can have only one DOWN phrase. Any DOWN phrase must be specified before any ACROSS phrase.

When the DOWN dimension is a composite or a conjoint dimension, you can provide a different width for each base dimension column by using the KEY function. You can produce a label column for each base dimension with the KEY function and use a separate WIDTH attribute for each column. For example, assume that `proddist` is a composite with the base dimensions `product` and `district`. In this case, you can use a statement similar to the following one.

```
REPORT DOWN < W 8 KEY(proddist, product) -
   W 12 KEY(proddist, district) > . . .
```

In place of *dimension* you can specify a text expression in order to provide formatted labels. The expression must be dimensioned only by the desired DOWN dimension, and each value of the expression should be descriptive text that corresponds to its associated dimension value. You can also use two or more text expressions (each dimensioned only by the DOWN dimension) as the *dimension* argument by placing them in angle brackets.

REPORT DOWN < *expression1 expression2* > . . .

For information on providing formatted labels for a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR, see "Formatting DAY, WEEK, MONTH, QUARTER, and YEAR Dimension Values" on page 20-70. For information on suppressing a label column, see "Suppressing a Column".

ROWTOTALS
Includes a column headed "TOTAL" at the right side of the report with a total for each numeric row. You must specify ROWTOTALS before the ACROSS keyword. Including a row total in your report does not imply either column subtotals or a grand total. The keyword SUBTOTALS before an ACROSS phrase produces subtotals for any DOWN *and* GROUP dimensions that are specified.

ACROSS *dimension* [*limit-clause*]:
Produces a row of headings across the top of your report, one for each value in *dimension.* Under each heading, REPORT produces a column of data for the data expression you specify. You can have more than one ACROSS dimension (or composite) in the report, each followed by a colon.

You can nest different ACROSS dimensions for one data expression, as illustrated in the following statement.

```
ACROSS district: ACROSS product: units
```

In place of *dimension* you can specify a text expression in order to provide formatted labels. The expression must be dimensioned only by the desired ACROSS dimension, and each value of the expression should be descriptive text that corresponds to its associated dimension value. For information on providing formatted labels for a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR, see "Formatting DAY, WEEK, MONTH, QUARTER, and YEAR Dimension Values" on page 20-70.

When you specify a composite in the ACROSS phrase, you cannot include a *limit-clause* argument. You must limit the base dimensions of a composite to the desired values before you execute a REPORT command.

However, when you specify a dimension in the ACROSS phrase, *limit-clause* enables you to change the status of that dimension. The new status will be in effect only for the duration of the REPORT command. The format of *limit-clause* is as follows.

[ADD|COMPLEMENT|KEEP|REMOVE|INSERT|TO] *valuelist* [IFNONE *label*]

To specify the temporary status, insert any of the LIMIT keywords (the default is TO) along with an appropriate value list or related-dimension list. You can use any valid LIMIT clause in *valuelist* (see the entry for the LIMIT command for further information). The following example temporarily limits month to the last six values, no matter what the current status of month is.

```
ACROSS month LAST 6: units
```

When the limits you specify result in an empty status for the dimension, an error occurs (regardless of the setting of the OKNULLSTATUS option). However, when you include the phrase IFNONE *label*, the error is suppressed and execution of your program branches to the specified label, where you can handle the error.

attributes
One or more format attributes from Table 20–5, " Format Attributes for Data Values" on page 20-62 that specify how to format the data. A group of attributes can apply to one or more data expressions. See "Attributes for Formatting Data" on page 20-67.

The purpose of negative attributes (such as NOPAREN) is to override options or attributes that are more globally set. The negative attributes have an asterisk (*) before their names in Table 20–5, " Format Attributes for Data Values" on page 20-62.

***expression* . . .**
The data to be shown in the report. The way the data looks depends on its data type and the attributes you specify. You can specify more than one expression; the expressions do not have to have the same dimensions.

When you specify only one data expression, REPORT produces one data column for each heading for the ACROSS dimension. However, when you want REPORT to show two or more expressions under each heading, enclose the expressions in angle brackets (< >).

```
ACROSS dimension: <expression1, expression2>
```

Table 20–5 Format Attributes for Data Values

Attribute	Meaning
HEADING *'text'*	Specifies text to use in place of default column headings.
WIDTH *n* (W *n*)	Makes the column *n* spaces wide. The default width for the first column is the value of the LCOLWIDTH option. For other columns, it is the value of the COLWIDTH option. The maximum width is 4000 characters. Columns with a width of 0 (zero) are suppressed.
SPACE *n* (SP *n*)	Precedes the column with *n* spaces. The default for the first column is `0`; for other columns, `1`.
INDENT *n*	Indents the value *n* spaces within its column. The default is 0.
LEFT (L)	Left-justifies the value within its column. This is the default for TEXT data.
RIGHT (R)	Right-justifies the value within its column. This is the default for numeric and Boolean data.
CENTER (C)	Centers the value within its column.
LSET *'text'*	Adds *text* to the left of the value.
*NOLSET	Does not add anything to the left of the value. (Default
RSET *'text'*	Adds *text* to the right of the value.
*NORSET	Does not add anything to the right of the value. (Default)
FILL *'char'*	Puts *char* into unused positions in the column. The default fill character is a space.
DECIMAL *n* (D *n*)	Shows *n* decimal places. Decimal places are separated by the character currently recorded in the DECIMALCHAR option. The default number of decimal places is controlled by the DECIMALS option.
*NODECIMAL	Shows the number of decimal places indicated by the DECIMALS option. (Default)

Table 20–5 (Cont.) Format Attributes for Data Values

Attribute	**Meaning**
COMMA	Marks thousands and millions with commas or the character currently recorded in the THOUSANDSCHAR option. The default is controlled by the COMMAS option.
*NOCOMMA	Does not mark thousands and millions.
PAREN	Uses parentheses to indicate negative numbers. The default is controlled by the PARENS option.
*NOPAREN	Uses the minus sign to indicate negative numbers. The default is controlled by the PARENS option.
LEADINGZERO	Puts a leading zero before decimal numbers between -1 and 1.
*NOLEADINGZERO	Suppresses leading zeros before decimal numbers between -1 and 1.
CNLEADINGZERO	Puts a leading zero before decimal numbers between -1 and 1 when it does not cut off any significant digits.
MNOTATION	Always uses M-notation (divides values by one million and appends "M").
CMNOTATION	Conditionally uses M-notation, when needed to make a value fit in a column.
*NOMNOTATION	Does not use M-notation (uses asterisks for oversize values).
MDECIMAL *n*	Shows *n* decimal places in numbers formatted with M-notation; *n* can be any number from 0 to 16, or 255.
ENOTATION	Always uses scientific notation, also called exponential notation or E-notation (appends "E", and includes a sign before the exponent, for example, `.230E+2` or `.230E-2`).
CENOTATION	Conditionally uses E-notation, when needed to make a value fit in a column.
*NOENOTATION	Does not use E-notation (defaults to conditional M-notation).
EDECIMAL *n*	Shows *n* decimal places in numbers formatted with E-notation; *n* can be any number from 0 to 16, or 255.
NASPELL '*text*'	Uses *text* in place of `NA` values. The default is controlled by the NASPELL option.
*NONASPELL	Spells `NA` values as indicated by the NASPELL option.
ZSPELL '*text*'	Uses *text* in place of zero numeric values. The default is controlled by the ZSPELL option.
*NOZSPELL	Spells zero values as indicated by the ZSPELL option.

Table 20–5 (Cont.) Format Attributes for Data Values

Attribute	**Meaning**
YESSPELL *'text'*	Text used for `TRUE` Boolean values. The default is recorded in the YESSPELL option.
NOSPELL *'text'*	Text used for `FALSE` Boolean values. The default is recorded in the NOSPELL option.
TRUNCATE (TRUNC)	Truncates a character value to the column width when it does not fit in the column.
*NOTRUNCATE (NOTRUNC)	Creates additional lines when the character value does not fit in the column.
FOLDUP	For a multiline character value, places all but the last line above the rest of the row, and the last line on the row with the other values; also strips any leading or trailing spaces.
FOLDDOWN	For a multiline character value, places the first line on the row with the other values, and places additional lines below the rest of the row; also strips any leading or trailing spaces.
VALONLY	Underlines or overlines the value only. (Used with UNDER and OVER.)
NOVALONLY	Underlines or overlines the entire width of the column. (Used with UNDER and OVER.)
UNDER *textexp*	Underlines the value with the value of a character expression (*textexp*). When *textexp* is a literal value, it must be enclosed in single quotes. Useful literal values include: `'-'` to underline value or column, `'='` to double underline value or column, and `''` to indicate that a value or column is not underlined. To underline only when a condition is met, for *textexp* use: `IF` `*boolean-expression*` `THEN '-' ELSE ''`
OVER *textexp*	Overlines the value with the value of a character expression (*textexp*). When *textexp* is a literal value, it must be enclosed in single quotes. Useful literal values include: `'-'` to overline value or column, `'='` to double overline value or column, and `''` to indicate that a value or column is does not have an overline. To overline only when a condition is met, for *textexp* use: `IF` `*boolean-expression*` `THEN '-' ELSE ''`

Notes

Report Options

A number of options effect reports created using the OLAP DML. These options are listed in Table 20–6, " Report Options".

Table 20–6 Report Options

Statement	Description
BMARGIN	An option that specifies the number of blank lines for the bottom margin of output pages.
COLWIDTH	An option that controls the default width of data columns in report output.
COMMAS	An option that controls the use of a character to separate thousands and millions in numeric output.
DECIMALCHAR	(Read-only) An option that records the character that is used as the decimal marker in output.
DECIMALS	An option that controls the number of decimal places that are shown in numeric output.
LCOLWIDTH	An option that controls the default width of the label column in reports.
LINENUM	An option that contains the current line number of the output.
LINESLEFT	(Read-only) An option that contains the number of lines left on the current page.
LSIZE	An option that specifies the line size within which the STDHDR program centers the standard header.
NASPELL	An option that controls the spelling that is used for NA values in output.
NOSPELL	(Read-only) An option that contains the text that is used for FALSE Boolean values in the output of OLAP DML statements.
PAGENUM	An option that contains the current page number of output.
PAGEPRG	An option that contains the name of a program or the text of a statement to be executed at the beginning of each page of output.
PAGESIZE	An option that contains the size of a page of output.
PAGING	An option that controls the production of paged output in Oracle OLAP.

Table 20–6 (Cont.) Report Options

Statement	Description
PARENS	An option that controls whether negative numbers are represented in output with parentheses or a minus sign.
THOUSANDSCHAR	(Read-only) An option that contains the character that is used as the thousands group marker in output.
TMARGIN	An option that defines the number of blank lines for the top margin of output pages, above the running page heading when PAGING is set to YES.
YESSPELL	(Read-only) An option that specifies the text that is used for TRUE Boolean values in the output of OLAP DML statements.
ZEROROW	An option that controls suppresses report rows with numeric values that are all NAs or all zeros or would be represented as zeros.
ZSPELL	An option that specifies the default text that is used for representing numeric zero values in output produced by the HEADING, REPORT, and ROW commands.

Default Layout

In determining the layout of its output, REPORT follows any layout keywords (GROUP, DOWN, or ACROSS) that you specify in the statement. By default, REPORT tries to format its output compactly. Normally, this means a two-dimensional report of the data with one of the dimensions down the side and the other across the top, much like a spreadsheet. Any additional dimensions of the data form "slices" or separate two-dimensional segments, like a series of spreadsheets.

When no layout keywords are specified, REPORT uses the following rules to determine the layout:

- The fastest-varying dimension in an expression (the one that appears first in the definition of that expression) goes across, the next fastest goes down, and any remaining dimensions become GROUP slices.
- The order of dimensions in a list of two or more expressions is a simple combination of the dimensions that appear in the definitions of the component expressions. The original order is preserved as far as possible, subject to the rule that repeated mentions of the same dimension are dropped. For example, the dimensions of the combined variables `price` and `industry.sales`, where `price` has the dimensions `<month product>` and `industry.sales` has the

dimensions `<quarter product region>`, are
`<month product quarter region>`.

When you produce a report of data for a variable dimensioned by a composite, REPORT automatically breaks out the data by the base dimensions of the composite that is used in the definition of the variable. When a particular combination of base dimension values does not exist in the composite, the report shows `NA` for the corresponding data cell. See Example 20–32, "Reporting Data Dimensioned by Composites" on page 20-74.

Layout Keywords

The layout keywords, GROUP, DOWN, and ACROSS, allow you to alter the default layout by changing the way in which the data's dimensions are arranged down and across the report.

When you specify some but not all of the dimensions of an expression in GROUP, DOWN, or ACROSS phrases, REPORT follows the default layout as closely as possible with the unspecified dimensions. (See "Default Layout" on page 20-66.) When you want a different layout, the easiest way to get it is to specify exactly what you want with the GROUP, DOWN, and ACROSS keywords.

When one of the dimensions has just one value in the status, you should specify it in a GROUP phrase for a more pleasing layout.

When you specify a composite in a GROUP, DOWN, or ACROSS phrase, you can override the default format of REPORT and break out the data by its composite. In this case, when a particular combination of base dimension values does not exist in the composite, the report does not include that combination. See Example 20–32, "Reporting Data Dimensioned by Composites" on page 20-74.

The dimensions that you specify in GROUP, DOWN, and ACROSS phrases are not required to be relevant to the data they loop over. See "Specifying Extra Dimensions" on page 20-70.

Unnamed Composites

You can specify an unnamed composite as the *dimension* argument by using the syntax that was used to create the unnamed composite.

Attributes for Formatting Data

You can use attributes to change the way the data is formatted. Attributes can apply to one data expression or a group of expressions, depending on where you specify them.

- When you do not specify any attributes, the default format is used for the data values in a report. Oracle OLAP automatically determines the width of the columns, the number of decimal places, whether commas are used to mark thousands in numeric values, and so on. However, by including format attributes prior to the data expression in your REPORT command, you can change the way in which the values are formatted.

  attributes expression

- When you have several data expressions in your REPORT command, you can specify different format attributes before each. When you want attributes to apply to two or more data expressions, enclose the expressions in angle brackets (< >).

  attributes <expression1, *expression2>*

- Attributes outside the brackets apply to all the expressions within the brackets. However, you can also specify attributes for only one of the expressions (even an attribute that contradicts one that applies to the group) within the brackets by including them immediately before the expression.

  attributes0 <attributes1 expression1, *expression2>*

  In this case, *attributes0* applies to both *expression1* and *expression2*; while *attributes1* only applies to *expression1.*

- You can specify attributes before an ACROSS phrase. Those attributes apply to the data values within the scope of that particular ACROSS phrase.

  attribs1 ACROSS *dimen1*: [*attribs2* ACROSS *dimen2*:]

- You can format the data in the labels column by specifying attributes before the DOWN phrase.

  attributes DOWN *dimension*

Table 20–5 shows attributes you can use to format values. Some attributes have a corresponding option (indicated in Table 20–5) that you can use to control the default.

Using Properties for Attributes

When a variable has a formatting property attached to its definition, you can use the OBJ function to obtain the value of that property and use it as the value of an attribute in the REPORT command.

HEADING Attribute

When you use the HEADING attribute, the position of the heading you specify will vary depending on how many expressions it must span in your report. This means that your heading may or may not replace a default heading.

When you use the HEADING attribute to specify a column title that is wider than the column width, the text of the title will wrap within the width of its column.

Suppressing All-Zero Rows

The ZEROROW option controls whether rows of all-zero data are included in a report.

INTEGER Data

The REPORT command suppresses decimal places in row and column totals of INTEGER data unless you specify the DECIMAL attribute for the totaled expression.

Decimal Values Between -1 and 1

When you set the DECIMAL attribute to `0` and you use the NOLEADINGZERO keyword, any decimal values between `-1` and `1` that are rounded to `0` will not be shown.

Maximum Line Width

The maximum width of a line in a report is 4000 characters.

Width of Columns

For data columns (ACROSS), the default width is the value of the COLWIDTH option (default is 10). The default width for the label column (DOWN) is the value of the LCOLWIDTH option (default is `14`). This default is used when you omit the DOWN phrase or when you specify a simple dimension in the DOWN phrase. When the DOWN phrase specifies a conjoint dimension or a composite, the default label width is the width of the COLWIDTH option (default is `10`) and there is a separate column for each base dimension.

You can specify widths for specific columns by using the WIDTH attribute (abbreviated W). For a composite or conjoint dimension, the WIDTH attribute applies to *each* base dimension column. The total label width is the number of base dimensions multiplied by the width you specify (plus one for the space between the columns).

When you use the default line width of 80 characters (determined by the LSIZE option) and the default column width settings (with a single label column of 14 characters) a line of output can accommodate the labels column and six data columns. The combined width of all the columns of a report cannot be greater than 4000 characters.

When a numeric value is too large to fit into a data column, REPORT rounds it off to the nearest million with the symbol M at the right side of the cell. When a value is still too large, REPORT replaces the value with a series of asterisks.

Suppressing a Column

You can suppress a column by specifying a column width of `0`.

LSET or RSET with NA Values

When you use the LSET or RSET attribute with an expression that contains `NA` values, the text you specify with LSET or RSET will not be included to the left or right of any `NA` values.

Formatting DAY, WEEK, MONTH, QUARTER, and YEAR Dimension Values

When you use a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR as the *dimension* in an ACROSS, DOWN, or GROUP phrase, you can use the CONVERT function to override the dimension's VNF (or the default VNF) and provide your own format for the dimension value names. To override the VNF, use the CONVERT function with a *vnf* argument in place of the *dimension* argument to the ACROSS, DOWN, or GROUP keyword. For example, in a report of `units` data, you can format the labels for the `month` dimension by using the following statement.

```
REPORT HEADING 'Month' DOWN -
   CONVERT(month TEXT '<mtextl> <yyyy>') units
```

Specifying Extra Dimensions

The REPORT command uses whatever dimensions you specify in laying out the report, regardless of whether the expressions to be shown are dimensioned by these dimensions. When an expression is not dimensioned by one or more of the dimensions specified, the values of that expression are repeated for each value of the extra dimension. This fact is sometimes useful for comparisons. SeeExample 20–31, "Repeating Price Data" on page 20-73.

Creating Running Totals

You can use the RUNTOTAL function within a REPORT command to create running totals.

Decimal Overflow

When a "decimal overflow" condition occurs while subtotals are being accumulated (that is, an out-of-range value is generated), all subtotals for the affected column are set to `NA` and processing continues when the DECIMALOVERFLOW option is set to `YES`. When DECIMALOVERFLOW is set to `NO`, an error occurs when a decimal overflow condition occurs.

Performance Tip for Reporting Variables Dimensioned by Composites

By default, when REPORT explicitly loops over a composite, it sorts the composite values according to the current order of the values in the composite's base dimensions. The task of sorting requires some processing time, so when variables are large, performance can be affected. When your variable is very large, and you are more concerned about performance than about the order in which REPORT output is produced, you can set the SORTCOMPOSITE option to `NO`.

NTEXT Values

The REPORT command is not equipped to deal with NTEXT values. Do not include them in any part of a report.

Examples

Example 20–28 Creating a Default Report

This example shows how to look at product prices for the first three months of 1996. You can use REPORT in its simplest form, without changing the default layout

```
LIMIT month TO 'Jan96' TO 'Mar96'
REPORT price
```

These statements produce the following output.

```
                --------------PRICE-------------
                --------------MONTH-------------
PRODUCT           Jan96       Feb96      Mar96
--------------- ---------- ---------- ----------
Tents               165.50     165.75     165.13
Canoes              200.25     200.09     200.05
Racquets             55.02      55.03      55.00
Sportswear           50.03      50.02      50.00
Footwear             38.01      38.01      38.01
```

Example 20–29 Including Column Totals

This example looks at unit sales for three districts for the first half of 1996, with `district` across the report and a subtotal for each column. (By default, months would be arranged across the report, since `month` is the fastest-varying dimension of `units`.) To make the report more compact, specify a smaller column width of `8` characters.

```
LIMIT month TO 'Jan96' TO 'Jun96'
LIMIT district TO 'Boston' 'Chicago' 'Dallas'
REPORT SUBTOTALS W 8 DOWN month -
   ACROSS district: W 8 units
```

These statements produce the following output.

```
PRODUCT: TENTS
         ----------UNITS-----------
         ---------DISTRICT---------
MONTH    Boston   Chicago  Dallas
-------- -------- -------- --------
Jan96         307      189      308
Feb96         209      190      324
Mar96         277      257      436
Apr96         372      318      560
May96         525      433      744
Jun96         576      466      838
-------- -------- -------- --------
TOTAL       2,266    1,853    3,210
   ...
```

REPORT produces a similar slice for each product.

Example 20–30 Comparing Two Variables

This example compares actual sportswear sales with the projected sales plan, looking only at whole-dollar figures. It reports the actual and planned values side-by-side for May and June, 1996, and provides a grand total of sales and planned sales for all districts.

```
LIMIT product TO 'Sportswear'
LIMIT month TO 'May96' 'Jun96'
LIMIT district TO ALL
REPORT GRANDTOTALS W 12 DOWN district ACROSS month: -
   DECIMAL 0 <sales sales.plan>
```

These statements produce the following output.

```
PRODUCT: SPORTSWEAR
             -------------------MONTH-------------------
             --------May96-------- --------Jun96--------
DISTRICT        SALES    SALES.PLAN   SALES    SALES.PLAN
------------ ---------- ---------- ---------- ----------
Boston           72,617     69,623     79,630     73,569
Atlanta         161,537    148,823    177,967    157,939
Chicago         101,873     94,545    112,793     97,427
Dallas          170,939    165,449    175,066    164,192
Denver           89,971     91,880     97,237     94,729
Seattle          57,713     55,905     60,323     56,808
             ---------- ---------- ---------- ----------
                654,651    626,224    703,017    644,664
             ========== ========== ========== ==========
```

Example 20–31 Repeating Price Data

This example compares sales across three districts, and it includes the unit price beside each sales figure for close comparison within each district. The REPORT command specifies two expressions, `sales` and `price`. Since `sales` has three dimensions, `month`, `product`, and `district`, the report shows these three dimensions. However, `price` is not dimensioned by `district`. Therefore, the report repeats the values of `price` for each `district`. The report for January 1995 shown.

```
LIMIT district TO FIRST 3
LIMIT product TO ALL
LIMIT month TO 'Jan95'
REPORT GROUP month W 10 DOWN product ACROSS district: -
   <W 9 sales W 6 price>
```

These statements produce the following output.

```
MONTH: Jan95
           -------------------DISTRICT------------------------
           -----Boston-----  ----Atlanta----- -----Chicago----
PRODUCT      SALES    PRICE   SALES    PRICE   SALES    PRICE
---------- --------- ------  --------- ------ --------- ------
Tents      32,153.52 160.77  40,674.20 160.77 29,098.94 160.77
Canoes     66,013.92 190.24  49,462.88 190.24 45,277.56 190.24
Racquets   52,420.86  52.84  54,798.82  52.84 54,270.39  52.84
Sportswear 53,194.70  48.54 114,446.26  48.54 72,123.47  48.54
Footwear   91,406.82  36.10 100,540.28  36.10 90,287.70  36.10
```

Example 20–32 Reporting Data Dimensioned by Composites

In this example, `d.sales` is a variable whose dimension list includes the dimension `month` and the unnamed composite `SPARSE <product district>`. The unnamed composite contains no values for the base dimension combinations for the `Boston` and `Chicago` districts and the `Tents`, `Racquets`, And `Footwear` products. When you use the default form of the REPORT command to produce a report of `d.sales` data, REPORT breaks out the report by `month` and by the base dimensions of the unnamed composite (`product` and `district`). For the combinations of base dimension values that do not exist in the composite, the report shows `NA` for the corresponding data cells.

```
LIMIT month TO 'Jan96' TO 'Mar96'
LIMIT district TO 'Boston' 'Chicago'
REPORT d.sales
```

These statements produce the following output.

```
DISTRICT: Boston
               ------------D.SALES-------------
               -------------MONTH--------------
PRODUCT          Jan96      Feb96      Mar96
-------------- ---------- ---------- ----------
Tents                  NA         NA         NA
Canoes             70,489     82,238     97,622
Racquets               NA         NA         NA
Sportswear         57,079     63,122     67,006
Footwear               NA         NA         NA

DISTRICT: Chicago
               ------------D.SALES-------------
               -------------MONTH--------------
PRODUCT          Jan96      Feb96      Mar96
-------------- ---------- ---------- ----------
Tents                  NA         NA         NA
Canoes             48,662     54,425     68,816
Racquets               NA         NA         NA
Sportswear         77,490     85,879     85,308
Footwear               NA         NA         NA
```

By specifying the composite in an ACROSS, DOWN, or GROUP phrase, you can override the default format of REPORT and break out the `d.sales` data by its composite. In this case, the report only includes the data cells for which the composite contains values.

```
REPORT DOWN SPARSE <product district> d.sales
```

This statement produces the following report.

```
                      ------------D.SALES-------------
                      -------------MONTH--------------
 PRODUCT    DISTRICT    Jan96      Feb96      Mar96
---------- ---------- ---------- ---------- ----------
Canoes     Boston         70,489     82,238     97,622
Sportswear Boston         57,079     63,122     67,006
Canoes     Chicago        48,662     54,425     68,816
Sportswear Chicago        77,490     85,879     85,308
```

Example 20–33   Reporting Values of Composites

To make it easy to examine the values that exist in a named or unnamed composite, REPORT loops in a special way when you report the values of the composite.

Instead of looping over all the values in the status for the base dimensions of the composite, REPORT observes the status of the base dimensions, but loops over the combinations of the base dimension values that exist in the composite. To see all the values of the `SPARSE<product district>` composite, you can use the following statements.

```
ALLSTAT
REPORT SPARSE <product district>
```

These statements produce a report of all the combinations of `product` and `district` that exist in `SPARSE<product district>`.

21
RESERVED to SPARSEINDEX

This chapter contains the following OLAP DML statements:

RESERVED

The RESERVED function can provide a list of all the words that are reserved because they are known to the OLAP DML parser, or it can indicate whether or not a word that you specify is known to the OLAP DML parser. Some other words are also reserved as discussed in "Other Reserved Words" on page 21-3.

Return Value

Either a multiline text expression or BOOLEAN, depending on whether or not you specify an argument to the function.

Syntax

RESERVED [(*word-expression*)]

Arguments

word-expression
A text expression that represents a word that may or may not be reserved in the OLAP DML. When you specify *word-expression,* the RESERVED function returns a BOOLEAN value indicating whether or not the word is reserved in OLAP DML. When you do not specify an argument, RESERVED returns a TEXT value consisting of all the reserved words in OLAP DML, with each word on a separate line.

Notes

Other Reserved Words

The RESERVED function only recognizes words known to the OLAP DML parser. This does not include the names of option objects and some other objects in the `EXPRESS` analytic workspace. The names of these objects are reserved in Oracle OLAP, but are ignored by the RESERVED function. To identify the names of these objects, issue the following statements.

```
AW ATTACH EXPRESS
LISTNAMES
```

NA is Reserved

When you specify `NA` for the argument, the RESERVED function returns `NO`. When you specify `NA`, the RESERVED function returns `YES`.

Case-Sensitivity

The list of reserved words returned by the RESERVED function contains some words in all uppercase and some in mixed case. Words all in uppercase are reserved in their entirety. Words in mixed case can be abbreviated to the uppercase portion. For such words, any subset of the word containing the uppercase portion is reserved. For example, one of the words in the list returned by RESERVED is `CODEVERsion`. The following are all reserved: `codever`, `codeversi`, `codeversio`, and `codeversion`. However, `codeve` is not reserved.

Examples

Example 21–1 Determining If a Word Is Reserved

The following example shows how you can use the RESERVED function to determine if a word is reserved in OLAP DML.

The function call

```
SHOW RESERVED('update')
```

returns the following value

```
YES
```

RESYNC

When an analytic workspace is attached in multiwriter mode, the RESYNC command drops private changes for the specified variables, relations, valuesets, and dimensions and promotes them so that the user now sees the data from the latest visible generations. You can only resychronize read-only objects. Oracle OLAP ignores any requests to resychronized acquired objects.

Syntax

RESYNC [*objects*] [*analytic_workspaces*]

Arguments

When no parameters are specified, all read-only objects in the current analytic workspace are resychronized.

objects
A list of one or more variables, relations, valuesets, or dimension names, separated by commas, that you want to resynchronize.

analytic_workspaces
A list of names, separated by commas, of one or more analytic workspaces presently attached in multiwriter mode. Oracle OLAP resynchronizes all read-only variables, relations, valuesets, and dimensions in a listed analytic workspace at the same time.

Notes

Keeping Logical Relationship of Objects

When using RESYNC keep in mind the logical relationship of different objects to avoid losing the logical consistency of the data by promoting some objects, but not others to a new generation.

Resynchronizing Objects that Share a Composite Dimension

All objects that share a composite dimension must be resynchronized together.

Examples

Example 21–2 Resynchronizing Objects

In this example, user A is periodically updating `actuals`, while user R needs to periodically check the latest view of the data. They could execute the following OLAP DML statements.

User A could execute the following OLAP DML statements.

```
AW ATTACH myworkspace MULTI
ACQUIRE actuals
...make modifications
UPDATE MULTI actuals
COMMIT
...make modification
UPDATE MULTI actuals
COMMIT
```

At the same time, user R could execute the following OLAP DML statements.

```
AW ATTACH myworkspace MULTI
...
RESYNC actuals
...
RESYNC actuals
...
RESYNC actuals
...
```

RETURN

Within an OLAP DML program, the RETURN command terminates execution of a program prior to its last line. You can optionally specify a value that the program will return. The value should have the same data type or dimension that you specified when you defined the program.

Syntax

RETURN [*expression*]

Arguments

expression
The expression to be returned to the calling program when the called program terminates.

Notes

User-Defined Function

To return a value, a program must be called as a function. That is, it must be used as an expression in a statement. The following is an example of a user-defined function being used as an argument to the REPORT command.

```
REPORT ISRECENT(actual)
```

When a program is called with a CALL command or by using the standalone command format, its return value is discarded.

Return Value Dimensionality

The value returned by a program is a single value, without any dimensions. However, within the context of the command that calls a user-defined function, the function expression has the dimensions of its arguments. For instance, in "User-Defined Function" on page 21-7, when `actual` is dimensioned by `line`, `division` and `month`, then the expression `ISRECENT(actual)` is also dimensioned by `line`, `division` and `month`. Therefore, Oracle OLAP will call the ISRECENT program once for every combination of `line`, `division` and `month` in the current status.

Return Value Data Type

When you specify a data type when you define a program, the return value will have that data type. When you specify a dimension when you define a program, the return value will be a single value in that dimension. When the expression in the RETURN command does not match the declared data type or dimension, Oracle OLAP will convert it to the declared data type.

When you do not specify a data type or dimension in the definition of a program, its return value is treated as worksheet data. This means Oracle OLAP will convert any return value to the data type that is required by the calling context. This may lead to unexpected results.

Dimension Location

When the program returns values of a dimension, the dimension must be declared in the same analytic workspace as the program. The program will be in the output of the LISTBY program, and OBJ(ISBY) will be `TRUE` for the dimension.

No Return Value

When a program has been invoked as a function, but it does not provide a return value, the value that is returned to the calling program is `NA`.

User-defined Functions

For more information about user-defined functions, see the entries for the ARGUMENT, CALL, and DEFINE PROGRAMM commands.

Examples

Example 21–3 Terminating a Program Early

In this example, suppose you want a report program that will produce a report only when a variable called `newfigures` is present in the current analytic workspace. In your program, you can use an IF command to check whether `newfigures` exists and a RETURN to stop execution when it does not.

```
DEFINE sales.report PROGRAM
PROGRAM
IF NOT EXISTS('newfigures')
   THEN DO
          SHOW 'The new data is not yet available.'
          RETURN
        DOEND
PUSH month
TRAP ON cleanup
LIMIT month TO LAST 3
REPORT ACROSS month: newfigures

cleanup:
POP month
END
```

Now when you run the program without `newfigures` in the analytic workspace, the program produces a message and the RETURN command terminates execution of the program at that point.

Example 21–4 Returning a Value

The following program derives next year's budget figures from the `actual` variable. It is a temporary calculation. You could call this program in a REPORT command, thus calculating and reporting the budget figures without storing them in an analytic workspace.

```
DEFINE budget.growth PROGRAM DECIMAL
PROGRAM
VARIABLE growth DECIMAL
VARIABLE factor DECIMAL
growth = TOTAL(actual(year 'Yr97') year) - TOTAL(actual(year -
   'Yr96') year)
factor = ( 1 + growth ) / TOTAL(actual(year 'Yr96') year)
RETURN TOTAL(actual(year 'Yr97') year) * (factor * factor/2)
END
```

REVERT

The REVERT command drops all changes made to the specified objects since they were last updated, resynchronized (using the RESYNC command), or acquired using ACQUIRE the RESYNC phrase, or since the analytic workspace was attached.

Syntax

REVERT [*objects*] [*analytic_workspaces*]

Arguments

When you do not specify any parameters, all objects in the current analytic workspace are reverted.

objects
A list of the names, separated by commas, of acquired variables, valuesets, relations, or dimensions in an analytic workspace attached in multiwriter mode or a list of variables, valuesets, relations, or dimensions in an analytic workspace attached in read-only mode.

analytic workspaces
A list of names, separated by commas, of analytic workspaces attached in either multiwriter or read-only mode. When you specify the name of an analytic workspace attached in multiwriter mode, all acquired variables, valuesets, relations, and dimensions in that workspace are reverted. When you specify the name of an analytic workspace attached in read-only mode, all variables, valuesets, relations, and dimensions in that workspace are reverted. Additionally, regardless of the attachment mode (multiwriter or read-only), all temporary variables and dimensions are emptied, all session-temporary objects are deleted, and all workspace-specific status and environment is reset.

Notes

Reverting a Dimension After Adding Dimension Values

Reverting a dimension after adding dimension values is not recommended since it can result in suboptimal space allocation for variables dimensioned by the dimension.

Examples

Example 21–5 Using REVERT to Undo Modifications

Assume that you have a variable named `budget` in an analytic workspace named `myworkspace`. Assume, also, that you need to modify `budget` in several steps but do not want to update the analytic workspace data until all steps are completed. For each step, you want to run several models to find the one that produces desired results. To perform this task, take the following steps:

1. Attach the analytic workspace in multiwriter mode.
2. Acquire `budget`.
3. For each step:
   a. Run the appropriate models, performing revert operations between them until you finds the desired model
   b. Update `budget`.
4. Commit and release `budget`.

The following code accomplishes these tasks.

```
AW ATTACH myworkspace MULTI
ACQUIRE RESYNC budget
...try model 1a --> not acceptable
REVERT budget
...try model 1b --> ok. Done with Step 1
UPDATE MULTI budget
...try model 2a --> not acceptable
REVERT budget
...try model 2b --> not acceptable
REVERT budget
...try model 2c --> ok. Done with Step 2
UPDATE MULTI budget
...try model 3a --> ok. Done with Step 3. Done with all steps.
UPDATE MULTI budget
COMMIT
RELEASE budget
AW DETACH myworkspace
```

ROLE

(Read-only) The ROLE option holds a list of Oracle Database roles associated with the user ID under which an Oracle OLAP session is running.

Data type

TEXT

Syntax

ROLE

Examples

Example 21–6 Displaying a List of Groups or Roles

This statement displays a list of the roles associated with the current session user ID.

```
SHOW ROLE
```

ROLLUP

The ROLLUP command calculates totals for a hierarchy of values where each level of the hierarchy is an aggregation of the values in the level below it.

The ROLLUP command only performs simple sum aggregation. Additionally, it only aggregates data when the members of the hierarchy are contained in a single rollup or "embedded-total" dimension, so called because it contains both a detail (lowest) level and levels that are aggregations of lower levels. A relation between the embedded-total dimension and itself, called a "parent relation," specifies the arrangement of the hierarchy. For each value of the dimension, the parent relation contains the value that is immediately above it in the hierarchy (that is, its "parent" value).

Before using ROLLUP, make sure that the data variable that is dimensioned by the embedded-total dimension has data for the lowest-level values in the hierarchy. ROLLUP uses the data at the lowest level to calculate the totals for the higher levels.

Note: Most applications aggregate data using an aggmap object rather than using the ROLLUP command. Aggmap objects allow you to write complex aggregation specifications. See "Aggregations" on page 4-2 for more information.

Syntax

ROLLUP *data* [OVER *embed-tot-dim*] [USING *parent-rel*] [ZEROFILL]

Arguments

data
A numeric variable whose values are to be rolled up. When the variable has more than one dimension, one of them must be the OVER dimension.

OVER *embed-tot-dim*
A dimension of *data* whose values form a hierarchy. When *data* has only one dimension, then that dimension is the OVER dimension by default and you can omit the OVER phrase.

USING *parent-rel*

A relation between the OVER dimension and itself, called a parent relation, that specifies a hierarchy among the dimension values. For each dimension value, this parent relation specifies another value of the dimension which is its immediate parent. The parent relation holds NAs for the values at the highest level of the hierarchy. When there is more than one relation between the OVER dimension and itself, then you must specify the relation you want to use as the parent relation.

ZEROFILL

Specifies that parent totals should be set to zero when all of their child values are `NA`. When you do not specify ZEROFILL, ROLLUP sets parent totals to `NA` when all of their child values are `NA`.

Notes

Generation Levels in a Parent Relation

In the hierarchy specified by the parent relation, you can think of the lowest level as the "child" values, all the other values as "parents," and each level as a "generation." The relation specifies the parent at the next higher level for each dimension value. The following example shows the values of an embedded-total rollup dimension called `area`, that has three levels, and the values of the child-parent relation `area.area`.

```
             AREA        AREA.AREA
             ----------  ----------
Level 1 ->   Totalus     NA
      2 ->   East        Totalus
      3 ->   Boston      East
      3 ->   Newyork     East
      2 ->   South       Totalus
      3 ->   Atlanta     South
```

A hierarchy can consist of several trees, so that there is more than one value at level 1. The value of the relation will be `NA` for all level-1 values, because these values have no parent in the hierarchy.

Including Child Values in the Status List

ROLLUP always looks to the lowest level of a hierarchy to calculate results. It rolls up *only* from the child values that are in the status list for the rollup dimension, but it still rolls up through all the levels in the hierarchy.

For example, suppose you use the area dimension and the area.area child-parent relation described in "Generation Levels in a Parent Relation" on page 21-14, and you change the data value for NewYork. When you then roll up with only the child values for East in the status list (Boston and NewYork), the rollup occurs without including the child value for South (Atlanta), but still includes level 2 as it goes from level 3 to level 1 (TotalUS). When you want *all* the child values included in rolling up to TotalUS, you must explicitly include all of them in the status list. In the example, you would limit area and add Atlanta to the status list.

Rolling Up from Changed Child Values

When the data has changed for some, but not all, of the child values in the embedded-total rollup dimension, you can set the status to roll up just the values that have changed. For example, assume your embedded-total dimension is called d2, and its parent relation is called reld2, first limit d2 to the values that have changed. Then use the following statements to add the appropriate additional values to the status list.

```
LIMIT d2 ADD ANCESTORS USING reld2
LIMIT d2 ADD CHILDREN USING reld2
```

Non-Rollup Dimensions

When the data variable being rolled up has more than one dimension, then the dimensions other than the rollup dimension are treated "normally." ROLLUP loops over their status and repeats the aggregation for each of their values.

Generation-Skipping Hierarchies

ROLLUP automatically distinguishes between generations in the parent relation, even to the extent of allowing *generation-skipping* hierarchies. For example, you can have a four-level hierarchy (for example, neighborhoods, cities, states, and total U.S.) that has a three-level branch (for example, Boston, Massachusetts, and total U.S.).

Status of the Rollup Dimension

Because ROLLUP automatically distinguishes parent values from child values, you can have all the values of the embedded-total rollup dimension in the status list when you execute ROLLUP.

Composites

When a variable includes a composite in its dimension list, you cannot roll up the data over the composite. However, you can roll up data over a base dimension of the composite. You specify which base dimension to loop over with the OVER keyword. The parent relation must be a relation between the base dimension and itself. When the composite is missing a value that is required for the rollup, ROLLUP will create the missing value.

Converting a Composite to a Conjoint Dimension

When your data is sparse, it is usually better to define a variable with a composite than with a conjoint dimension. However, in some situations you might find it advantageous to convert the composite to a conjoint dimension before you execute a ROLLUP command. When a variable is dimensioned by a conjoint dimension, you can define a parent relation between the conjoint dimension and itself and roll up the data over the conjoint dimension.

To convert a composite to a conjoint dimension, use the CHGDFN command.

Conjoint Dimensions

When a variable is dimensioned by a conjoint dimension, you can roll up the data over the conjoint dimension, but you cannot roll up over a base dimension of the conjoint. The parent relation must be a relation between the conjoint dimension and itself.

Multidimensional Parent Relations

You can have a multidimensional parent relation that defines more than one hierarchy, so that child values contribute to more than one higher-level total. However, at each level, the hierarchies should each point to a separate higher-level total, so that the data is not counted more than once at the higher level. See Chapter 21–8, "Using a Multidimensional Relation" on page 21-19.

Note: A multidimensional parent relation cannot share any dimensions with the data variable other than the embedded-total dimension.

Avoiding Circular Hierarchies

The parent relation must not create a *circular* hierarchy. That is, the relation must not contain any dimension values that are their own parent, either directly or indirectly. A parent relation that creates a circular hierarchy would put the calculations of

ROLLUP into an infinite loop. In your application, you should ensure that your hierarchies are not circular. To do so, use the HIERCHECK program to check every parent relation in your analytic workspace for circularity. You can use HIERCHECK either as a command or as a function.

Improving Performance

When you feel that the ROLLUP command is taking longer than expected, consider the following strategies:

- When your data is sparse, define the data using a composite. Use only one composite for each variable, and be sure to make the composite the slowest-varying dimension (the last dimension in the list). It is a good idea to keep the first dimension in the list dense, and put all the other dimensions into a single composite.
- Use the CHGDFN command with the SEGWIDTH keyword to specify the size of a variable's segments.
- Check the parent relation to make sure it does not define a circular hierarchy. See "Avoiding Circular Hierarchies" on page 21-16.
- Use the LIMIT command to limit the amount of data being rolled up at one time. You can design a program that explicitly loops through a series of subsets of the desired data.
- Set the dimension status to selectively roll up the data. For example, when only some values of a variable have changed, you only need to roll up the data over the ancestors of those values. See "Rolling Up from Changed Child Values" on page 21-15.
- Execute an UPDATE command after every ROLLUP command.

Examples

Example 21–7 Rolling up Sales Data

This example illustrates the use of ROLLUP. You can create an embedded-total `geography` dimension by combining the values in the `district` and `region` dimensions. Another value, `TotalUS`, is the parent of the regions. The order of the dimension values does not matter because the parent relation (that you define later) provides the parent-child information. After the following statements are executed.

```
DEFINE geography DIMENSION TEXT
MAINTAIN geography ADD 'TotalUS' VALUES(region) -
   VALUES(district)
REPORT geography
```

The following report is created.

```
GEOGRAPHY
-----------
TotalUS
East
Central
West
Boston
Atlanta
Chicago
Dallas
Denver
Seattle
```

Next, create the child-parent relation, `geog.geog`, which is the relation between `geography` and itself.

```
DEFINE geog.geog RELATION geography <geography>
```

Each value of the `geog.geog` relation should be the parent of the corresponding `geography` value. You can add the values that are shown in the following report.

```
GEOGRAPHY      GEOG.GEOG
-------------- ----------
TotalUS        NA
East           TotalUS
Central        TotalUS
West           TotalUS
Boston         East
Atlanta        East
Chicago        Central
Dallas         Central
Denver         West
Seattle        West
```

Finally, you can define a variable, `g.units`, for the data that is currently held in the `units` variable plus the totals for the higher levels. After limiting `geography` to the values of `district`, you can transfer the `units` data to `g.units` and use ROLLUP to fill in the totals.

```
DEFINE g.units INTEGER <month product geography>
LIMIT geography TO VALUES(district NOSTATUS)
g.units = UNRAVEL(units)
ROLLUP g.units OVER geography USING geog.geog
LIMIT geography TO ALL
LIMIT product TO 'Tents'
LIMIT month TO 'Jan95' TO 'Jul95'
REPORT W 9 DOWN geography W 9 geog.geog ACROSS month: W 5 -
   g.units
```

The preceding statements produce the following output.

```
PRODUCT: TENTS
         ------------------------G.UNITS----------------------
        -------------------------MONTH------------------------
GEOGRAPHY GEOG.GEOG Jan95 Feb95 Mar95 Apr95 May95 Jun95 Jul95
--------- --------- ----- ----- ----- ----- ----- ----- -----
TotUS     NA        1,429 1,440 1,860 2,534 3,378 3,779 4,058
East      TotUS       453   479   589   848 1,092 1,248 1,315
Central   TotUS       478   494   666   848 1,137 1,247 1,360
West      TotUS       498   467   605   838 1,149 1,284 1,383
Boston    East        200   203   269   359   507   556   545
Atlanta   East        253   276   320   489   585   692   770
Chicago   Central     181   181   247   304   416   443   461
Dallas    Central     297   313   419   544   721   804   899
Denver    West        227   210   283   358   497   573   642
Seattle   West        271   257   322   480   652   711   741
```

Example 21–8   Using a Multidimensional Relation

In this example, we have defined a new dimension called `area` that includes the values in the `geography` dimension that was created in "Rolling up Sales Data" on page 21-17. In addition, `area` includes European and Asian regions and countries that roll up into these regions.

There is also a multidimensional parent relation named `area.area` that defines two hierarchies. The relation `area.area` has `area` for one of its dimensions, while its other dimension, `hier`, holds a list of hierarchies. One of these hierarchies, `Nation`, specifies continental and global totals. The second hierarchy, `Corporate`, divides the child values into divisions and groups of divisions. When an `area` value is not part of a hierarchy, `area.areahier` has an `NA` value for that area. The `area.area`relation also has an `NA` for the top-level area in each hierarchy, since the top level has no parent value. Executing a `DESCRIBE area HIER a.ah` statement shows the following definitions that have been created in the analytic workspace.

```
DEFINE area DIMENSION TEXT
DEFINE hier DIMENSION TEXT
DEFINE area.area RELATION area <area hier>
```

Assume that a `REPORT DOWN area W 20 area.area` statement executes.

The resulting report shows the values of the multidimensional parent relation `area.area`.

```
               ---------------AREA.AREA-----------------
               ------------------HIER-------------------
AREA                  NATION              CORPORATE
-------------- -------------------- --------------------
Global         NA                   NA
GroupI         NA                   Global
GroupII        NA                   Global
DivI           NA                   GroupI
DivII          NA                   GroupI
DivIII         NA                   GroupII
TotalUS        Global               NA
TotInternation Global               NA
TotalEurope    TotInternation       NA
Germany        TotalEurope          Divii
England        TotalEurope          DivI
Spain          TotalEurope          Diviii
France         TotalEurope          Diviii
TotalAsia      TotInternation       NA
India          TotalAsia            DivI
Malaysia       TotalAsia            Diviii
East           TotalUS              NA
Central        TotalUS              NA
West           TotalUS              NA
Boston         East                 DivI
Atlanta        East                 DivI
Chicago        Central              DivI
Dallas         Central              DivI
Denver         West                 DivI
Seattle        West                 DivI
```

The analytic workspace also contains a variable named `a.units` that has `area` as one of its dimensions. Its definition is as follows.

```
DEFINE a.units INTEGER <month product area>
```

After data has been loaded into `a.units` for the lowest level areas (the districts in the United States and the countries of Europe and Asia), you can execute the following ROLLUP command to roll up the data and fill in the totals in the `a.units` variable. The command rolls up data over the `area` dimension, using the multidimensional parent relation `area.area`. This aggregates data both in the `Nation` hierarchy and in the `Corporate` hierarchy.

```
ROLLUP a.units OVER area USING area.area
```

When you use the following statements to produce a report of the `a.units` data, the data for each of the two hierarchies (`Nation` and `Corporate`) will be shown on separate pages of the report.

```
FOR hier
DO
   LIMIT area TO a.ah NE NA
   LIMIT area ADD 'Global'
   REPORT DOWN area W 14 area.area ACROSS month: W 7 a.units
DOEND
```

ROOTOFNEGATIVE

The ROOTOFNEGATIVE option determines the result of any attempt to obtain a root of a negative number.

Data type

BOOLEAN

Syntax

ROOTOFNEGATIVE = YES|NO

Arguments

YES
Allows any attempt to obtain a root of a negative number. This means that a statement that attempts to obtain a root of a negative number will execute without an error; however, the result of the attempt to obtain the root will be `NA`. When you are working with a dimensioned variable or expression, setting ROOTOFNEGATIVE to `YES` enables you to obtain the root of most of the expression's values when a few of the values might be negative.

NO
Disallows any attempt to obtain a root of a negative number. Any statement that attempts to obtain a root of a negative number will stop executing and an error message will be produced. (Default)

Notes

Raising to a Noninteger Power

Raising a number to a noninteger power (for example, `5 ** 0.3` or `14 ** 2.7`) is an attempt to obtain a root.

Examples

Example 21–9 The Effect of ROOTOFNEGATIVE

The following example shows the effect of changing the value of the ROOTOFNEGATIVE option.

The variable TESTNUMBER has a value of `-56`. When you execute a SHOW command such as the following one, without changing the ROOTOFNEGATIVE option from its default value of `NO`, an attempt is made to obtain the square root and then an error message is produced.

```
SHOW SQRT(testnumber)
```

When you change ROOTOFNEGATIVE to `YES`, the same command executes without error

```
ROOTOFNEGATIVE = YES
SHOW SQRT(testnumber)
```

and produces the following result.

```
NA
```

ROUND

Depending on the syntax you specify, the ROUND function performs a numeric operation or a date and time operation. Because the syntax for the ROUND function differs for each type of operation, there are two topics for the ROUND function:

- ROUND (for dates and time)
- ROUND (for numbers)

ROUND (for dates and time)

When a DATETIME expression is specified as an argument, the ROUND function returns a date and time value rounded to a specified date format. When you do not specify a format, the date and time value is rounded to the nearest day.

Return Value

DATETIME

Syntax

ROUND(*datetime_exp*, *format)*

Arguments

datetime-exp
An *expression* that identifies a date and time number.

format
A text expression that specifies one of the format models shown in the following table. A format model indicates how the date and time number should be rounded.

Table 21–1 Format Models for ROUND for Dates and Time

Format Model	**Description**
CC SCC	One greater than the first two digits of a 4-digit year to indicate the next century. For example, `1900` becomes `2000`. `S` indicates that BC dates are marked with a negative (`-`) prefix .
D DAY DY	Starting day of the week (1 to 7). The day of the week that is number 1 is controlled by NLS_TERRITORY.
DD	Day of month (1 to 31).
DDD	Day of year (1 to 366).
HH HH12	Hour of day (1 to 12).
HH24	Hour of day (0 to 23).

Table 21–1 (Cont.) Format Models for ROUND for Dates and Time

Format Model	Description
IW	Same day of the week as the first day of the ISO year.
IYY IY I	Last 3, 2, or 1 digit(s) of ISO year.
IYYY	4-digit year based on the ISO standard.
J	Julian day; that is, the number of days since January 1, 4712 BC.
MI	Minute (0 to 59).
MM	Two-digit numeric abbreviation of month (01 to 12, where January is 01); month rounds up on the sixteenth day.
MON	Abbreviated name of the month; month rounds up on the sixteenth day.
MONTH	Name of the month padded with blanks to 9 characters; month rounds up on the sixteenth day.
Q	Quarter of year (1, 2, 3, 4; JAN to MAR is Q1); quarter rounds up on the sixteenth day of the second month of the quarter.
RM	Roman numeral month (`I` to `XII`, where January is I); month rounds up on the sixteenth day.
WW	Same day of the week as the first day of the year.
W	Same day of the week as the first day of the month.
YEAR SYEAR	Nearest year, spelled out (rounds up on July 1). `S` indicates that BC dates are marked with a negative (`-`) prefix.
YYYY SYYYY	Nearest 4-digit year (rounds up on July 1). `S` indicates that BC dates are marked with a negative (`-`) prefix.
YYY YY Y	Last 3, 2, or 1 digit(s) of nearest year (rounds up on July 1).

Examples

Example 21–10 Rounding to the Nearest Year

When the value of the NLS_DATE_FORMAT option is `DD-MON-YY`, then this statement:

```
SHOW ROUND ('27-OCT-92','year')
```

returns this value:

```
01-JAN-93
```

ROUND (for numbers)

When a number is specified as an argument, the ROUND function returns the number rounded to the nearest multiple of a second number you specify or to the number of decimal places indicated by the second number.

Return Value

DECIMAL (when the round type is MULTIPLE)

NUMBER (when the round type is DECIMAL)

Syntax

ROUND(*number_exp roundvalue*) [<u>MULTIPLE</u>|DECIMAL]

Arguments

number_exp
An expression that identifies the number to round.

roundvalue
A value that specifies the basis for rounding.

When the round type is MULTIPLE:

- *number_exp* is rounded to the nearest multiple of *roundvalue*.
- *roundvalue* can be an integer or decimal number.

When the round type is DECIMAL:

- *roundvalue* specifies the number of places to the right or left of the decimal point to which *number_exp* should be rounded. When *roundvalue* is positive, digits to the right of the decimal point are rounded. When it is negative, digits to the left of the decimal point are rounded.
- When *roundvalue* is omitted, *number_exp* is rounded to `0` decimal places.
- *roundvalue* must be an integer.

MULTIPLE
Specifies that rounding is performed by rounding to the nearest multiple of *roundvalue*. (Default)

DECIMAL
Specifies that rounding is performed by rounding to the number of decimal places indicated by *roundvalue*.

Notes

Using ROUND to Compare Expressions

A DECIMAL value might be stored in a slightly different form than shows up at the level of significant digits you are using. This small difference can cause unexpected results when you are comparing two expressions. The problem can occur even when you are comparing INTEGER expressions that involve calculations because many calculations are done only after converting INTEGER values to DECIMAL values. You do not generally see the difference in reports because reports usually show only two or three decimal places.

For example, when you compare two numbers with the EQ or NE operators, you probably want to ignore any difference caused by the least significant digits. When `expense` was stored as `100.00000001`, the least significant digit would not be ignored by the simple form of the comparison.

The statement

```
SHOW expense EQ 100.00
```

produces the following result.

```
NO
```

However, you can use ROUND to force EQ or NE to ignore the least significant digits.

```
SHOW ROUND(expense, .01) EQ 100.00
```

This statement produces the following result.

```
YES
```

Using ABS to Compare Expressions

When speed of calculation is important in your application, you may want to use the ABS function with LT to compare numbers, instead of using ROUND with EQ or NE. You can use LT and test whether the absolute difference between the two numbers is less than what you regard as significant. For example, you can subtract the two numbers, use the absolute value function, and then compare the result to `.01`.

The statement

```
SHOW ABS(expense - 100.00) LT .01
```

produces the following result.

```
YES
```

Examples

Example 21–11 Rounding to Different Multiples

The following statements show the results of rounding the expression `2/3` to different multiples. The value of the DECIMALS setting is `2`.

The statement

```
SHOW ROUND(2/3, .01)
```

produces the following result.

```
0.67
```

The statement

```
SHOW ROUND(2/3, .1)
```

produces the following result.

```
0.70
```

The statement

```
SHOW ROUND(2/3, .5)
```

produces the following result.

```
0.50
```

Example 21–12 Rounding to the Nearest Thousand

The following example shows `sales` rounded to the nearest thousand.

```
LIMIT month TO FIRST 4
LIMIT district TO FIRST 1
REPORT ROUND(sales 1000)
```

These statements produce the following output.

```
DISTRICT: BOSTON
               -------------ROUND(SALES 1000)-------------
               -------------------MONTH-------------------
PRODUCT          Jan95      Feb95      Mar95      Apr95
-------------- ---------- ---------- ---------- ----------
Tents           32,000.00  33,000.00  43,000.00  58,000.00
Canoes          66,000.00  76,000.00  92,000.00 126,000.00
Racquets        52,000.00  57,000.00  59,000.00  69,000.00
Sportswear      53,000.00  59,000.00  63,000.00  68,000.00
Footwear        91,000.00  87,000.00 100,000.00 108,000.00
```

Example 21–13 Rounding to the Nearest Multiple of 12

To show `units` rounded to the nearest multiple of 12, use the following statements.

```
LIMIT month TO FIRST 4
LIMIT district TO FIRST 1
REPORT DECIMAL 0 ROUND(units 12)
```

These statements produce the following output.

```
DISTRICT: BOSTON
               --------------ROUND(UNITS 12)--------------
               -------------------MONTH-------------------
PRODUCT          Jan95      Feb95      Mar95      Apr95
-------------- ---------- ---------- ---------- ----------
Tents                 204        204        264        360
Canoes                348        396        480        660
Racquets              996      1,080      1,116      1,308
Sportswear          1,092      1,212      1,296      1,404
Footwear            2,532      2,400      2,772      2,976
```

Example 21–14 Rounding to Decimal Places

The following statements show the results of rounding `15.193` to various decimal places.

The statement

```
ROUND(15.193, 1)
```

produces the following result

```
15.2
```

The statement

```
ROUND(15.193, -1)
```

produces the following result

```
20
```

ROW command

The ROW command produces a line of data in cells, one after another in a single row. A series of ROW commands that produce corresponding cells are often used to build up columns of data. For this reason, we normally speak of the ROW command as producing a line of columns. Output from the ROW command is sent to the current outfile.

The ROW command is typically used in conjunction with other statements, functions, and options that you can think of collectively as *report-writing* statements

The ROW command itself consists of a series of column descriptions that specify the data to be produced and, optionally, the output format of the data.

In addition, ROW has a versatile capability for doing row and column arithmetic. It can perform calculations and include the calculation results in the output. It can use any kind of calculated expression in the column descriptions; and it can take advantage of row and column totaling functions (see Table 21–3, " Row and Column Arithmetic" on page 21-38).

ROW is primarily used in report programs to produce the lines of the report.

Syntax

ROW [*attributes*] [ACROSS *dimension* [*limit-clause*]:] {*exp1*|SKIP } -

    [[*attributes*] [ACROSS *dimension* [*limit-clause*]:] {*expn*|SKIP }]

Arguments

attributes
One or more attributes for a column. Attributes are format specifications that determine how the data value is formatted within the column. There is no limit to the number of attributes that you can use to describe a column format. (See Table 21–2, " Column Attributes for ROW" on page 21-35for an explanation of each of the available attributes.) The default for some format attributes is determined by the current setting of Oracle OLAP options (see Table 21–4, " Report-Related Options" on page 21-38for a list of these options). ROW with no arguments produces a blank line.

ACROSS *dimension* [*limit-clause*]:
An ACROSS phrase lets you include more than one value of a dimensioned expression in a single row by looping over one of the dimensions (or composites) of the expression. Normally ROW just shows the value that corresponds to the first dimension value within the current limits. With an ACROSS phrase, ROW produces one data column for each dimension value currently in the status.

You can apply a single ACROSS phrase to multiple data expressions, or you can use separate ACROSS phrases for different data expressions. See "Multiple Expressions" on page 21-40 and "Separate ACROSS Phrases" on page 21-41.

When you show data for a variable dimensioned by a composite and you do not include an ACROSS phrase, ROW shows output for all data cells that correspond to the base dimension values of the composite. When a particular combination of base dimension values does not exist in the composite, ROW shows `NA` for the corresponding data cell. Likewise, when you specify one of the composite's base dimensions in an ACROSS phrase, ROW shows `NA` for a data cell for which the composite contains no value.

However, when you specify a composite in the ACROSS phrase, ROW shows output only for data cells for which combinations of base dimension values exist in the composite. This provides a more concise report that better reflects your data.

When the dimension specified in an ACROSS phrase has null status, ROW does not produce any data columns for that ACROSS phrase.

When you specify a composite in the ACROSS phrase, you cannot include a *limit-clause* argument. You must limit the base dimensions of a composite to the desired values before you execute a ROW command.

However, when you specify a dimension in the ACROSS phrase, *limit-clause* enables you to change the status of that dimension. The new status will be in effect only for the duration of the ROW command. The format of *limit-clause* is as follows.

[ADD|COMPLEMENT|KEEP|REMOVE|INSERT|TO] *valuelist* [IFNONE *label*]

To specify the temporary status, insert any of the LIMIT keywords (the default is TO) along with an appropriate value list or related-dimension list. You can use any valid LIMIT clause (see the entry for the LIMIT command for further information). The following example temporarily limits month to the last six values, no matter what the current status of month is.

```
ACROSS month LAST 6: units
```

When the limits you specify result in empty status for the dimension, an error occurs. However, when you include the phrase IFNONE *label*, the error is suppressed and execution of your program branches to the specified label where you can handle the error.

SKIP

Used in place of an expression to indicate that the column is to be left blank.

Table 21–2 Column Attributes for ROW

Attribute	**Meaning**
WIDTH *n* (W *n*)	Makes the column *n* spaces wide. The default width for the first column is the value of the LCOLWIDTH option. For other columns, it is the value of the COLWIDTH option. The maximum width is 4000 characters. Columns with a width of `0` are suppressed.
SPACE *n* (SP *n*)	Precedes the column with *n* spaces. The default for the first column is `0`; for other columns, `1`.
INDENT *n*	Indents the value *n* spaces within its column. The default is 0.
LEFT (L)	Left-justifies the value within its column. This is the default for TEXT data.
RIGHT (R)	Right-justifies the value within its column. This is the default for numeric and Boolean data.
CENTER (C)	Centers the value within its column.
LSET '*text*'	Adds *text* to the left of the value.
<u>NOLSET</u>	Does not add anything to the left of the value.
RSET '*text*'	Adds *text* to the right of the value.
<u>NORSET</u>	Does not add anything to the right of the value.
FILL '*char*'	Puts *char* into unused positions in the column. The default fill character is a space.
DECIMAL *n* (D *n*)	Shows *n* decimal places. Decimal places are separated by the character currently specified by the DECIMALCHAR option. The default number of decimal places is controlled by the DECIMALS option.
<u>NODECIMAL</u>	Shows the number of decimal places indicated by the DECIMALS option.

Table 21–2 (Cont.) Column Attributes for ROW

Attribute	Meaning
COMMA	Marks thousands and millions with commas or the character currently recorded in the THOUSANDSCHAR option. The default is controlled by the COMMAS option.
NOCOMMA	Does not mark thousands and millions.
PAREN	Uses parentheses to indicate negative numbers. The default is controlled by the PARENS option.
NOPAREN	Uses the minus sign to indicate negative numbers. The default is controlled by the PARENS option.
LEADINGZERO	Puts a leading zero before decimal numbers between `-1` and `1`.
NOLEADINGZERO	Suppresses leading zeros before decimal numbers between `-1` and `1`.
<u>CNLEADINGZERO</u>	Puts a leading zero before decimal numbers between `-1` and 1 when it does not cut off any significant digits.
MNOTATION	Always uses M-notation (divides values by one million and appends `M`).
<u>CMNOTATION</u>	Conditionally uses M-notation, when needed to make a value fit in a column.
NOMNOTATION	Does not use M-notation (uses asterisks for oversize values).
MDECIMAL *n*	Shows *n* decimal places in numbers formatted with M-notation; *n* can be any number from 0 to 16, or 255.
ENOTATION	Always uses scientific notation, also called exponential notation or E-notation (appends `E`, and includes a sign before the exponent, for example, `.230E+2` or `.230E-2`).
CENOTATION	Conditionally uses E-notation, when needed to make a value fit in a column.
NOENOTATION	Does not use E-notation (defaults to conditional M-notation).
EDECIMAL *n*	Shows *n* decimal places in numbers formatted with E-notation; *n* can be any number from 0 to 16, or 255.
NASPELL '*text*'	Uses *text* in place of `NA` values. The default is controlled by the NASPELL option.
<u>NONASPELL</u>	Spells `NA` values as indicated by the NASPELL option.
ZSPELL '*text*'	Uses *text* in place of zero numeric values. The default is controlled by the ZSPELL option.

Table 21–2 (Cont.) Column Attributes for ROW

Attribute	**Meaning**
NOZSPELL	Spells zero values as indicated by theZSPELL option.
YESSPELL *'text'*	Text used for `TRUE` Boolean values. The default is recorded in the YESSPELL option.
NOSPELL *'text'*	Text used for `FALSE` Boolean values. The default is recorded in the NOSPELL option.
TRUNCATE (TRUNC)	Truncates a character value to the column width when it does not fit in the column.
NOTRUNCATE (NOTRUNC)	Creates additional lines when the character value does not fit in the column.
FOLDUP	For a multiline character value, places all but the last line above the rest of the row, and the last line on the row with the other values; also strips any leading or trailing spaces.
FOLDDOWN	For a multiline character value, places the first line on the row with the other values, and places additional lines below the rest of the row; also strips any leading or trailing spaces.
VALONLY	Underlines or overlines the value only. (Used with UNDER and OVER.)
NOVALONLY	Underlines or overlines the entire width of the column. (Used with UNDER and OVER.)
UNDER *textexp*	Underlines the value or column with the value of a character expression (*textexp*). When *textexp* is a literal value, it must be enclosed in single quotes. Useful literal values include: `'-'` to underline value or column, `'='` to double underline value or column, and `''` to indicate that a value or column is not underlined. To underline only when a condition is met, for *textexp* use `IF` *`boolean-expression`* `THEN '-' ELSE ''`
OVER *textexp*	Overlines the value or column with the value of a character expression (*textexp*). When *textexp* is a literal value, it must be enclosed in single quotes. Useful literal values include: `'-'` to overline value or column, `'='` to double overline value or column, and `''` to indicate that a value or column is does not have an overline To overline only when a condition is met, for *textexp* use `IF` *`boolean-expression`* `THEN '-' ELSE ''`

Use the functions that are listed in Table 21–3, " Row and Column Arithmetic" to perform calculations on the values generated so far in a report.

Table 21–3 Row and Column Arithmetic

Function	Data Type	Value Returned
COLVAL(*n*)	DECIMAL	Value in the nth column of the current row. When *n* `> 0`, an absolute column number (from the left margin, moving to the right). When *n* < `0`, a relative column number (from the current column, moving left).

RUNTOTAL(*n*)
n = 1,2, ...32
	DECIMAL	Total of all numbers generated in the current column since the last SUBTOTAL or ZEROTOTAL for *n*. Does not reset total for *n* to `0`.

SUBTOTAL(*n*)
n = 1,2, ...32
	DECIMAL	Total of all numbers generated in the current column since the last SUBTOTAL or ZEROTOTAL for *n*. Resets total for *n* to `0`.

The options that are listed in Table 21–4, " Report-Related Options" on page 21-38 affect the default format for a ROW command.

Table 21–4 Report-Related Options

Option	Meaning
COLWIDTH	Column width for all but the first column when the WIDTH attribute is not used. The default is `10`.
COMMAS	Specifies whether a thousands group separator is used when neither the COMMA attribute nor the NOCOMMA attribute is used. The default is `YES` (uses a separator).
DECIMALS	Number of decimal places when the DECIMAL attribute is not used. The default is `2`.
LCOLWIDTH	Column width for the first column when the WIDTH attribute is not used. The default is `14`.
LSIZE	Defines the line size within which the STDHDR program centers the standard header. The default is `80` characters.
NASPELL	Text used for `NA` values when the NASPELL attribute is not used. The default text is `NA`.

Table 21–4 (Cont.) Report-Related Options

Option	**Meaning**
NLS_LANGUAGE	Specifies the text used for `TRUE` and `FALSE` Boolean values. These values are reflected in the YESSPELL and NOSPELL options.
NLS_TERRITORY	Specifies the character used for the decimal marker and the thousands group separator. These values are reflected in the DECIMALCHAR and THOUSANDSCHAR options.
PARENS	Parentheses usage for negative numbers when neither the PAREN attribute nor the NOPAREN attribute is used. The default is `NO` (does not use parentheses; uses a minus sign).
ZEROROW	Controls generation or suppression of rows in which all numeric values are zero. The default is `NO` (generates zero rows).
ZSPELL	Text used for zero values when theZSPELL attribute is not used. The default text is `OFF`, which shows a zero (`0`).

Use the statements that are listed in Table 21–5, " OLAP DML Statements That Are Compatible with the ROW Command" with the ROW command.

Table 21–5 OLAP DML Statements That Are Compatible with the ROW Command

Command	**Action**
BLANK *n*	Produces *n* blank lines. The default is one line.
HEADING column-description(s)	Produces titles and column headings for a report. Numeric values in headings are not added to column totals.
PAGE	Forces a page break in output when PAGING is set to `YES`.
ZEROTOTAL	Resets all 32 totals to `0` for all columns.
ZEROTOTAL ALL *col(s)*	Resets all 32 totals to `0` for the specified columns, or for all columns when there are no column arguments.
ZEROTOTAL *n col(s)*	Resets the indicated total (*n*) to `0` for the specified columns, or for all columns when there are no column arguments.

Notes

Report-Writing Commands

The ROW command and its associated options and commands are referred to collectively as *report-writing* statements. Table 21–3, " Row and Column Arithmetic" on page 21-38 lists functions you can use for performing row and column arithmetic in reports. Table 21–4, " Report-Related Options" on page 21-38 lists report-related options that determine the default format for ROW output. Table 21–5, " OLAP DML Statements That Are Compatible with the ROW Command" on page 21-39 lists additional statements that are used in combination with ROW to create reports.

Paging Options

You can use the PAGING option and associated paging-related options to produce your report program in a page-oriented format.

Maximum Row Width

The maximum width of any row in a report is 4000 characters.

Unnamed Composites

You can specify an unnamed composite as the *dimension* argument by using the syntax that was used to create the unnamed composite.

Labels for Composites and Conjoint Dimensions

When you produce a report of data that has a composite or a conjoint dimension in its dimension list, you can produce a label column for each base dimension by using the KEY function. You can also provide a separate WIDTH attribute for each label column. For example, when `proddist` is a composite with the base dimensions `product` and `district`, you can use statements similar to the following ones.

```
FOR proddist
   ROW W 12 KEY(proddist district) W 8 KEY(proddist product) ...
```

Multiple Expressions

When you want the same format attribute or ACROSS phrase to apply to more than one data expression, you can enclose the expressions in angle brackets (< >) and place the common attributes or ACROSS phrase immediately before the bracketed expressions.

attributes <*expression1*, *expression2*, ...>

or

ACROSS *dimension*: <*expression1*, *expression2*, ...>

When you have attributes that apply to only one of the expressions within the brackets, place the specific attributes immediately before the expression.

attributes1 <*expression1*, *attributes2 expression2*>

When an attribute inside angle brackets (specific to a column) conflicts with an attribute outside the brackets (common to several columns), the specific attribute overrides the common attribute.

You can nest brackets to any depth, as long as you have an equal number of right and left brackets.

Separate ACROSS Phrases

For data generated with an ACROSS phrase, you can produce all the columns for one expression and then all the columns for additional expressions by using separate ACROSS phrases.

ACROSS *dim*: *expression1*, ACROSS *dim*: *expression2*

You also can nest ACROSS phrases to show data columns for two or more dimensions of an expression across a row.

ACROSS *dim1*: ACROSS *dim2*: *expression*

Using Properties for Attributes

When a variable has a formatting property attached to its definition, you can use the OBJ function to obtain the value of that property and use it as the value of an attribute in the ROW command.

Large Data Values

When a numeric value is too large to fit into a data cell, ROW rounds it off to the nearest million with the symbol M at the right side of the cell. When a value is still too large, ROW replaces the value with asterisks.

Decimal Values Between -1 and 1

When you set the DECIMAL attribute to `0` and you use the NOLEADINGZERO keyword, any decimal values between `-1` and `1` that are rounded to 0 will not be shown.

LSET or RSET with NA Values

When you use the LSET or RSET attribute with an expression that contains `NA` values, the text you specify with LSET or RSET will not be included at the left or right of any `NA` values.

Setting Options

When you plan to use Oracle OLAP options to format the data shown by ROW commands within a program, set these options before they are used in the ROW command so that they have the values you want to use. The following statements set the DECIMALS option before the ROW command uses it to produce sales data.

```
DECIMALS = 0
ROW district month product sales
```

Row and Column Arithmetic

See Table 21–3, " Row and Column Arithmetic" on page 21-38 for a list of the functions available for row and column arithmetic. You can use these functions to perform calculations on the values already generated in a report. Oracle OLAP maintains 32 running totals for each column, so you can include up to 32 levels of subtotals in a report.

Decimal Overflow

When a "decimal overflow" condition occurs while subtotals are being accumulated (that is, an out-of-range value is generated), all subtotals for the affected column are set to `NA` and processing continues when the DECIMALOVERFLOW option is set to `YES`. The subtotals for the column will continue to be `NA` until they are reset by a ZEROTOTAL command. When DECIMALOVERFLOW is `NO`, an error occurs when a decimal overflow condition occurs.

Processing Output from ROW

You can also use ROW as a function that returns the ROW output for further processing, rather than sending the output to the current outfile. For more information, see ROW function.

Improving Report Performance

When you know ahead of time that you will not need the subtotaling capability of the ROW command, you can save execution time by using the HEADING command instead of ROW to produce the lines of your report, since Oracle OLAP will not be keeping track of subtotals.

Performance Tip for Using ROW with Variables Dimensioned by Composites

By default, when ROW explicitly loops over a composite, or when ROW is executed in a FOR loop that explicitly loops over a composite, Oracle OLAP sorts the composite values according to the current order of the values in the composite's base dimensions. The task of sorting requires some processing time, so when variables are large, performance can be affected. When your variable is very large, and you are more concerned about performance than about the order in which ROW output is produced, you can set the SORTCOMPOSITE option to `NO`.

Using the ROW Command in a Program

For information on using the ROW command in a program, see the entries for FOR, DO ... DOEND, and WHILE.

Examples

Example 21–15 Labeling Data Values

In this example, ROW produces a line of output that contains a value of `sales`, along with the corresponding dimension values for `district`, `month`, and `product` that identify it.

```
ROW W 8 district month product sales
```

The preceding statement produces the following row of output.

```
Boston        Jan95      Tents  32,153.52
```

Example 21–16 Reporting Two Variables

The line of output produced by this ROW command contains the current dimension value of `district`, followed by the values of `sales` and `sales.plan` for Sportswear for each of the first two months of 1996.

```
LIMIT month TO 'Jan96' 'Feb96'
LIMIT product TO 'Sportswear'
ROW W 8 district ACROSS month: <sales sales.plan>
```

These statements produce the following row of output.

```
Boston    57,079.10  61,434.20  63,121.50  64,006.91
```

Example 21–17 Formatting and Labeling the Output

In this ROW command, you want to see the actual and planned sales of tents for June 1996. You want to limit the status of `month` only for this one ROW command, so you include the value `Jun96` in the ACROSS phrase. You format the values as whole dollar amounts, and you also add a dollar sign to the values, along with individual labels that identify the actual and planned figures.

```
LIMIT product TO 'Tents'
ROW WIDTH 15 name.product ACROSS month 'Jun96': -
   DECIMAL 0 LSET '$' W 18 -
   <RSET ' (actual)' sales -
   RSET ' (plan)' sales.plan>
```

These statements produce the following row of output.

```
3-Person Tents    $95,121 (actual)     $80,138 (plan)
```

Example 21–18 Reporting on a Variable Dimensioned by a Composite

In this example, D.SALES is a variable whose dimension list includes the dimension `month` and the unnamed composite `SPARSE <product district>`. By specifying the composite in an ACROSS phrase of a ROW command, you can produce a report that includes only the data cells for which the composite contains values.

```
LIMIT product TO ALL
LIMIT district TO 'Atlanta'
LIMIT month TO 'Jan96'
ROW ACROSS SPARSE <product district>: d.sales
```

ROW function

The ROW function returns a line of data in cells, one after another in a single row. It is identical to the ROW command, except that it returns a text value, instead of sending the text to the current outfile.

The ROW function, just like the ROW command, consists of a series of column descriptions that specify the data to be returned and, optionally, the way in which it is to be formatted. The ROW function lets you assign the returned value to a text variable, send it to your current outfile with the SHOW or REPORT command, or process it further as an argument to one of the character manipulation functions.

Return Value

TEXT

Syntax

ROW([*attribs*] [ACROSS *dimension* [*limit-clause*]:] {*exp1*|SKIP} -

  [[*attribs*] [ACROSS *dimension* [*limit-clause*]:] {*expn*|SKIP}])

See the ROW command for a complete description of the arguments.

The ROW function without any arguments returns a blank line.

Notes

ROW Command Attributes

You can use the same attributes that are available for the ROW command. Refer to the ROW command entry for a table of attributes that format the data and a table of functions that perform row and column arithmetic.

ROW Command Notes

The notes for the ROW command also apply to the ROW function.

Examples

Example 21–19 Assigning Output to a Text Variable

The following assignment statement assigns three lines of output to the text variable `textvar`.

```
textvar = ROW(OVER '-' UNDER '=' 'This is a Row.')
SHOW textvar
```

These statements produce the following output.

```
--------------
This is a Row.
==============
```

Example 21–20 Producing Multiple Rows of Output

You can use the ROW function with JOINLINES in a program to loop over a group of dimension values and assign several rows of data to a text variable. Instead of using the SHOW command in the following program excerpt, you could use the contents of `textvar` for some other purpose.

```
LIMIT month TO 'Jan95' 'Feb95'
LIMIT district TO 'Boston' 'Atlanta' 'Chicago'
textvar = NA
textvar = ROW(W 8 SKIP ACROSS month: <month SKIP>)
textvar = JOINLINES(textvar ROW(W 8 SKIP ACROSS month: -
   CENTER <'Sales' 'Plan'>))
FOR district
textvar = JOINLINES(textvar ROW(W 8  district ACROSS month: -
   <sales sales.plan>))
SHOW textvar
```

These statements produce the following output.

```
OUTPUT:
         Jan95                Feb95
          Sales       Plan     Sales       Plan
Boston   32,153.52  42,346.89  32,536.30  43,265.50
Atlanta  40,674.20  54,583.41  44,236.55  57,559.87
Chicago  29,098.94  36,834.37  29,010.20  37,667.66
```

RPAD

The RPAD function returns an expression, right-padded to a specified length with the specified characters; or, when the expression to be padded is longer than the length specified after padding, only that portion of the expression that fits into the specified length.

To left-pad a text expression, use LPAD.

Return Value

TEXT or NTEXT based on the data type of the expression you want to pad (*text-exp*).

Syntax

RPAD (*text-exp* , *length* [, *pad-exp*])

Arguments

text-exp
A text expression that you want to pad.

length
The total length of the return value as it is displayed on your terminal screen. In most character sets, this is also the number of characters in the return value. However, in some multibyte character sets, the display length of a character string can differ from the number of characters in the string.

When you specify a value for *length* that is shorter than the length of *text-exp*, then this function truncates the expression to the specified length.

pad-exp
A text expression that specifies the padding characters. The default value of *pad-exp* is a single blank.

Examples

The following example right-pads a name with the letters "ab" until it is 12 characters long.

```
SHOW RPAD('Morrison',12,'ab')
Morrisonabab
```

RTRIM

The RTRIM function removes characters from the right of a text expression, with all the rightmost characters that appear in another text expression removed. The function begins scanning the base text expression from its last character and removes all characters that appear in the trim expression until reaching a character that is not in the trim expression and then returns the result.

To leading characters, use LTRIM. To trim both leading or trailing characters, use TRIM.

Return Value

TEXT or NTEXT based on the data type of the first argument.

Syntax

RTRIM (*text-exp* [, *trim-exp*])

Arguments

text-exp
A text expression that you want trimmed.

trim-exp
A text expression that is the characters to trim. The default value of *trim-exp* is a single blank.

Examples

The following example trims all of the right-most a's from a string.

```
SHOW RTRIM('Last Wordxxyxy','xy')
Last Word
```

RUNTOTAL

The RUNTOTAL function returns the running total of an expression. You can use the RUNTOTAL function in a ROW command, ROW function, or REPORT command to generate a running total of the value of an expression.

Return Value

DECIMAL

Syntax

RUNTOTAL(*n*)

Arguments

n
One of the 32 subtotals (1 to 32) that Oracle OLAP accumulates for the current column of a report. RUNTOTAL returns the value of this subtotal for the specified column, but does not reset the value of the subtotal to zero.

Notes

How RUNTOTAL Works

Unlike the SUBSTR function, RUNTOTAL does not reset the indicated subtotal to zero, nor does it add the value returned by RUNTOTAL to the indicated subtotal. However, the value returned by RUNTOTAL *is* added to the other 31 accumulating totals for the current column.

Accessing Data from Another Column

You can obtain a running total of an expression shown in another column of a report by adding that expression to RUNTOTAL. You can use the COALESCE function to refer to the values in the other column. For example, to show the sales for each month in the first data column of a row, and a cumulative total of sales in the second data column, you could use this statement.

```
ROW month sales COLVAL(-1) + RUNTOTAL(1)
```

Resetting Subtotals

When you use the ROW command to produce a report, you can use the ZEROTOTAL command to reset any subtotal of any column to zero. Normally, you should do this at the beginning of a report program to make sure all totals begin at zero. The REPORT command automatically resets all subtotals to zero before producing output.

Referring to Subtotals

The numbers by which the 32 subtotals are referenced (1 to 32) have no intrinsic significance; all the subtotals are the same until you reference them.

NA Values

RUNTOTAL ignores `NA` values unless all values are `NA`. When all values are `NA`, the total is `NA`.

Decimal Overflow

When a "decimal overflow" condition occurs while subtotals are being accumulated (that is, an out-of-range value is generated), all subtotals for the affected column are set to `NA` and processing continues when the DECIMALOVERFLOW option is set to `YES`. The subtotals for the column will continue to be `NA` until they are reset by a ZEROTOTAL command. When DECIMALOVERFLOW is `NO`, an error occurs when a decimal overflow condition occurs.

Examples

Example 21–21 Calculating a Running Total in a Report

In a report, suppose you want column 2 to contain a running total of the values in column 1.

Assume that you issue the following OLAP DML statements

```
ZEROTOTAL ALL
ROW W 4 R 2 RUNTOTAL(1) + COLVAL(1)
ROW W 4 R 5 RUNTOTAL(1) + COLVAL(1)
ROW W 4 R 3 RUNTOTAL(1) + COLVAL(1)
```

These statements produce the following output.

```
2     2.00
5     7.00
3    10.00
```

Example 21–22 Calculating a Running Total over Two Districts

In this example, you want your report to contain the unit sales of tents for two districts for the first six months of 1996. Along with the monthly sales figures, you want to see a running total of tent sales for these two districts for the year to date. To produce this cumulative total, use the RUNTOTAL function.

```
LIMIT product TO 'Tents'
LIMIT month TO 'Jan96' TO 'Jun96'
LIMIT district TO 'Boston' 'Chicago'
REPORT ACROSS district: units -
   DECIMAL 0 TOTAL(units, month)+RUNTOTAL(1)
```

These statements produce the following output.

```
PRODUCT: TENTS
               --------UNITS--------
               ------DISTRICT-------
                                     TOTAL(UNIT
                                         S,
                                     MONTH)+RUN
MONTH            Boston    Chicago    TOTAL(1)
-------------- ---------- ---------- ----------
Jan96                 307        189        496
Feb96                 209        190        895
Mar96                 277        257      1,429
Apr96                 372        318      2,119
May96                 525        433      3,077
Jun96                 576        466      4,119
```

SECONDS

(Read-only) The SECONDS option holds the number of seconds since January 1, 1970. As an aid to enhancing a program's speed, SECONDS can be used to determine how many real seconds elapse while the program is running.

Data type

INTEGER

Syntax

```
SECONDS
```

Notes

Related Statements

For information about holding the number of seconds in decimal form, see the DSECONDS command. For information about programs, see the PROGRAM command.

Examples

Example 21–23 Timing a Program

The following program puts the value of SECONDS at the start of the program in a variable called `t1`, then displays the difference between `t1` and the value of SECONDS at the end of the program.

```
DEFINE prodsummary PROGRAM
PROGRAM
VARIABLE t1 INTEGER
t1 = seconds
LIMIT product TO ALL
BLANK
FOR product
DO
  ROW WIDTH 16 name.product ACROSS month Jun96: DECIMAL 0 LSET -
   '$'WIDTH 18 <RSET ' (actual)' sales RSET ' (plan)' sales.plan>
DOEND
BLANK
ROW WIDTH 35 LSET 'the program took ' RSET ' SECOND(s).' -
 (SECONDS-t1)
END
```

Running this program produces the following results.

```
3-Person Tents     $95,121 (actual)     $80,138 (plan)
Aluminum Canoes   $157,762 (actual)    $132,931 (plan)
Tennis Racquets    $97,174 (actual)     $84,758 (plan)
Warm-up Suits      $79,630 (actual)     $73,569 (plan)
Running Shoes     $153,688 (actual)    $109,219 (plan)

      The program took 2 second(s).
```

SESSCACHE

Typically used only when debugging, the SESSCACHE option controls whether Oracle OLAP creates an Oracle OLAP session cache described in "What is an Oracle OLAP Session Cache?" on page 21-54.

Syntax

SESSCACHE = {YES|NO}

Arguments

YES
The session cache is created to hold the data described in "What is an Oracle OLAP Session Cache?" on page 21-54.

NO
Oracle OLAP does not read or write to the session cache. When you specify `NO`, caching does not occur even when you have specified caching by coding a CACHE SESSION statement in the specification for one or more aggmap objects, by setting one or more $VARCACHE properties to `SESSION`, or by setting the VARCACHE option to `SESSION`.

Notes

What is an Oracle OLAP Session Cache?

An Oracle OLAP session cache is a special place in memory used to hold:

- All data that was calculated on the fly when an AGGREGATE function executed in the following situations:
  - The specification for the aggregation included a CACHE SESSION.
  - The specification for the aggregation did not include a CACHE SESSION statement, but the variable being aggregated had a $VARCACHE property with the value of `SESSION`.
  - The specification for the aggregation did not include a CACHE SESSION statement and the variable being aggregated did not have a $VARCACHE property, but the VARCACHE option was set to `SESSION`.

- The NA values (only) that were calculated when an AGGREGATE function executed and the specification for the aggregation included a CACHE NA statement.
- All data that was calculated when a $NATRIGGER expression executed in the following situations:
  - The variable with the $NATRIGGER property also had a $VARCACHE property with the value of SESSION.
  - The variable with the $NATRIGGER property did not have a $VARCACHE property, but the VARCACHE option was set to SESSION.

There is one internal cache for a session. Cached data is ignored by UPDATE and COMMIT statements. However, once data is cached, Oracle OLAP uses the values in the cache for all calculations except when an AGGREGATE function with the FORCECALC keyword executes. In this case, the FORCECALC keyword specifies that Oracle OLAP recalculate the values.

When a session is terminated, its cache is cleared. To clear the session cache without terminating the session, issue a CLEAR statement.

The effectiveness of a session cache is tracked in the V$AW_CALC dynamic performance view which is discussed in the *Oracle OLAP Reference*.

SET

The SET command, also called an assignment statement or the = command, assigns one or more values to a variable, option, relation, or dimension surrogate. When an object has one or more dimensions, teh SET command loops over the values in status for each dimension of the target object and assigns a data value to the corresponding cell of the target object.

When the target is an object defined with a composite in its dimension list, Oracle OLAP automatically creates any missing target cells that are being assigned non-`NA` values. This step also adds to the composite all the dimension value combinations that correspond to those new cells. Thus, both the target object and the composite might be larger after an assignment. When you want to assign values only to cells that already exist in the target, use the ACROSS keyword.

> **See also:** You can use UNRAVEL in conjunction with an assignment statement to assign values of an expression into the cells of a variable when the dimensions of the expression are not the same as the dimensions of the variable.

Syntax

[SET] *target-name* [=] *expression* [ACROSS *composite*]

Arguments

SET
SET is optional. It is an older command form of this functionality, and is included for compatibility.

target-name
The name of the target object where the data will be assigned and stored. For a list of analytic workspace objects that can be a target object, see Table 21–6, " Use of Analytic Workspace Objects in OLAP DML Assignment (SET) Statement".

=
The = (assignment or equal) operator assigns one or more values to a variable, option, or relation.

expression

The source of the data values to be assourcearget object, see Table 21–6, " Use of Analytic Workspace Objects in OLAP DML Assignment (SET) Statement"

ACROSS *composite*

When you are assigning data to a variable dimensioned by a composite the default behavior is to loop over all the values in status for each of the base dimensions of the object. Oracle OLAP automatically creates any missing target cells that are being assigned non-NA values, and it automatically adds the required dimension value combinations to the composite.

When you want to assign values only to existing cells of a variable defined with a composite, use the ACROSS keyword, which causes = to change the way it loops for those dimensions of the target that are part of the composite. Instead of looping over all possible combinations of the values in the status of those dimensions, = loops only over those combinations of the values in the status that already exist in the composite.

The ACROSS keyword is intended for specifying a composite. However, when you specify a base dimension of the composite instead, be aware that the assignment statement could add many values to your composite.

Notes

Triggering Program Execution When an Assignment Statement Executes

Using the TRIGGER command, you can make the SET command an event that automatically executes an OLAP DML program. See "Trigger Programs" on page 1-14 for more information

Dimensionality and Performance

When the target has more than one dimension, the = command loops over the dimension values in the order in which they were added, regardless of their logical order as reflected by the default status. In a multidimensional case, the looping is over the compound dimension. The first dimension listed in the definition varies the fastest. When you are setting the target to the values of an expression, Oracle OLAP performs much more efficiently when the source expression has the same dimensions, in the same order, as the target.

Differently Dimensioned Variables in an Expression

When an assignment statement involves a number of differently dimensioned objects, the calculation can appear complicated. The following list outlines the process followed by a complicated assignment statement. When the command is A = B, where A is the object being set to the expression B, Oracle OLAP first determines the dimensions of A. Then it determines the status of those dimensions. For each combination of dimension values in the status of those dimensions:

1. Oracle OLAP determines which single value of A (sometimes called a cell) is going to be set.
2. For each component of the expression B (each variable, formula, function, qualified data reference, or literal), Oracle OLAP determines the single value that corresponds to the cell of A that is being set. When a component of the expression is not dimensioned or is a literal, Oracle OLAP simply uses its value. When a component of the expression has dimensions different from A, Oracle OLAP uses the first value in the status of these dimensions.
3. Oracle OLAP performs the specified calculation on the single values obtained in Step 2 and stores the result in the cell of A chosen in Step 1.

Using Objects in Assignment Statements

Table 21–6, " Use of Analytic Workspace Objects in OLAP DML Assignment (SET) Statement" outlines the objects that you can use in assignment statements and indicates whether you can use them as a target or source expression.

Table 21–6 Use of Analytic Workspace Objects in OLAP DML Assignment (SET) Statement

Object	Target Expression	Source Expression
Variable	Yes	Yes
Relation	Yes	Yes
Dimension	Only in models	Yes
Surrogate	Yes	Yes
Composite	No	Yes
Worksheet	Yes	Yes
Function	No	Yes
Formula	Yes	Yes
Valueset	No	Yes

Assigning Values to Variables When you use an = (SET) statement to assign the value of a single-cell expression to a single cell, a single value is stored. However, when you use an = statement to assign the value of a single-cell expression to a target variable that has one or more dimensions, then the assignment loops over the values in status for each dimension of the target variable and assigns a data value to the corresponding cells of the variable.

When you assign a multiline value to a fixed-width text variable, then the variable is set to the first line only. To assign a multiline value to a fixed-width text variable, you use the JOINCHARS function to make the multiline value one line long. For example, suppose you have a non-fixed-width text variable called `textvar`. The statement

```
SHOW textvar
```

produces the following output, in which each line of the value in `textvar` is shown as a separate line.

```
This is a variable
that has a multiline
text value.
```

To assign this value to a variable called `fixedtext` with a fixed width of 60 bytes and show the value, you would use the following statements.

```
fixedtext = JOINCHARS(textvar)
SHOW fixedtext
```

These statements produce the following output, in which the value of `textvar` is shown as a single line.

```
This is a variable that has a multiline text value.
```

When the actual number of bytes in the `textvar` variable's value exceeds the width of the `fixedtext` variable, then the value of `textvar` will be truncated when it is stored in `fixedtext`.

Assigning Values to Relations You can assign values to a relation using a SET statement as illustrated in Example 21–25, "Assigning Values to a Relation" on page 21-65. When executing the assignment statement, a loop is performed over the values in status for each dimension of the target relation and assigns a data value to the corresponding cell of the target relation.

You can assign values to a relation with a text dimension by assigning one of the following:

- A text value of the dimension.
- An INTEGER that represents the position of the dimension value in the default status list of the dimension.

Assigning Values to Dimensions The only time you use an = statement to assign a value to a dimension is when the result of a calculation in a model equation is numeric. In this situation, you can use the = operator to assign the results to a dimension value. However, equations (that is, expressions) in models differ in several ways from expressions used in other contexts. See "Rules for Equations in Models" on page 21-61 for information on using the assignment statement within models.

Assigning Values to Dimension Surrogates You assign values to a dimension surrogate with an = (SET) statement. For example, the following statements define the dimension surrogate storename, which is a TEXT type surrogate for the NUMBER type dimension store_id, assign a value to the fourth position of storename, and then report the value of the surrogate for the fourth value of store_id, which is 100.

```
DEFINE storename SURROGATE store_id TEXT
storename(storename 4) = 'Molly\'s Emporium'
REPORT W 25 storename(store_id 100)

STORENAME(STORE_ID 100)
-------------------------
Molly's Emporium
```

For example, when you define the INTEGER dimension surrogate `intsurr` for a NUMBER dimension `numdim` that has five values, then a report of `intsurr` produces the following.

```
INTSURR
-------
      1
      2
      3
      4
      5
```

Like a dimension, the values of a dimension surrogate must be unique. However, unlike a dimension, a dimension surrogate can have `NA` values, unless it is an

INTEGER type. The same value can be a value of the dimension and of any of its surrogates.

Assigning Values to Specific Cells of a Data Object

You can use a QDR with the target of an = (SET) statement. This lets you assign a value to specific cells in a variable or relation.

The following example assigns the value 10200 to the data cell of the `sales` variable that is specified in the qualified data reference. When the variable named `sales` does not already have a value in the cell associated with `Boston`, `Tents`, and `Jan99`, then the value is assigned to the cell and thus it is added to the variable. When a value already exists in the cell, the value 10200 overwrites the previous value.

```
sales(market 'Boston' product 'Tents' month 'Jan99')= 1020
```

Expressions Dimensioned Conjoint Dimensions

When an expression is dimensioned by a conjoint dimension, Oracle OLAP uses the dimension's relationship to its base dimension values to assign data to the correct cells. You can set the values of a variable dimensioned by a conjoint dimension to an expression dimensioned by one of its base dimensions. The converse is also true. See "Compacting Your Data" on page 21-69.

TEXT and NTEXT Source and Target

When the source is of type TEXT and the target is of type NTEXT, Oracle OLAP converts the TEXT value to NTEXT. Similarly, when the source is of type NTEXT and the target is of type TEXT, Oracle OLAP converts the NTEXT value to TEXT. Note that data can be lost when NTEXT is converted to TEXT.

Rules for Equations in Models

The equations in a model use an OLAP DML assignment statement to assign values to variables or dimension values. Equations in models differ in several ways from equations used in other contexts in Oracle OLAP:

- In a model equation, you can use the name of a dimension value anywhere you would normally use the name of a variable. You can base calculations on a dimension value, and you can assign the results of a calculation to a dimension value. When an equation refers directly to one or more dimension values, it is called a dimension-based equation.
- You cannot use ampersand substitution in model equations.

- You can include a program as a component in a calculation only when it is used as a function.
- Within a single dimension-based equation, all the dimension values must belong to the same dimension.
- When you assign the results of a calculation to a dimension value, the results must be numeric.
- Each dimension on which the model equations are based must be listed in a DIMENSION (in models) statement. When the model contains an INCLUDE command, the appropriate DIMENSION statements must be inherited from the included model. When the model does not contain an INCLUDE command, it must contain the appropriate DIMENSION statements. When you compile or run the model, Oracle OLAP searches through the dimensions listed in explicit or inherited DIMENSION statement to identify the dimension to which each dimension value belongs.

Dimension Status and Model Equations When a model contains an assignment statement to assigns data to a dimension value, then the dimension is limited temporarily to that value, performs the calculation, and then restores the initial status of the dimension.

Formatting Conjoint Dimension Values A special format is required when dimension-based equations refer to values of a conjoint dimension:

- Enclose the entire dimension value specification in angle brackets and then enclose this entire specification in single quotes; do not enclose the individual values in single quotes.
- Use the exact upper- and lowercase spellings for the base dimension values.
- When the specification includes a text value with an embedded blank, you must separate the dimension values with commas.

For example, assume that `item.org` is a conjoint dimension with base dimensions `item` and `org`. In this case, you use the following format to refer to values of `item.org`.

```
'<Expenses, Direct Sales>'
```

Formatting Text Dimension Values When dimension-based equations refer to text dimension values with embedded blanks or mixed upper- and lowercase letters, enclose the dimension value in single quotes. Use the exact upper- and lowercase spelling for the value.

For example, assume that a text dimension named `lineitem` contains a value with an embedded blank. In this case, you use the following format.

```
'Software Revenue'
```

Formatting DAY, WEEK, MONTH, QUARTER, YEAR Values When a model equation is based on a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR, you must use the dimension's VNF (value name format), rather than a date format, to specify the dimension's values. In addition, the VNF must format dimension values as follows:

- The value must start with a letter.
- The value can only contain letters, digits, underscores, and periods.

When the WEEK, MONTH, QUARTER, YEAR dimension of type does not have a VNF assigned to it, you can use the default VNF for the dimension. The entry for the VNF command lists the default VNF for each of these dimension types, and it explains how to assign a VNF to a dimension.

The default VNF for DAY dimensions is not acceptable because it specifies a digit as the first character of each dimension value. For a DAY dimension, specify the dimension name and enclose the value in parentheses and single quotes.

For example, for a DAY dimension named `daydim`, you can use the following format.

```
daydim('01jul97')
```

Formatting INTEGER Dimension Values When dimension-based equations refer to values of an INTEGER dimension, enclose the dimension value in single quotes.

For example, for an INTEGER dimension named `intdim`, use the following format to refer to the first dimension value.

```
'1'
```

When the model is based on more than one dimension, the model compiler might not be able to correctly identify the dimension to which a literal integer value belongs. In this case, specify the name of the dimension and enclose the value in parentheses and single quotes as described in "Formatting Ambiguous Dimension Values" on page 21-64.

Formatting Ambiguous Dimension Values In some cases the model compiler might be unable to correctly identify the dimension to which a dimension value belongs. For instance, this can happen under the following circumstances:

- Two or more dimensions have a dimension value with the same name.
- A DAY dimension uses the default VNF (which starts with a digit).
- An integer value could be interpreted either as a position within a dimension or as a literal integer value of a dimension.

In cases such as these, you can avoid ambiguity in model-based equations by following these rules:

- Enclose the dimension value in single quotes.
- Enclose the quoted value in parentheses.
- Precede the parentheses with the name of the dimension.

For example, for an INTEGER dimension named `intdim`, use the following format to refer to the first dimension value.

```
intdim('1')
```

Examples

Example 21–24 Assigning Values to a Variable

For the first example, suppose you have defined two variables, `units` and `price`, that are both dimensioned by `product`. The following example calculates dollar sales (`units` times `price`) for each value in the `product` dimension. Using an assignment statement, it stores the result in the variable `sales`, which is also dimensioned by `product`.

```
sales = units*price
```

For another example, assume the `choicedesc` variable is dimensioned by `choice`. Before you enter data for the variable, the cells of the variable contain only NA values.

```
CHOICE          CHOICEDESC
-------------- --------------------
Report         NA
Graph          NA
Analyze        NA
Data           NA
Quit           NA
```

Suppose you initialize the `choicedesc` variable using the following command.

```
choicedesc =  JOINCHARS ('Description for ' choice)
```

Now all of the `choicedesc` cells of the variable contain the appropriate values.

```
CHOICE            CHOICEDESC
-------------- -------------------------
Report         Description for Report
Graph          Description for Graph
Analyze        Description for Analyze
Data           Description for Data
Quit           Description for Quit
```

The next example shows an expression that is dimensioned by `time`, `product`, and `district` and is assigned to a new variable. The expression calculates a 2002 sales plan based on unit sales in 2001.

```
DEFINE units.plan INTEGER <month product district>
LIMIT month TO 'DEC02'
units.plan = LAG(units 12 month) * 1.15
```

Example 21–25 Assigning Values to a Relation

Assume that your analytic workspace contains the following definitiions for a hierarchical dimension for Geography named geog and a relation named geog_parentrel that contains values that represent the child-parent relationships in the Geography hierarchy.

```
DEFINE geog DIMENSION TEXT
DEFINE geog_parentrel RELATION geog <geog>
```

You can use the following MAINTAIN ADD statements to populate the hierarchical dimension.

```
" Populate the geog dimension with values for all levels
MAINTAIN geog ADD 'North America' 'Europe' 'United States' 'Canada' 'France'
'Germany'
MAINTAIN geog ADD 'Massachusetts' 'California' 'Quebec' 'Ontario'
MAINTAIN geog ADD 'Boston''Springfield' 'San Francisco''Los Angeles' 'Toronto'
'Ottawa'
MAINTAIN geog ADD 'Montreal''Quebec City' 'Paris' 'Marseilles' 'Bonn' 'Berlin'
```

You can use the following assignments statements to populate `geog_parentrel`. Note that you must limit `geog` to the appropriate values before you assign values to `geog_parentrel`.

```
"  Limit geog (and therefore geog_parentrel) to countries and assign
"   parent value (continent name) to those countries in geog_parentrel
LIMIT geog to 'United States' 'Canada'
geog_parentrel = 'North America'
LIMIT geog to ALL
LIMIT geog to 'France' 'Germany'
geog_parentrel = 'Europe'

"  Limit geog (and therefore geog_parentrel) to states or provinces and assign
"   parent value (country name) to those states or provinces in geog_parentrel
LIMIT geog to ALL
LIMIT geog to 'Massachusetts' 'California'
geog_parentrel = 'United States'
LIMIT geog to ALL
LIMIT geog to 'Quebec' 'Ontario'
geog_parentrel = 'Canada'
```

```
"  Limit geog (and therefore geog_parentrel) to cities and assign
"   parent value (state, province, or country) to those cities in geog_parentrel
LIMIT geog to ALL
LIMIT geog to 'Boston' 'Springfield'
geog_parentrel = 'Massachusetts'
LIMIT geog to ALL
LIMIT geog to 'San Francisco' 'Los Angeles'
geog_parentrel = 'California'
LIMIT geog to ALL
LIMIT geog to 'Montreal' 'Quebec City'
geog_parentrel = 'Quebec'
LIMIT geog to ALL
LIMIT geog to 'Toronto' 'Ottawa'
geog_parentrel = 'Ontario'
LIMIT geog to ALL
LIMIT geog to 'Paris' 'Marseilles'
geog_parentrel = 'France'
LIMIT geog to ALL
LIMIT geog to 'Bonn' 'Berlin'
geog_parentrel = 'Germany'
LIMIT geog to ALL
```

A report of `geog_parentrel` shows the values have been assigned.

```
COLWIDTH = 20
REPORT geog_parentrel
REPORT geog_parentrel

GEOG                GEOG_PARENTREL
---------------- --------------------
North America    NA
Europe           NA
United States    North America
Canada           North America
France           Europe
Germany          Europe
Massachusetts    United States
California       United States
Quebec           Canada
Ontario          Canada
Boston           Massachusetts
Springfield      Massachusetts
San Francisco    California
Los Angeles      California
Toronto          Ontario
Ottawa           Ontario
Montreal         Quebec
Quebec City      Quebec
Paris            France
Marseilles       France
Bonn             Germany
Berlin           Germany
```

Example 21–26 Using a Qualified Data Reference

This example uses an assignment statement with a qualified data reference to assign values to the variable `budget`. The values assigned to one budget line item (`Net.Income`) are calculated as the difference between two other line items (`Opr.Income` and `Taxes`), so you have to use a qualified data reference to obtain the correct data values.

```
budget(line Net.Income)= budget(line Opr.Income) - budget(line Taxes)
```

Example 21–27 Assigning Values to Variables with Composites

To have data assigned from sales only into existing data cells of sparse_sales, whose associated dimension values are in status, use the following command.

```
sparse_sales = sales ACROSS SPARSE<product market>
```

The ACROSS keyword is particularly helpful when the source expression is a single value. When there are no limits on the dimensions of sparse_sales, then an assignment command like the following creates cells for every combination of dimension values because there are no cases where the source expression is NA.

```
sparse_sales = 0
```

This defeats the purpose of a dimensioning a variable with a composite.

In contrast, the following command sets only existing cells of sparse_sales to 0 (zero).

```
sparse_sales = 0 ACROSS SPARSE<product market>
```

Example 21–28 Compacting Your Data

Suppose you only sell some of your products in each district. You currently have a variable sales that has data for certain combinations of districts and products and NA values for the rest. You can create a dense array of sales data by defining a composite or a conjoint dimension and using it as a dimension of a new variable. Use an assignment statement to assign the data directly to the new variable. When the values of the composite or conjoint dimension include all the combinations with data, you can then delete the original variable and save space in the analytic workspace.

```
DEFINE proddist DIMENSION <product district>
MAINTAIN proddist ADD <'Tents' 'Boston'> <'Canoes' 'Seattle'> -
   <'Sportswear' 'Atlanta'>
DEFINE sales.dense DECIMAL <month proddist>
sales.dense = sales
LIMIT month TO FIRST 4
```

Issuing a `REPORT sales.dense` statement produces the following output.

```
                    ----------------SALES.DENSE----------------
-----PRODDIST------ -------------------MONTH-------------------
PRODUCT    DISTRICT   Jan95      Feb95      Mar95      Apr95
-------- ---------- ---------- ---------- ---------- ----------
Tents      Boston    32,153.52  32,536.30  43,062.75  57,608.39
Canoes     Seattle   64,111.50  71,899.23  83,943.86  14,383.90
Sportswear Atlanta  114,446.26 123,164.92 138,601.64 141,365.66
```

An alternative method would be to use a composite instead of a conjoint dimension. In this case, you could use the following statements.

```
DEFINE sales.compact DECIMAL <month SPARSE <product district>>
sales.compact = sales
```

Oracle OLAP automatically creates the unnamed composite when you define `sales.compact`, and it automatically adds dimension value combinations to the composite when you use an assignment statement. Oracle OLAP creates dimension value combinations only for the non-`NA` values of `sales`.

SET1

The SET1 command assigns a single value to a variable, option, relation, or dimension surrogate. When an object has one or more dimensions, the SET1 command assigns the value to the object cell that is in current status.

Since the SET1 command does not loop through a dimensioned object, you can use it in Assign trigger programs to assign a value to an object.

Syntax

SET1 *target-name* = *expression*

Arguments

target-name
The name of the target object where the data will be assigned and stored. For a list of analytic workspace objects that can be a target object, see Table 21–6, " Use of Analytic Workspace Objects in OLAP DML Assignment (SET) Statement".

expression
The source of the data values to be assourcearget object, see Table 21–6, " Use of Analytic Workspace Objects in OLAP DML Assignment (SET) Statement"

Examples

For an example of using SET1, see Example 24–9, "Setting Values in an ASSIGN Trigger Program" on page 24-20.

SHOW

The SHOW command shows a single value of an expression. Normally, you would use SHOW to show the value of a single-cell variable or to show a message. SHOW is useful in programs when you want to generate an error-like message without creating an error condition. The output from SHOW is sent to the current outfile.

Syntax

SHOW *expression* [NONL]

Arguments

expression
The value you want to show. When *expression* is dimensioned, only the first value of the expression is shown, based on the current status of its dimensions. When you are showing a text literal, you must enclose the value in single quotes.

NONL
Indicates that a new line sequence should not be appended to the end of the value. By default, SHOW appends a new line sequence.

Notes

Concatenating Output Lines

The NONL argument to SHOW is useful in programs. Using this argument you can concatenate several values into a single line of output. To accomplish this, include one or more SHOW commands with the NONL argument, followed by a single SHOW command without the NONL argument. The values from all the SHOW commands are concatenated into a single output value, in the order specified. Depending on the length of the line, this value might actually produce more than one line of output.

Generating Error Messages

SHOW can be used as an alternative to SIGNAL when you want to generate an error message from a program. Unlike SIGNAL, SHOW produces a message without signaling an error condition and thus halting execution of the program. Your error message may be most useful when you send it to a debugging file. When you use the DBGOUTFILE command to direct messages to a debugging file, the output from SHOW is sent to the debugging file as well as to your current outfile.

Showing Values of Composites

When SHOW is used with a named or unnamed composite, an `NA` value is shown when the composite does not have a value that corresponds to the first values in the status for its base dimensions. For example, the statement

```
SHOW SPARSE <market product>
```

will produce an `NA` value when the combination of the current values of `market` and `product` does not exist in the composite.

Breaking Lines of Text

To break a text expression into two or more lines of output text, insert newline delimiters (\n) at the appropriate places in the text.

NTEXT Values

The SHOW command converts NTEXT values to the character set of the outfile. When an NTEXT value cannot be represented in the outfile character set, the character is not displayed correctly.

Examples

Example 21–29 Showing the Value of an Option

This example uses SHOW to report the current value of the DECIMALS option. The OLAP DML statement

```
SHOW DECIMALS
```

produces the following output.

```
2
```

Example 21–30 Showing a Data Value

When you use SHOW to report the value of a dimensioned variable, only the first value of the variable, based on the current status of its dimensions, is shown. The OLAP DML statement

```
SHOW JOINCHARS('Actual = ' actual)
```

produces the following output.

```
Actual = 533,362,88
```

Example 21–31 Creating Error Messages Using SHOW

When you want to produce a warning message without branching to an error label, then you can use the `SHOW` command.

```
select:
LIMIT month TO nummonths
IF STATLEN(month) GT 9
   THEN DO
     SHOW 'You can select no more than 9 months.'
     GOTO finish
     DOEND
REPORT DOWN district W 6 units
finish:
POP month
RETURN
```

SIGN

The SIGN function returns a value that indicates when a specified number is less than, equal to, or greater than 0 (zero).

Return Value

INTEGER. SIGN returns `-1` when `n<0`, `0` (zero) when `n=0`, or `1` when `n>0`.

Syntax

SIGN (*n*)

Arguments

n
A numeric expression.

Examples

The following example indicates that the function's argument (`-15`) is less than `0` (zero).

```
SHOW SIGN(-15)
 -1
```

SIGNAL

The SIGNAL command produces an error message from within a program and halts normal execution of the program. Oracle OLAP sends the error message to the current outfile. When the program contains an active trap label, execution branches to the label. Without a trap label, execution of the program terminates and, when the program was called by another program, execution control returns to the calling program.

Syntax

SIGNAL {*errname* [*message*]|STOP}

Arguments

errname
A TEXT expression that indicates the name of the error message to be produced. When you execute the SIGNAL command, Oracle OLAP stores the *errname* in the ERRORNAME option. Normally, the name of the error does not appear in the error message. However, when you omit *message,* the error name (*errname*) will appear along with a stock message as described in the *message* argument.

message
A TEXT expression that specifies the error message to be produced. When you omit this argument, SIGNAL produces the following message.

```
ERROR: (errname) Please contact the administrator of your
   Oracle Oracle OLAP application.
```

When you execute the SIGNAL command, Oracle OLAP stores *message* in the ERRORTEXT option.

STOP
Immediately stops execution of all currently running programs. No error message is produced. The error condition is not trapped by an active TRAP label.

Notes

Error Message Format

When you supply a long line as your error message, you must add your own line breaks to format the text. Type the newline escape sequence (\n) where you want each line to end. You can type up to a limit of 6 lines or 4000 characters, whichever you reach first. An error occurs when you try to supply a single line longer than 4000 characters.

Transfer of Control

SIGNAL creates an error condition that halts execution of a program. Control is passed back up any chain of nested programs until it reaches a trap label in one of the programs. See the TRAP command.

TRAP Labels

When you execute a SIGNAL command when TRAP is ON, execution branches to the trap label. Any statements following the trap label in the program are then executed.

PRGERR Argument

You can use the special name `PRGERR` to communicate to a calling program that an error has occurred. The statement `SIGNAL PRGERR` sets ERRORNAME to a blank value and passes an error condition to the calling program without causing another error message to be displayed. For a complete explanation of how to use SIGNAL to pass an error up a chain of nested programs, see the TRAP command.

Examples

Example 21–32 Signaling an Error

Suppose you have written a program that requires one argument. When no argument is supplied, there is no purpose in running the program. Therefore, the first thing the program does is check if an argument has been passed. When it has not, the program terminates after sending an error message to the current outfile.

The following program lines check for the argument and signal an error when it is not found.

```
IF ARGS EQ ''
THEN SIGNAL msg1 'You must supply an argument.'
```

SIGNAL sends the following message to the current outfile.

```
ERROR: You must supply an argument.
```

Example 21–33 Signaling an Error When an Argument Value is Invalid

Suppose your program produces a report that can present from one to nine months of data. You can signal an error when the program is called with an argument value greater than nine. In this example, `nummonths` is the name of the argument that must be no greater than nine.

```
select:
TRAP ON error
PUSH month
LIMIT month TO nummonths
IF STATLEN(month) GT 9
   THEN SIGNAL toomany -
     'You can specify no more than 9 months.'
REPORT DOWN district W 6 units
finish:
POP month
RETURN
error:
POP month
IF ERRORNAME EQ 'TOOMANY'
   THEN SHOW 'No report produced'
```

SIN

The SIN function calculates the sine of an angle expression. The result returned by SIN is a decimal value with the same dimensions as the specified expression.

Return Value

DECIMAL

Syntax

SIN(*angle-expression*)

Arguments

angle-expression
A numeric expression that contains an angle value, which is specified in radians.

Examples

Example 21–34 Calculating the Sine of an Angle in Radians

This example calculates the sine of an angle of 1 radian. The statements

```
DECIMALS = 5
SHOW SIN(1)
```

produce the following result.

```
0.84147
```

Example 21–35 Calculating the Sine of an Angle in Degrees

This example calculates the sine of an angle of 30 degrees. Since `1` degree `= 2*(pi)/360` radians, `30` degrees is about `30*2*3.14159/360` radians. The OLAP DML statement

```
SHOW SIN(30 * 2 * 3.14159 / 360)
```

produces the following result.

```
0.50000
```

SINH

The SINH function calculates the hyperbolic sine of an angle expression.

Return Value

DECIMAL

Syntax

SINH(*expression*)

Arguments

expression
A numeric expression that contains an angle value, which is specified in radians.

Examples

Example 21–36 Calculating the Hyperbolic Sine of an Angle

This example calculates the hyperbolic sine of an angle of 1 radian. The statements

```
DECIMALS = 5
SHOW SINH(1)
```

produce the following result.

```
1.17520
```

SLEEP

Within an OLAP DML program, the SLEEP command suspends the operation of Oracle OLAP for at least the specified number of seconds.

Syntax

SLEEP *n*

Arguments

n
A numeric expression that specifies the number of seconds for Oracle OLAP to "sleep." Program execution will be suspended for at least this number of seconds.

Notes

SLEEP Rarely Used

SLEEP is rarely used in Oracle OLAP programs, because there is seldom a need to suspend program execution.

Examples

Example 21–37 Suspending Program Execution

In a program, suppose you execute a statement that might take 10 seconds to complete. You can follow that statement with this SLEEP command, which suspends program execution for 10 seconds.

```
SLEEP 10
```

SMALLEST

The SMALLEST function returns the smallest value of an expression. You can use this function to compare numeric values or date values.

Return Value

The data type of the expression. It can be INTEGER, LONGINT, DECIMAL, or DATE.

Syntax

SMALLEST(*expression* [[STATUS] *dimensions*])

Arguments

expression
The expression whose smallest value is to be returned.

STATUS
Can be specified when one or more of the dimensions of the result of the function are not dimensions of the expression. (See the description of the *dimensions* argument.) When you specify the STATUS keyword when this is not the case, Oracle OLAP produces an error.

In cases where one or more of the dimensions of the result of the function are not dimensions of the expression, the STATUS keyword might be *required* in order for Oracle OLAP to process the function successfully, or the STATUS keyword might provide a performance enhancement. See "The STATUS Keyword" on page 21-83.

dimensions
The dimensions of the result. By default, SMALLEST returns a single value. When you indicate one or more dimensions for the results, SMALLEST calculates the smallest value along the dimensions that are specified and returns an array of values. Each dimension must be either a dimension of *expression* or related to one of its dimensions. When it is a related dimension, you can specify the name of the relation instead of the dimension name. This enables you to choose which relation is used when there is more than one.

Notes

NA Values

SMALLEST is affected by the NASKIP option. When NASKIP is set to `YES` (the default), SMALLEST ignores `NA` values and returns the smallest value or values that are not `NA`. When NASKIP is set to `NO`, SMALLEST returns `NA` when any value of the expression is `NA`. When all the values of the expression are `NA`, SMALLEST returns `NA` for either setting of NASKIP.

Calculating over a Time Dimension

When *expression* is dimensioned by dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR, you can specify any other DAY, WEEK, MONTH, QUARTER, or YEAR dimension as a related *dimension.* Oracle OLAP uses the implicit relation between the dimensions. To control the mapping of one DAY, WEEK, MONTH, QUARTER, or YEAR dimension to another (for example, from weeks to months), you can define an explicit relation between the two dimensions and specify the name of the relation as the *dimension* argument to the SMALLEST function.

For each time period in the related dimension, Oracle OLAP finds the smallest data value in any source time period that ends in the target time period. This method is used regardless of which dimension has the more aggregate periods.

The STATUS Keyword

When one or more of the dimensions of the result of the function are not dimensions of the expression, Oracle OLAP creates a temporary variable to use while processing the function. When you specify the STATUS keyword, Oracle OLAP uses the current status instead of the default status of the related dimensions for calculating the size of this temporary variable.

When the size of the temporary variable for the results of the function would exceed 2 gigabytes, you *must* specify the STATUS keyword in order for Oracle OLAP to execute the function successfully. When the dimensions of the expression are limited to a few values and are physically fragmented, you can specify the STATUS keyword to improve the performance of the function.

When you use SMALLEST with the STATUS keyword for an expression that requires going outside of the status for results (for example, with the LEAD or LAG functions or with a qualified data reference), the results outside of the status will be returned as `NA`.

Examples

Example 21–38 Finding the Month with the Least Amount of Sportswear Sales

This example uses the SMALLEST function to find the smallest monthly sportswear sales for three districts during the first half of 1996. To see the smallest sales figure for each district, specify `district` as the dimension of the results.

```
LIMIT product TO 'Sportswear'
LIMIT district TO FIRST 3
LIMIT month TO 'Jan96' TO 'Jun96'
REPORT HEADING 'Smallest Sales' SMALLEST(sales district)
```

The preceding statements produce the following output.

```
                Smallest
DISTRICT         Sales
-------------- ----------
Boston          57,079.10
Atlanta        129,616.08
Chicago         77,489.51
```

SMOOTH

The SMOOTH function computes a single or a double exponential smoothing of a numeric expression.

Return Value

DECIMAL

Syntax

SMOOTH(*expression* {SINGLE *alpha*|DOUBLE *alpha beta m*} [BASEDON *dimension-list*])

Arguments

expression
The numeric expression whose values are to be smoothed.

SINGLE
DOUBLE
The method to use in the exponential smoothing of the values in *expression*. The SINGLE method specifies single exponential smoothing and requires an *alpha* argument. The DOUBLE method specifies double exponential smoothing (also known as Holt's linear exponential smoothing) and requires an *alpha* argument, a *beta* argument, and an *m* argument.

alpha
A number in the range from 0 to 1 that smooths the difference between the observed data forecast and the last forecast. The higher the value, the more weight the most recent forecast has, so smoothing decreases as the smoothing factor increases. A smoothing factor of 0 completely smooths the forecasts and always returns the first forecast, which is the first data observation. A smoothing factor of 1 produces no smoothing at all and returns the previous data observation. (See "Results of alpha Values" on page 21-86.)

beta
A number in the range from 0 to 1 that smooths the difference between the previous forecast and the current forecast. As with the *alpha* argument, smoothing decreases as the smoothing factor increases.

m

A positive integer between 1 and the total number of periods of data in the data series. The *m* argument specifies the number of periods on which to base the forecasts.

BASEDON *dimension-list*

An optional list of one or more of the dimensions of *expression* to include in the exponential smoothing. When you do not specify the dimensions, then SMOOTH bases the smoothing on all of the dimensions of *expression*.

Notes

The Effect of NASKIP

SMOOTH is affected by the NASKIP option. When NASKIP is set to `YES` (the default), then SMOOTH ignores `NA` values. When NASKIP is set to `NO`, then SMOOTH returns `NA` for every forecast after the `NA` value.

Results of *alpha* Values

This note illustrates the results of using different *alpha* values for single exponential smoothing. The results are based on the `sales` variable with the dimensions limited by the following statements.

```
LIMIT month TO 'Jan96' TO 'Dec96'
LIMIT product TO 'Tents'
LIMIT district TO 'Boston'
REPORT DOWN month SMOOTH(sales, SINGLE, ALPHA, BASEDON month)
```

The following table shows the data values of the `sales` variable and also shows the results of the SMOOTH function in the preceding statement when the *alpha* argument variable has the different values shown in the table.

MONTH	**Sales of tents in Boston**	**alpha = 0**	**alpha = .1**	**alpha = .5**	**alpha = .9**
Jan96	50,808.96	NA	NA	NA	NA
Feb96	34,641.59	50,808.96	50,808.96	50,808.96	50,808.96
Mar96	45,742.21	50,808.96	49,192.22	42,725.28	36,258.33
Apr96	61,436.19	50,808.96	48,847.22	44,233.74	44,793.82
May96	86,699.67	50,808.96	50,106.12	52,834.97	59,771.95
Jun96	95,120.83	50,808.96	53,765.47	69,767.32	84,006.90
Jul96	93,972.49	50,808.96	57,901.01	82,444.07	94,009.44
Aug96	94,738.05	50,808.96	61,508.16	88,208.28	93,976.18
Sep96	75,407.66	50,808.96	64,831.15	91,473.17	94,661.86
Oct96	70,622.91	50,808.96	65,888.80	83,440.41	77,333.08
Nov96	46,124.99	50,808.96	66,362.21	77,031.66	71,293.93
Dec96	36,938.27	50,808.96	64,338.49	61,578.33	48,641.88

Examples

Example 21–39 Smoothing Values

These statements limit the dimensions of the `sales` variable, set the data column width for reports, and report the data values for `sales`.

```
LIMIT month TO 'Jan96' TO 'Dec96'
LIMIT product TO 'Tents'
LIMIT district TO 'Boston'
COLWIDTH = 14

REPORT W 6 DOWN month sales
```

The preceding statements produce the following output.

```
DISTRICT: Boston
       ----SALES-----
       ---PRODUCT----
MONTH      Tents
------ --------------
Jan96       50,808.96
Feb96       34,641.59
Mar96       45,742.21
Apr96       61,436.19
...
Nov96       46,124.99
Dec96       36,938.27
```

This statement reports the results of using the SMOOTH function on the `sales` variable with the SINGLE method, a data smoothing factor of `.5`, and based on the `month` dimension.

```
REPORT W 6 DOWN month SMOOTH(sales, SINGLE, .5, BASEDON month)
```

The preceding statement produces the following output.

```
DISTRICT: Boston
       SMOOTH(SALES,-
       -SINGLE, .5,--
       BASEDON MONTH)
       ---PRODUCT----
MONTH      Tents
------ --------------
Jan96              NA
Feb96       50,808.96
Mar96       42,725.28
Apr96       44,233.74
...
Nov96       77,031.66
Dec96       61,578.33
```

SORT

The SORT command arranges the order of values in the current status list of a dimension or a dimension surrogate, or in a valueset.

Syntax

SORT *dimension* {A|D} *criterion1* [{A|D} *criterionN*]

Arguments

dimension

A text expression whose value is the name of a dimension, a dimension surrogate, or a valueset.

A

D

The order in which the values are to be sorted. `A` means ascending order (alphabetical when the sorting criterion is TEXT, ID, or a relation), and `D` means descending order (reverse alphabetical when the sorting criterion is TEXT, ID or a relation).

criterion

The expression to be used as a sorting criterion. Each *criterion* must be dimensioned by *dimension.* The first expression is the major sorting criterion. When the expression is multidimensional, SORT uses the first value in status for all dimensions other than the dimension being sorted. You cannot use a valueset as the sorting criterion.

Notes

Sorting a Dimension and a Valueset

When Oracle OLAP sorts a dimension, it sorts the temporary status list of a dimension, not the data dimensioned by it. Since many OLAP DML statements operate on data according to the current status of its dimensions, sorting a dimension *appears* to have the effect of sorting data. A dimension and any dimension surrogates for it share the same status. Therefore, a SORT command on a dimension or any of its surrogates sorts them all.

When Oracle OLAP sorts a valueset, it sorts the actual values within the valueset. When you execute UPDATE and COMMIT commands after sorting a valueset, the values in the valueset are stored in that sorted order.

Sorting Alphabetically

To sort a TEXT or ID dimension or its valueset in alphabetical order, use the dimension itself as the sorting criterion.

```
SORT district A district
```

Sort Order

The sort order for textual data in an alphabetical sort is controlled by the NLS_SORT option.

Sorting a Time Dimension

The values of dimensions of type DAY, WEEK, MONTH, QUARTER, and YEAR are stored internally as numbers. Therefore, when you sort a dimensions of type DAY, WEEK, MONTH, QUARTER, and YEAR dimension or its valueset in ascending order, with the dimension itself as the sorting criterion, then the values in the status list or valueset are placed in chronological order. When you sort a dimensions of type DAY, WEEK, MONTH, QUARTER, and YEAR dimension or its valueset in descending order, then the values are placed in reverse chronological order.

Sorting Based on a Relation

When you use a relation as your sorting criterion, then the sorting is done alphabetically; that is, the dimension or valueset is sorted according to an alphabetical list of the related dimension values. To use a relation as the sorting criterion and keep the related dimension values in their original order, you must use the following expression as your sorting criterion See Example 21–40, "Sorting Based on a Relation" on page 21-92.

```
CONVERT(relation, INTEGER)
```

Sorting Conjoint Dimensions

You can sort a conjoint dimension or its valueset by criteria dimensioned by either the conjoint dimension itself or by one of its base dimensions.

Sorting Concat Dimensions

You can sort a concat dimension or its valueset by criteria dimensioned by either the concat dimension itself or by one of its component dimensions. See Example 21–41, "Sorting Based on a Concat" on page 21-93 and Example 21–42, "Sorting Based on a Component" on page 21-94.

Sorting a Worksheet

You cannot use a worksheet as a sort criterion. You must first use CONVERT to specify the data type to which values of the worksheet should be converted.

Examples

Example 21–40 Sorting Based on a Relation

This example sorts districts according to their unit sales of tents for July 1996. They are sorted first by the region to which they belong and then in descending order of dollar sales. Notice that in the following SORT command, a relation is used as the primary sorting criterion. This means that the districts are sorted by regions listed alphabetically.

```
LIMIT month TO 'Jul96'
LIMIT product TO 'Tents'
SORT district A Region.District D sales
```

Assume you issue the following REPORT command.

```
REPORT DOWN district HEADING 'Region' region.district sales
```

The preceding statement produces the following report that reflects the work of the SORT command.

```
PRODUCT: Tents
               --------MONTH--------
               --------JUL96--------
DISTRICT         Region     SALES
-------------- ---------- ----------
Dallas         Central    154,914.23
Chicago        Central     79,934.42
Atlanta        East       140,711.00
Boston         East        93,972.49
Seattle        West       123,700.17
Denver         West       100,413.49
```

In the following SORT command, CONVERT is used to keep the regions in their original order.

```
SORT district A CONVERT(region.district INTEGER) D sales
```

Assume that you issue the following REPORT statement.

```
REPORT DOWN district HEADING 'Region' region.district sales
```

The preceding statement produces the following report that reflects the work of the last SORT command.

```
PRODUCT: Tents
               --------MONTH--------
               --------JUL96--------
DISTRICT         Region     SALES
-------------- ---------- ----------
Atlanta        East       140,711.00
Boston         East        93,972.49
Dallas         Central    154,914.23
Chicago        Central     79,934.42
Seattle        West       123,700.17
Denver         West       100,413.49
```

When you want the dimension to keep the sorted order of its values permanently, use the MAINTAIN command after you sort the dimension.

```
SORT district A district
MAINTAIN district MOVE VALUES(district) FIRST
```

Example 21–41 Sorting Based on a Concat

The following statements sort the concat dimension `reg.dist.ccdim` in ascending order based on all of its values and report the result.

```
sort reg.dist.ccdim d reg.dist.ccdim
report reg.dist.ccdim
```

The preceding statement produces the following results.

```
REG.DIST.CCDIM
--------------------
<Region: West>
<Region: East>
<Region: Central>
<District: Seattle>
<District: Denver>
<District: Dallas>
<District: Chicago>
<District: Boston>
<District: Atlanta>
```

The following statements sort the concat dimension `reg.dist.ccdim` in ascending order based on all of its values and report the result.

```
SORT reg.dist.ccdim A reg.dist.ccdim
REPORT reg.dist.ccdim
```

The preceding statement produces the following results.

```
REG.DIST.CCDIM
--------------------
<District: Atlanta>
<District: Boston>
<District: Chicago>
<District: Dallas>
<District: Denver>
<District: Seattle>
<Region: Central>
<Region: East>
<Region: West>
```

Example 21–42 Sorting Based on a Component

The following statements sort the concat dimension `reg.dist.ccdim` in ascending order based on the values of one of its base dimensions and in descending order based on the values of its other base dimension, and report the result.

```
SORT reg.dist.ccdim A region D district
REPORT reg.dist.ccdim
```

The preceding statement produces the following results.

```
REG.DIST.CCDIM
--------------------
<REGION: CENTRAL>
<REGION: EAST>
<REGION: WEST>
<DISTRICT: SEATTLE>
<DISTRICT: DENVER>
<DISTRICT: DALLAS>
<DISTRICT: CHICAGO>
<DISTRICT: BOSTON>
<DISTRICT: ATLANTA>
```

SORTCOMPOSITE

The SORTCOMPOSITE option indicates whether Oracle OLAP should perform sorting on composite values when you issue a statement, such as REPORT, that explicitly loops over the composite. The sorting brings the composite values in line with the current order of the composite's base dimension values.

By default, SORTCOMPOSITE is set to `YES`, and Oracle OLAP performs the required sorting. You set SORTCOMPOSITE to `NO` only when you do not care how composite values are sorted and you want to save the processing time Oracle OLAP would have spent on the sorting.

SORTCOMPOSETE affects Oracle OLAP behavior only when you have explicitly specified that looping should occur over a composite, for example when you specify the composite name after a DOWN or ACROSS keyword in a REPORT command. Of course, when the composite has already been sorted according to the current order of its base dimensions values, Oracle OLAP does not unnecessarily sort the values again.

Syntax

SORTCOMPOSITE = {YES|NO}

Arguments

YES
In an explicitly specified loop over a composite, Oracle OLAP sorts the composite values according to the order of the composite's base dimension values (when they have not already been sorted in this way). The task of sorting requires some processing time, so when variables are large, performance can be affected. (Default)

NO
In an explicitly specified loop over a composite, Oracle OLAP does not sort the composite values according to the order of the composite's base dimension values. Eliminating this sorting step can improve Oracle OLAP performance, when large variables are involved. See "Results with SORTCOMPOSITE Set to NO" on page 21-97.

Notes

Results with SORTCOMPOSITE Set to NO

When SORTCOMPOSITE is set to `NO`, the sort order of the composite value is undefined. It is the order that demands the least processing effort from Oracle OLAP, so it depends on the activities that have preceded the statement that requires the looping. The order will differ from session to session and from time to time within a session. It is not necessarily the default order for the values of the composite.

Examples

Example 21–43 Sorting on a Composite

In the following example, a variable called `coupon_count` holds the number of coupons that were redeemed for certain products in certain districts. `coupon_count` is dimensioned by a composite called `coupon_composite`, which holds the combinations of products and districts for which coupons were distributed.

```
DEFINE coupon_composite COMPOSITE <product district>

DEFINE coupon_count VARIABLE  -
   INTEGER <month coupon_composite <product district>>
```

Assume that you issue the following statements.

```
SORTCOMPOSITE = YES
LIMIT month TO FIRST 1
SORT product D TOTAL(coupon_count, product)
REPORT DOWN coupon_composite W 15 coupon_count
```

With SORTCOMPOSITE set to `YES`, and after the following LIMIT and SORT commands, the preceding REPORT command produces the following report. Notice that the products are listed in decending order according to the *total* of Boston and Chicago figures for each product.

```
                      -COUPON_COUNT--
                      -----MONTH-----
 PRODUCT    DISTRICT       Jan95
---------- ---------- ---------------
Racquets   Boston                  93
Tents      Boston                  42
Canoes     Boston                  67
Sportswear Boston                  29
Racquets   Chicago                102
Tents      Chicago                 51
```

When SORTCOMPOSITE had been set to `NO`, Oracle OLAP would not necessarily have looped over the `product` dimension according to the sorted values of `coupon_count`. The looping order would have been the order that required the least processing effort from Oracle OLAP. If `coupon_count` had been a very large variable, the performance improvement might have been significant.

SORTLINES

The SORTLINES function sorts the lines in a multiline TEXT value.

Return Value

TEXT or NTEXT

Syntax

SORTLINES(*text-expression* [A|D])

Arguments

text-expression
A multiline text expression whose lines SORTLINES sorts. When you specify a TEXT expression, the return value is TEXT. When you specify an NTEXT expression, the return value is NTEXT.

A
D
Specifies whether the sorting order should be *ascending*, or alphabetical (**A**), or *descending*, or reverse alphabetical (**D**). The default is **A** (ascending).

Notes

Sort Order

The sort order is controlled by the NLS_SORT option.

Examples

Example 21–44 Sorting Text Lines

This example shows how to sort the lines in a multiline text value in a variable called MKTREGIONS.

The statement

```
SHOW mktregions
```

produces the following output.

```
New York
Boston
Atlanta
San Francisco
```

The statement

```
SHOW SORTLINES(mktregions)
```

produces the following output.

```
Atlanta
Boston
New York
San Francisco
```

SPARSEINDEX

The SPARSEINDEX option controls the type of index algorithm that composites use to load and access their values. The value of SPARSEINDEX at the time a named composite is defined, or an unnamed composite is created, determines the type of algorithm the composite uses by default. See "Overriding the Default" on page 21-102.

Choosing an index algorithm is important only in regard to performance issues. Any recommendations are for the version of Oracle OLAP that is associated with this documentation. You can test how using different algorithms affect performance by using the CHGDFN command to change the algorithm for a composite (for example, before loading data).

Data type

TEXT

Syntax

SPARSEINDEX = {'<u>BTREE</u>'|'HASH'}

Arguments

BTREE
HASH
Specifies the index algorithm that Oracle OLAP uses to load and access the values of new composites that are defined or created. BTREE is the default algorithm.

Notes

When to Use BTREE

BTREE is a standard indexing method that is recommended for composites. Use BTREE as the default unless you are an advanced user and have a special need that requires HASH. BTREE tends to group similar values together, which results in better locality of access.

When to Use HASH

HASH is a standard indexing method that should only be used when a composite has only two or three base dimensions. HASH is generally not recommended for composites. Using HASH results in a very large index table, which can be too large to fit into memory.

Overriding the Default

When you define a named composite, you can specify either BTREE or HASH as its index algorithm. When you specify an index algorithm in the DEFINE COMPOSITEcommand, this overrides the default specified by the SPARSEINDEX option. After you have defined a composite, you can also use the CHGDFN command to change the composite's index algorithm to either BTREE or HASH.

NOHASH Unavailable

A composite cannot use the NOHASH index algorithm for loading and accessing its values.

Examples

Example 21–45 Using the HASH Algorithm

The following example sets SPARSEINDEX to `HASH` so that composites that are subsequently defined or created will use the HASH index algorithm by default.

```
SPARSEINDEX = 'HASH'
```

22
SQL to STATVAL

This chapter contains the following OLAP DML statements:

SQL

The SQL command passes instructions written in Structured Query Language (SQL) to the relational manager from Oracle OLAP. Using the SQL command, you can insert and update data in relational tables, retrieve data from relational tables into analytic workspace objects, and execute stored procedures.

To use the SQL command, you must be familiar with SQL syntax and with the data structures in your relational database. SQL*Plus Worksheet and Oracle Enterprise Manager can be useful tools for browsing through your database.

This entry describes the SQL command in general, and subsequent entries discuss the use of the OLAP DML SQL command for specific SQL statements:

- SQL CLEANUP
- SQL CLOSE.
- SQL DECLARE CURSOR
- SQL EXECUTE
- SQL FETCH
- SQL IMPORT
- SQL OPEN
- SQL PREPARE
- SQL PROCEDURE
- SQL SELECT

Syntax

SQL *sql-statement*

Arguments

sql-statement

For *sql-statement* you can specify most SQL statements that can be executed dynamically, as well as several associated non-dynamic statements.You can also specify PROCEDURE for a stored procedure as described in SQL PROCEDURE.

You can*not* specify the following SQL statements for *sql-statement* :

- `COMMIT` -- To commit your changes, issue the OLAP DML COMMIT command.
- `ROLLBACK` -- You cannot rollback using the OLAP DML. When you specify SQL `ROLLBACK`, you receive an error message stating that `ROLLBACK` is not supported as an argument to the OLAP DML SQL command.

> **Important:** When you use the OLAP DML SQL command to request a rollback in some other fashion (for example, using SQL EXECUTE), Oracle OLAP issues a system error message, abnormally terminates the OLAP DML program that issued the command. Oracle OLAP also detaches, in an indeterminate state, the analytic workspace that contains the OLAP DML program that made the rollback request and any other attached analytic workspaces with uncommitted updates.

Oracle OLAP evaluates some SQL statements before sending them to the relational manager. For example, Oracle OLAP evaluates SQL PREPARE and SQL EXECUTE, and SQL statements that copy data from relational tables into analytic workspace objects (See "Copying Relational Data into Analytic Workspace Objects" on page 22-4 for a list of these statements).

Notes

Copying Relational Data into Analytic Workspace Objects

You can copy relational data into analytic workspace objects using either an implicit cursor or an explicit cursor:

- To copy data from relational tables into analytic workspace objects using an implicit cursor, use the SQL SELECT command. You can use this OLAP DML command interactively in the OLAP Worksheet or within an OLAP DML program.
- To copy data from relational tables into analytic workspace objects using an explicit cursor, use the following commands within an OLAP DML program in the order indicated:

  1. SQL DECLARE CURSOR defines a SQL cursor by associating it with a SELECT statement or procedure.
  2. SQL OPEN activates a SQL cursor.

3. SQL FETCH and SQL IMPORT retrieve and process data specified by a cursor.
4. SQL CLOSE closes a SQL cursor.
5. SQL CLEANUP cancels all SQL cursor declarations and frees the memory resources of an SQL cursor.

Oracle OLAP evaluates all of these statements before sending them to the relational manager.

For the syntax of these statements, see the individual topics. For the syntax of other SQL statements, refer to the *Oracle Database SQL Reference*.

SQL Embed Options

A number of options are available to you when embedding SQL. These options are listed in Table 22–1, " Embedded SQL Options".

Table 22–1 Embedded SQL Options

Statement	Description
SQLBLOCKMAX	An option that contains the maximum number of records retrieved from an Oracle relational database at one time.
SQLCODE	(Read-only) An option that contains the value returned by the Oracle RDBMS after the most recently attempted SQL operation.
SQLERRM	(Read-only) After the Database reports an error and SQLCODE has a nonzero value, an option that contains the text that explains the problem.
SQLMESSAGES	An option that determines whether error messages are sent to the current output file.

Software Support

Before you use the SQL command, ensure that you have access rights to the tables that you want to use.

SQL Terminology

In this topic, OLAP DML is the "host language," an OLAP DML program is a "host program," and an OLAP DML variable used within a SQL statement is a "host variable." There are two types of host variables: input host variables and output host variables. Host variable names must be preceded by a colon (for example, :MYVAR).

Input Host Variables

Input host variables are values supplied by Oracle OLAP as parameters to a SQL statement. They specify the data to be selected or provide values for data that is being modified.

You can use input host variables in SQL `WHERE` clauses, parameter list for procedures, `UPDATE` statements, and the value clause of `INSERT`.

When you specify a dimension or a dimensioned variable as an input host variable, the first value in status is used; no implicit looping occurs, although you can use a FOR or ACROSS command to loop through all of the values. An input host variable can be any expression with an appropriate data type. The value of an input host variable is taken when a cursor is opened, not when it is declared. See Example 22–1, "Inserting Data in a Table" on page 22-7.

To update or insert relational CLOB and NCLOB data, you use `WIDE` in the host variable for the OLAP DML expression as described in "Inserting Large Text Values into a Table" on page 22-40.

Error Checking

Oracle OLAP can detect some syntax errors in the arguments to the SQL statement, but most errors are detected by the Oracle RDBMS. Error codes and messages are returned to Oracle OLAP. You should check the value of SQLCODE after each SQL statement to determine when it resulted in an error. When it does cause an error (that is when `SQLCODE EQ -1`), check the value of SQLERRM for information about the cause of the error.

WHERE CURRENT OF *cursor*

`SQL UPDATE` statements can contain a `WHERE` clause, which specifies a particular search condition. In addition to the search conditions typically used in SQL, the phrase `WHERE CURRENT OF` *`cursor`* is supported for single tables and views that include columns from only one table. The cursor must have been defined with the `FOR UPDATE` clause.

Inserting Data into a Relational Table

Refer to the notes for SQL PREPARE and SQL EXECUTE.

Length Restriction

A SQL statement cannot exceed 128K bytes including the values of all non-text input host variables.

Examples

Example 22–1 Inserting Data in a Table

You can use SQL statements such as the following to create a table and add rows to that table. The SQL INSERT statement adds a row to the `sales` table using values from the dimension `salesperson` and the variable `dollars`. It adds one row using the first value of `salesperson` that is in status.

```
SQL CREATE TABLE sales (name CHAR(12), dollars INTEGER)
SQL INSERT INTO sales VALUES (:salesperson, :dollars)
```

SQL CLEANUP

The SQL CLEANUP command cancels all SQL cursor declarations and frees the memory resources for all SQL cursors. After you have cancelled SQL cursors, you cannot use them again unless you issue new SQL PREPARE, SQL DECLARE CURSOR, and SQL OPEN commands.

Syntax

```
SQL CLEANUP
```

Notes

Related OLAP DML Commands

You use the SQL CLEANUP command in combination with other SQL commands to copy data from relational tables into analytic workspace objects as outlined in "Copying Relational Data into Analytic Workspace Objects" on page 22-4.

Examples

For an example of the use of SQL CLEANUP, see Example 22–11, "Fetching Data into a Concat Dimension" on page 22-26.

SQL CLOSE

The SQL CLOSE command closes a SQL cursor. After you have closed a cursor, you cannot use it again unless you issue a new SQL OPEN command.

Syntax

SQL CLOSE *cursor*

Arguments

cursor
The name of a cursor previously opened with a SQL OPEN command.

Notes

Related OLAP DML Commands

You use the SQL OPEN command in combination with other SQL commands to copy data from relational tables into analytic workspace objects as outlined in "Copying Relational Data into Analytic Workspace Objects" on page 22-4.

Redefining the Result Set

You can change the result set associated with a cursor by closing the cursor, setting the value of an input host variable, and issuing a new SQL OPEN command. You do not have to free the cursor and redeclare it.

SQL DECLARE CURSOR

The SQL DECLARE CURSOR command defines an explicit SQL cursor by associating it with a `SELECT` statement or procedure. The `SELECT` statement specifies the scope of the data (the rows and columns) selected by the cursor.

Two pseudo procedures, `SQLTABLES` and `SQLCOLUMNS`, allow you to obtain information about tables and columns.

Syntax

SQL DECLARE *cursor* CURSOR FOR {*select-statement* [FOR UPDATE]|*table-info*}

where *table-info* can be used only when declaring a cursor for use by the SQL FETCH command and is one of the following:

PROCEDURE SQLTABLES [*owner*, *table*]

PROCEDURE SQLCOLUMNS [*owner*, *table*, *column*]

Arguments

cursor
The name of the cursor you are defining. See "Cursor Names" on page 22-12.

select-statement
A SQL SELECT statement that identifies the data you want to associate with the cursor. For the syntax of an SQL SELECT statement, refer to *Oracle Database SQL Reference.*

FOR UPDATE
Indicates that SQL FETCH will be used to write data to the table. This clause is required when the cursor will be used in an `UPDATE` statement with a `WHERE CURRENT OF` *`cursor`* clause. The names of the columns to be updated can be listed in an `OF` clause (for example, `FOR UPDATE OF COL1, COL2, COL3`).

Note: The FOR UPDATE clause is ignored by SQL IMPORT and SQL SELECT.

PROCEDURE SQLTABLES

When declaring a cursor for use by SQL FETCH, calls the pseudo procedure `SQLTABLES`, which returns information about tables. When declaring a cursor for use by SQL IMPORT, you cannot use this clause.

PROCEDURE SQLCOLUMNS

When declaring a cursor for use by SQL FETCH, calls the pseudo procedure `SQLCOLUMNS`, which returns information about columns. When declaring a cursor for use by SQL IMPORT, you cannot use this clause.

owner

Literal text or the name of a host variable whose value specifies one or more owners. This expression acts as a filter to limit the results to only tables belonging to the specified owners. The keyword NULL or a host variable with an `NA` value causes all table owners to be included in the results.

The expression can be specific, such as `'SCOTT'`, or it can contain wildcard characters such as `'S%T'` (all owners whose name begins with `S` and ends with `T`). The value retains its case when it is passed to the database, so be sure to enter the value with the appropriate use of upper- and lowercase letters. For example, Oracle relational databases by default store all values in uppercase and will not match `'scott'` or `'Scott'` with `'SCOTT'`.

table

Literal text or the name of a host variable whose value specifies one or more tables. This expression acts as a filter to limit the results to only tables with the specified names. The keyword NULL or a host variable with an `NA` value causes all tables to be included in the results.

The expression can be specific, such as `'PAYROLL'`, or it can contain wildcard characters such as `'%ROLL'` (all tables whose name ends with `ROLL`). The value retains its case when it is passed to the database, so be sure to enter the value with the appropriate use of upper- and lowercase letters. For example, Oracle relational databases by default store all values in uppercase and will not match `'payroll'` or `'Payroll'` with `'PAYROLL'`.

column

Literal text or the name of a host variable whose value specifies one or more columns. This expression acts as a filter to limit the results to only columns with the specified names. The keyword `NULL` or a host variable with an NA value causes all tables to be included in the results.

The expression can be specific, such as `'SALARY'`, or it can contain wildcard characters such as `'SAL%'` (all columns whose name begins with `SAL`). The value retains its case when it is passed to the database, so be sure to enter the value with the appropriate use of upper- and lowercase letters. For example, Oracle relational databases by default store all values in uppercase and will not match `'salary'` or `'Salary'` with `'SALARY'`.

Notes

Related OLAP DML Commands

You use the SQL DECLARE CURSOR command in combination with other SQL commands to use an explicit cursor to copy data from relational tables into analytic workspace objects as outlined in "Copying Relational Data into Analytic Workspace Objects" on page 22-4.

General Restrictions

The following restrictions apply to the SQL DECLARE CURSOR command:

- You can use it only in a program.
- It cannot contain ampersand substitution.

Restrictions when Declaring a Cursor for Use by SQL IMPORT

When declaring a cursor to be used by the SQL IMPORT command, you can only use the following simplified syntax.

SQL DECLARE *cursor* CURSOR FOR *select-statement*

where *select-statement* is a SQL `SELECT` statement that identifies the data you want to associate with the cursor. You cannot use the `FOR UPDATE` clause or the *`table-info`* clause.

Cursor Names

Cursor names can consist of 1 to 18 alphanumeric characters or the symbols `@`, `_`, `$`, or `#`. A name that contains symbols `@`, `$`, or `#` must be enclosed in single quotes. The first character cannot be a number or an underscore. Cursor names are internal to Oracle OLAP. Unless you have issued a SQL CLEANUP statement, when you try to declare a cursor with the same name as a previously declared cursor, but with a different SQL `SELECT` command, an error is signaled.

Cursor's Result Set

A cursor's result set is determined at the time it is opened, and it is not updated later. Therefore, when you change the value of an input host variable after you open its cursor, the change does not affect the cursor's result set. A cursor remains open until a SQL CLOSE command is executed for that cursor or until a SQL CLEANUP command closes all cursors. A cursor is not automatically closed at the termination of the program in which it was opened.

Optimizing Fetches

When fetching values into a multidimensional input variable, list the columns that correspond to the dimensions in an ORDER BY clause in the *select-statement* argument of the SQL DECLARE CURSOR command, with the slowest-varying dimension first. This will optimize performance.

Ambiguous WHERE Clauses

The *select-statement* argument of a SQL DECLARE CURSOR command can include a WHERE clause. Since both OLAP DML syntax and SQL syntax allow you to use AND and OR, you should construct the clause clearly so that Oracle OLAP can identify the end of an input host variable. For example, the following WHERE clause is ambiguous, because the first host variable could be either ":MARKET AND PRDCODE" or simply ":MARKET."

```
... SELECT ... WHERE mktcode = :market AND prdcode = :product
```

Use the following construction instead.

```
... SELECT ... WHERE :market = mktcode AND :product = prdcode
```

You can also use parenthesis to clarify the syntax, particularly when using a SQL operator that is unknown in Oracle OLAP.

```
... SELECT ... WHERE (mktcode = :market) AND (prdcode LIKE :product)
```

SQLTABLES

SQLTABLES is a pseudo procedure that returns the following values for each table that matches the search criterion. See Example 22–3, "Discovering Information About Relational Tables" on page 22-15.

tableowner -- A text value identifying the owner of the table.

tablename -- A text value identifying the name of the table.

tabletype -- A text value identifying the type of table using one of the following: TABLE, VIEW, SYSTEM TABLE, ALIAS, SYNONYM, LOCAL TEMPORARY, GLOBAL TEMPORARY, or `NA` (indicating an unrecognized type).

SQLCOLUMNS

SQLCOLUMNS is a pseudo procedure that returns the following values for each column that matches the search criterion. See Example 22–4, "Discovering Information About the Columns of a Relational Table" on page 22-16.

tableowner -- A text value identifying the owner of the table.

tablename -- A text value identifying the name of the table.

colname -- A text value identifying the name of the column.

coltype -- A text value identifying the data type of the column.

olaptype -- A text value identifying the data type that most closely matches *coltype.*

length -- An integer value identifying the length of column values.

precision -- An integer value identifying the precision of numeric column values.

scale -- An integer value identifying the scale of column values.

nullable -- A text value of `Y` or `N` indicating whether the column can contain null values.

Examples

Example 22–2 Testing for the Value of SQLCODE

Cursor `c1` is declared for three columns in the table `mkt`, which is owned by user `sqldba`. Values from the three columns are fetched into three analytic workspace objects. The first host variable is the `market` dimension, which is temporarily limited to the retrieved value. Because of the temporary status of `market`, the other column values are assigned to the appropriate cells of the other host variables.

This example tests the value of SQLCODE in two places. A more complete program would do more error checking.

```
DEFINE market DIMENSION TEXT
DEFINE mkt.desc TEXT <market>
DEFINE mkt.abbrev ID <market>
DEFINE sql.market PROGRAM
PROGRAM
TRAP ON ERROR
SQL DECLARE c1 cursor FOR -
   SELECT mktcode, mktabbrev, mktdesc FROM sqldba.mkt
SQL OPEN c1
IF SQLCODE NE 0
   THEN SIGNAL SQLERR 'open cursor failed.'
WHILE SQLCODE EQ 0
   SQL FETCH c1 INTO :APPEND market, :mkt.abbrev, :mkt.desc
SQL CLOSE c1
   ...
RETURN
error:
   ...
END
```

Example 22–3 Discovering Information About Relational Tables

The following program fetches information about all tables owned by `Scott`. Notice that the value of the *ownername* variable is set after the SQL DECLARE cursor command; it can be set any time before the SQL OPEN command. The *tablename* variable is not set, but is initialized automatically to `NA`, which is passed as a null value.

```
DEFINE ownername TEXT     "Search criteria
DEFINE tablename TEXT     "Search criteria
DEFINE tblowner TEXT      "Search results
DEFINE tblname TEXT       "Search results
DEFINE tbltype TEXT       "Search results

SQL DECLARE c1 CURSOR FOR PROCEDURE sqltables(:ownername, :tablename)
ownername = 'Scott'
SQL OPEN c1
WHILE SQLCODE EQ 0
   DO
   SQL FETCH c1 INTO :tblowner, :tblname, :tbltype
      ...     "Process fetched values
   DOEND
```

Example 22–4 Discovering Information About the Columns of a Relational Table

The following program fetches information about all columns in the `employee` table owned by `Scott`. Notice that `NULL` (and not `NA`) is used for the value of the third argument to `SQLCOLUMNS` since it is processed by the relational manager, not Oracle OLAP.

```
DEFINE tblname TEXT             "Search results
DEFINE tblowner TEXT            "Search results
DEFINE colname TEXT             "Search results
DEFINE coltype TEXT             "Search results
DEFINE olaptype TEXT            "Search results
DEFINE length INTEGER           "Search results
DEFINE precision INTEGER        "Search results
DEFINE scale INTEGER            "Search results
DEFINE nullable TEXT            "Search results

SQL DECLARE c1 CURSOR FOR PROCEDURE sqlcolumns('Scott', -
   'Employee', NULL)
SQL OPEN c1
WHILE SQLCODE EQ 0
   DO
   SQL FETCH c1 INTO :tblowner, :tblname, :colname, :coltype, -
   :olaptype, :length, :precision, :scale, :nullable
      ...  "Process fetched values
    DOEND
```

SQL EXECUTE

The SQL EXECUTE command executes SQL statements that have been compiled using SQL PREPARE. Typically, the SQL statements that you precompile are statements that will be executed repeatedly, particularly those involving input host variables, such as `INSERT`, `UPDATE`, and `DELETE`.

Syntax

SQL EXECUTE *statement-name*

Arguments

statement-name
The name that you assigned to the executable code when you prepared it using SQL PREPARE.

Notes

Restrictions

The SQL PREPARE and SQL EXECUTE commands can only be used within the same DML program.

Examples

Example 22–5 Updating a Relational Table Using Analytic Workspace Data

The next example shows a simple update of a table using data stored in Oracle OLAP. The market dimension is limited to one value at a time in the FOR loop. The SQL phrase `WHERE s.market=:market` specifies that the sales value in the row for that market is the value that is changed.

```
FOR market
   SQL UPDATE mkt SET sales=:mkt.sales WHERE s.market=:market
```

An UPDATE statement should be used in a SQL PREPARE command and executed in a FOR loop.

```
SQL PREPARE s2 FROM UPDATE mkt -
   SET sales=:mkt.sales WHERE s.market=:market
FOR market
   DO
      SQL EXECUTE s2
      IF SQLCODE NE 0
      THEN BREAK
   DOEND
```

SQL FETCH

The SQL FETCH command retrieves and processes data specified by a named SQL cursor. SQL FETCH assigns the retrieved data to OLAP objects. When you include a THEN clause, SQL FETCH may perform processing on the retrieved data.

Syntax

SQL FETCH *cursor* [LOOP [*loopcount*]] -

  INTO :*targets*... [THEN *action-statements*...]

where:

targets is one or more of the following:

[MATCH] *dimension|surrogate*

APPEND [*position*] *dimension*

ASSIGN *surrogate*

variable | *qualified data reference* | *relation* | *composite*

position is one of the following:

AFTER *dimension-value*

BEFORE *dimension-value*

FIRST

<u>LAST</u>

Arguments

cursor
The name of a declared and opened cursor.

LOOP
Specifies that Oracle OLAP should implicitly loop over the rows obtained from a relational table. For each row, Oracle OLAP copies the data in individual fields to objects specified as target analytic workspace objects. When you include a LOOP clause, SQL FETCH will continue processing rows until it reaches the end of the active set specified by the cursor, or an error occurs, or *loopcount* is satisfied. In most cases, you should use the LOOP clause to improve the performance of SQL FETCH.

When you do not specify a LOOP clause and the cursor contains more than one row in its active set, you must code the SQL FETCH command within a WHILE loop. This loop must be based on the value of the SQLCODE option, which returns a nonzero value to indicate the end of the data or an error.

loopcount

Optional integer argument to the LOOP keyword. *Loopcount* controls how SQL FETCH will loop over the rows from a relational table. *Loopcount* can be a literal value, a host variable, or `NA`. When *loopcount* is less than or equal to zero, no looping occurs and no data is fetched.

When you specify a LOOP clause without a value for *loopcount*, SQL FETCH will continue reading rows and copying their contents to target analytic workspace objects until there are no more rows or an error occurs. Internally, each row is processed until SQLCODE is nonzero.

When you specify a literal value for *loopcount*, SQL FETCH will process the number of rows specified by *loopcount* or until SQLCODE is nonzero.

When you specify a variable for *loopcount*, it must be in the form of a host variable (preceded by a colon). This variable acts as both an input and an output host variable. The initial value of *loopcount* specifies the number of rows that SQL FETCH will attempt to process. Upon completion of the SQL FETCH, *loopcount* contains the number of rows actually processed.

When you specify `NA` for loopcount, SQL FETCH will process rows until SQLCODE is nonzero. However, upon completion of the SQL FETCH, *loopcount* will contain the number of rows actually processed.

targets

Identifies the analytic workspace objects in which you want to store data that is retrieved from a relational table. This list of target analytic workspace objects must correspond in number and data type with the list of table columns specified in the *select-statement* argument of the SQL DECLARE CURSOR command that declared *cursor*. A target can be a variable, a qualified data reference, a relation, a dimension, a composite, or a conjoint.

Important: The order in which you specify the target analytic workspace objects effects dimension status. For each dimension value, Oracle OLAP temporarily limits the status of the dimension to the fetched value. Values are assigned to subsequent analytic workspace objects according to this temporary status. See "Conjoints as Target Analytic Workspace Objects" on page 22-22 and "Composites as Target Analytic Workspace Objects" on page 22-23.

A target must be preceded by a colon. When the target is a dimension, it can include the MATCH and APPEND keywords to specify dimension handling; in this case, the colon precedes the keywords.

[MATCH] dimension
[MATCH] surrogate
Oracle OLAP does not perform dimension maintenance on the *target* dimension or surrogate. It uses the incoming values to align data that is being fetched into dimensioned objects. When a value from the relational database does not match any value in the dimension or surrogate, an error is signaled. (Default)

APPEND [*position*] *dimension*
Oracle OLAP performs dimension maintenance on the *target* dimension, adding new values to the dimension. It uses both old and new dimension values to align data being fetched into dimensioned objects. By default, new values are added to the end of a dimension or surrogate. The *position* can also be used to control how dimension values are processed in action statements.

AFTER *dimension-value*
Any new values are added after *dimension-value* in the status list.

BEFORE *dimension-value*
Any new values are added immediately before *dimension-value* in the status list.

FIRST
Any new values are added to the beginning of the status list.

LAST
Any new values are added to the end of the status list.

ASSIGN *surrogate*
Assigns the values to the specified surrogate.

THEN *action-statements*

You may optionally include a THEN clause to specify any number of *action-statements* to be performed each time a row of data is fetched and assigned to target analytic workspace objects. An *action-statement* can be one of the following:

assignment-statement

IF *statement*

SELECT-*statement*

ACROSS *statement*: *action-statement*

<*action-statement-group*>

Refer to the SQL IMPORT command for a complete description of the syntax of *action-statements*.

Notes

Related OLAP DML Commands

You use the SQL FETCH command in combination with other SQL commands to copy data from relational tables into analytic workspace objects as outlined in "Copying Relational Data into Analytic Workspace Objects" on page 22-4.

Effect of Order Targets on Dimension Status

For each dimension value, Oracle OLAP temporarily limits the status of the dimension to the fetched value. Values are assigned to subsequent analytic workspace objects according to this temporary status.

Differences Between SQL FETCH and SQL IMPORT

SQL FETCH and SQL IMPORT both copy data from relational tables into analytic workspace objects. Although SQL FETCH offers the most functionality, SQL IMPORT offers improved performance when copying large amounts of data from relational tables into analytic workspace objects.

Conjoints as Target Analytic Workspace Objects

You can use a conjoint dimension as a target analytic workspace object, but you must ensure that you select the same number of columns from the relational table as there are simple base dimensions. When Oracle OLAP executes a SQL FETCH command for a target that is a conjoint dimension, it uses the dimension order that was specified when the conjoint was defined.

Composites as Target Analytic Workspace Objects

You can specify analytic workspace objects for composites just as you would for dimensioned variables. For example, to fetch data into a variable `var1` dimensioned by `dim1` and `dim2`, you would specify the following list of target analytic workspace objects.

```
:dim1 :dim2 :var1
```

To fetch data into a variable `var2` dimensioned by a composite whose dimensions are `dim1` and `dim2`, you would specify the following list of target analytic workspace objects.

```
:dim1 :dim2 :var2
```

Null Values

A null value in SQL is equivalent to an `NA` value in Oracle OLAP, so null values fetched into target analytic workspace objects are given `NA` values. Since Oracle OLAP handles null values in this way, the SQL command does not support `INDICATOR` variables in the INTO clause of a SQL FETCH command. When fetching null values into a dimension, however, Oracle OLAP discards the values for the entire row.

Converting Oracle RDBMS Data Types into Oracle OLAP Data Types

Table 22–2, " RDBMS Data Type Conversion to OLAP DML Data Types" on page 22-35 shows which Oracle RDBMS data types can be automatically converted into Oracle OLAP data types. You must explicitly convert or cast other data types in the SELECT statement within the SQL DECLARE CURSOR command.

Boolean Data

You can use Boolean variables as input and target analytic workspace objects for OLAP SQL commands. In input host variables, Oracle OLAP treats Boolean values as integers with a value of `1` (`TRUE`) or `0` (`FALSE`).

As target analytic workspace objects, Boolean variables can receive values from any numeric (or bit) column in a relational table.

Date Data

When fetching text data into a DATE variable, the current setting of the DATEORDER option is used to interpret the value. For example, a text value of

`12-08-96` could be interpreted as December 8, 1996, or August 12, 1996, depending on the setting of DATEORDER.

Unusable Data Types

You cannot transfer data with the following data types: RAW, LONG RAW, ROWID, UROWID, BLOB, and BFILE.

Examples

Example 22–6 Fetching Data From Relational Tables -- A Simple SQL FETCH

he following program fragment shows the basic steps of declaring and opening a cursor, and fetching the data. Relational data from the `Prod_ID` and `Prod_Name` columns of the `Products` table are fetched into the `prod` dimension and `prod_label` variable. The variable `prod_label` is dimensioned by `prod`. Notice that the SQL FETCH command in this example does not include a LOOP clause; it therefore retrieves a single row of data each time it is called.

```
VARIABLE set_price SHORT
set_price = 20
     ...
SQL DECLARE highprice CURSOR FOR SELECT Prod_ID, Prod_Name -
   FROM Products WHERE suggested_price > :set_price
SQL OPEN highprice
WHILE SQLCODE EQ 0
     SQL FETCH highprice INTO :prod, :prod_label
```

Example 22–7 Fetching Data From Relational Tables with a THEN Clause

The following program fragment shows the SQL FETCH command from the previous example with the addition of the LOOP keyword and a THEN clause. Because of the LOOP keyword, this SQL FETCH command does not need to run within a WHILE loop. The action statement following the THEN keyword copies any product names stored in `prod_label` that start with the letter `A` into a multiline text variable called `a_product`.

```
SQL FETCH highprice LOOP INTO :prod, :prod_label -
   THEN IF UPCASE(EXTCHARS(prod_label, 1, 1)) EQ 'a' -
      THEN a_product = JOINLINES(a_product prod_label)
```

Example 22–8 Populating with Relational Data While Maintaining a Conjoint Dimension

In this example, a conjoint dimension (named `mpt`) is used as a target analytic workspace object. To populate a conjoint dimension, you must select values from the relational database for each of its base dimensions. Here, the three base dimensions are `market`, `product`, and `time`. Therefore, the `SELECT` statement specifies the three corresponding columns (`Mktcode`, `Prdcode`, and `Percode`). The program assumes that the `market`, `product`, and `time` dimensions are already populated with up-to-date values; Oracle OLAP does not update the base dimensions unless you explicitly specify them as target analytic workspace objects.

```
DEFINE mpt DIMENSION <market product time>
DEFINE sql.mpt PROGRAM
PROGRAM
   ...
SQL DECLARE c1 CURSOR FOR -
   SELECT Mktcode, Prdcode, Percode FROM Sqldba.Data
IF SQLCODE NE 0
   THEN SIGNAL sqlerrm
SQL OPEN c1
SQL FETCH c1 LOOP INTO :append mpt
SQL CLOSE c1
   ...
END
```

Example 22–9 Populating Data While Maintaining Base and Conjoint Dimensions

To retrieve current values for the base and conjoint dimensions, or to retrieve the values for the first time, you can fetch the values for the base dimensions immediately before you fetch the values for the conjoint dimension. In the following example, the SQL DECLARE CURSOR and SQL FETCH commands have been edited to fetch both base and conjoint dimension values. Notice that the number of columns selected from the relational table must match the number of *base* dimensions fetched. There are six column specifications in the SELECT statement. The first three match the three base dimensions, and the last three match the conjoint dimension itself.

```
SQL DECLARE c1 CURSOR FOR -
   SELECT Mktcode, Prdcode, Percode, Mktcode,  -
      Prdcode, Percode FROM Sqldba.Data
   ...
  SQL FETCH c1 LOOP INTO :APPEND market, :APPEND product, -
   :APPEND time, :APPEND mpt
```

Example 22–10 Populating Variables with Relational Table Data while Maintaining Dimensions

In the next example, variable `dollars.mpt` is dimensioned by the conjoint `mpt`, and its values are populated in the same SQL FETCH command with the dimension values. The SQL DECLARE CURSOR and SQL FETCH commands have been edited again with the new column and target analytic workspace object added.

```
DEFINE dollars.mpt DECIMAL <mpt>
SQL DECLARE c1 CURSOR FOR -
   SELECT Mktcode, Prdcode, Percode, Mktcode, Prdcode, -
      Percode, Dollars FROM Sqldba.Data
   ...
SQL FETCH c1 LOOP INTO :APPEND market, :APPEND product, -
   :APPEND time, :APPEND mpt, :DOLLARS.mpt
```

Example 22–11 Fetching Data into a Concat Dimension

Assume that a relational table has four columns of product data and that you decide to create a Product hierarchy with four levels in your analytic workspace to hold this data. The levels in the hierarchy (`prod_id`, `prod_subcategory`, `prod_category`, and `products_all`) map to columns in the `products` tables. The lowest level of the hierarchy is `prod_id` and the highest level is `products_all`. There is also a column with supplier information in the table.

To hold the data in the analytic workspace you define a dimension was defined for each level of the Product hierarchy, a concat dimension for the hierarchy itself, and a child-parent relation between the values in the hierarchy. You also define a dimension for the supplier data and a relation that holds the relationship between suppliers and products with the following definitions.

```
DEFINE aw_prod_id DIMENSION NUMBER (6)
DEFINE aw_prod_subcategory DIMENSION TEXT
DEFINE aw_prod_category DIMENSION TEXT
DEFINE aw_products_all DIMENSION TEXT
DEFINE aw_products DIMENSION CONCAT (aw_products_all -
                                  aw_prod_category -
                                  aw_prod_subcategory -
                                  aw_prod_id)
DEFINE aw_products.parents RELATION aw_products <aw_products>
DEFINE aw_supplier_id DIMENSION NUMBER (6)
DEFINE aw_prod_id.aw_supplier_id RELATION aw_supplier_id <aw_prod_id>
```

Assume that you write a program named `get_products_hier` that consists of the following code.

```
' get_products_hier Program
ALLSTAT
" Fetch values into the products hierarchy
SQL DECLARE grabprods CURSOR FOR SELECT prod_total, -
                                        prod_category, -
                                        prod_subcategory, -
                                        prod_id -
                                   FROM sh.products
SQL OPEN grabprods
SQL IMPORT grabprods INTO :APPEND aw_products_all -
                              :APPEND aw_prod_category -
                              :APPEND aw_prod_subcategory -
                              :APPEND aw_prod_id

SQL CLOSE grabprods
SQL CLEANUP
" Update the analytic workspace and make the updates permanent
UPDATE
COMMIT
" Fetch values into supplier_id
SQL DECLARE grabsupid CURSOR FOR SELECT supplier_id -
                                 FROM sh.products
SQL OPEN grabsupid
SQL IMPORT grabsupid INTO :APPEND aw_supplier_id
SQL CLOSE grabsupid
SQL CLEANUP
" Update the analytic workspace and make the updates permanent
UPDATE
COMMIT

" Populate self-relation for concat dimension
" and relation between aw_prod_id and aw_supplier_id
SQL DECLARE makerels CURSOR FOR SELECT prod_total, -
                                       prod_category, -
                                       prod_subcategory, -
                                       prod_id, -
                                       supplier_id -
                                 FROM sh.products
SQL OPEN makerels
SQL FETCH makerels LOOP INTO :MATCH aw_products_all -
                             :MATCH aw_prod_category -
                             :MATCH aw_prod_subcategory -
                             :MATCH aw_prod_id -
                             :MATCH aw_supplier_id -
            THEN aw_products.parents(aw_products aw_prod_id) -
                = aw_products(aw_prod_subcategory aw_prod_subcategory) -
            aw_products.parents(aw_products aw_prod_subcategory) -
               = aw_products(aw_prod_category aw_prod_category) -
            aw_products.parents(aw_products aw_prod_category) -
               = aw_products(aw_products_all aw_products_all) -
            aw_prod_id.aw_supplier_id = aw_supplier_id
SQL CLOSE makerels
SQL CLEANUP
" Update the analytic workspace and make the updates permanent
UPDATE
COMMIT
```

The `get_products_hier` program copies the data from the dimension tables into the base dimensions of the `aw_products` concat dimension using `SQL FETCH` commands with the `APPEND` keyword. As the base dimensions of `aw_products` are populated, Oracle OLAP automatically populates `aw_products`, itself. As the `THEN` clause of the `SQL FETCH` command executes, Oracle OLAP fetches data into the child-parent self-relation for `aw_products`. This program also populates the `aw_supplier_id` dimension and its relation.

SQL IMPORT

The SQL IMPORT command retrieves and processes data specified by an explicit SQL cursor. SQL IMPORT assigns the retrieved data to OLAP objects. When you include a THEN clause, SQL IMPORT may perform processing on the retrieved data SQL IMPORT is particularly effective in copying fact data from relational tables into analytic workspace variables.

Syntax

SQL IMPORT *cursor* INTO :*targets*... [THEN *action-statements*...]

where:

targets is one or more of the following:

[<u>MATCH</u>|MATCHSKIPERR [[*position*]] {*dimension*|*surrogate*|*valueset*|*relation*}

APPEND *dimension*

ASSIGN *surrogate*

{*variable*|*relation*|*qualified data reference*}

action-statements is one of the following:

assignment-statement

IF-statement

SELECT-statement

ACROSS-statement: *action-statement*

<*action-statement-group*>

Arguments

cursor
The name of a declared cursor.

targets
Identifies the analytic workspace objects in which you want to store data that is retrieved from a relational table. This list of target analytic workspace objects must correspond in number and data type with the list of table columns specified in the

select-statement argument of the SQL DECLARE CURSOR command that declared *cursor*. A target can be a variable, a qualified data reference, a relation, a dimension, or a composite.

Important: The order in which you specify the analytic workspace objects effects dimension status. For each dimension value, Oracle OLAP temporarily limits the status of the dimension to the fetched value. Values are assigned to subsequent analytic workspace objects according to this temporary status.

MATCH
Oracle OLAP does not copy values from the corresponding relational table column into the *target* dimension or surrogate. It merely uses the values to align data that is being fetched into dimensioned objects. When a value from the relational database does not match any value in the dimension, an error is signaled. (Default)

MATCHSKIPERR
Oracle OLAP does not copy values from the corresponding relational table column into the *target* dimension or surrogate. It merely uses the values to align data that is being fetched into dimensioned objects. When a value from the relational database does not match any value in the dimension, the value is ignored and processing continues without signaling an error.

position
The one-based logical position of the value.

APPEND
Oracle OLAP performs dimension maintenance on the *target* dimension, adding new values from the corresponding relational table column to the dimension. It uses both old and new dimension values to align data being fetched into dimensioned objects. New values are added to the end of a dimension.

ASSIGN
Oracle OLAP assigns the corresponding relational value to the specified surrogate.

dimension
The name of the analytic workspace dimension.

surrogate
The name of an analytic workspace surrogate.

valueset
The name of the analytic valueset.

relation
The name of the analytic workspace relation.

variable
The name of a variable.

qualified_data_reference
A QDR is a qualifier that limits one or more of the dimensions of a variable or a relation to a single value. Oracle OLAP evaluates QDRs in a SQL IMPORT command, as follows:

- When the QDR includes an expression, the expression is evaluated only once before the data is retrieved. In other words, the expression is, in essence, a constant.
- When the QDR includes a relation, the values of the QDR vary depending on the status of the dimensions of that relation.

THEN *action-statements*
You may optionally include a THEN clause to specify any number of *action-statements* to be performed each time a row of data is imported and assigned to analytic workspace objects. Action statements may contain simple assignment statements, conditional assignment statements, and assignments across dimensions.

Action statements allow you to examine and manipulate the fetched data on a row-by-row basis. For example, you may want to specify temporary objects as analytic workspace objects and only update your permanent objects once you have performed certain actions on the row of fetched data. However, action statements do not have to reference the imported data. For example, one of your action statements might be an assignment statement that executes a user-defined function (that is, a program) that performs complex processing and then simply increments a counter.

A THEN clause can improve SQL loading performance by eliminating the need for postprocessing upon completion of a SQL IMPORT.

Note: The syntax of an action statement within SQL IMPORT is essentially the same as the syntax of an action statement within FILEREAD. Exceptions are in the syntax of an assignment statement and the use of the VALUE keyword. In SQL IMPORT action statements, assignments must be explicit; they must include a source, target, and equal sign. In FILEREAD action statements, assignments may be implicit and specify only the target. The VALUE keyword is supported in FILEREAD action statements, but not in SQL IMPORT action statements. When you have already specified action statements for use with FILEREAD, you can reuse the code with SQL IMPORT by simply adjusting the assignment statements and eliminating the VALUE keyword (if necessary). Most of the attributes listed in FILEREAD (with the exception of the attributes that control dimension processing) are not meaningful for SQL loading and are ignored when executing within SQL IMPORT.

For best performance, within a THEN clause reference only the data within the imported row.

In your list of action statements, *be sure to process dimensions before variables*. Oracle OLAP processes each action statement from left to right for each row in the relational table. When an action statement performs dimension processing, the resulting status remains in effect for subsequent action statements. When you do not first specify action statements that limit a variable's dimensions, Oracle OLAP uses the first value in status to target a cell in the variable. Unless you specify an ACROSS phrase, Oracle OLAP assigns a single value from a row to a single cell in an Oracle OLAP variable. By default, Oracle OLAP does *not* loop over a variable's dimensions when assigning data to the variable.

assignment-statement
An assignment statement (SET) that assigns a value that is the result of an expression to an Oracle OLAP object.

IF-*statement*
An IF...THEN...ELSE statement that performs some action depending on whether a Boolean expression is `TRUE` or `FALSE`.

SELECT *statement*
A SQL SELECT statement lets you perform some action based on the value of an expression. A SELECT statement has the following form.

SELECT *select-expression*

[WHEN *expression1 action*

[WHEN *expression2 action* . . .]

[ELSE *action*]

SELECT evaluates the SELECT expression and then sequentially compares the result with the WHEN expressions. When the first match is found, the associated *action* occurs. When no match is found, the ELSE *action* (if specified) occurs.

ACROSS-statement: *action-statement*

An ACROSS statement causes the following action statement to execute once for every value in status of the ACROSS dimension. When you want the looping to apply to more than one action statement, enclose the action statements in angle brackets. An ACROSS statement has the following form.

ACROSS *dimension* [*limit*]:

action-statement

limit temporarily change the status of *dimension*, as long as you are not in a FOR loop over *dimension*. The new status is in effect only for the duration of the SQL FETCH command. The format of *limit* is as follows.

[ADD|COMPLEMENT|KEEP|REMOVE|TO] *limit-clause*

To specify the temporary status, insert any of the LIMIT command keywords (the default is TO) along with an appropriate list of dimension values or related dimensions. You can use any valid LIMIT clause (see LIMIT command for further information). The following example limits `month` to the last six values, no matter what the current status of `month` is.

```
ACROSS month last 6: units
```

<action-statement-group>

You can group several action statements together by enclosing them in angle brackets. An *action-statement-group* has the following form.

<*action-statement1* -

[*action-statement2* . . .]>

A typical use for action statement groups is after an ACROSS statement. With the angle bracket syntax, you can cause more than one action statement to execute for every value in status of the ACROSS dimension.

Notes

Related OLAP DML Commands

You use the SQL IMPORT command in combination with other SQL commands to copy data from relational tables into analytic workspace objects as outlined in "Copying Relational Data into Analytic Workspace Objects" on page 22-4.

Effect of Order Targets on Dimension Status

For each dimension value, Oracle OLAP temporarily limits the status of the dimension to the fetched value. Values are assigned to subsequent analytic workspace objects according to this temporary status.

Differences Between SQL FETCH and SQL IMPORT

SQL FETCH and SQL IMPORT both copy data from relational tables into analytic workspace objects. SQL IMPORT offers improved performance when copying large amounts of data from relational tables into analytic workspace objects.

Restrictions When Declaring a Cursor for Use by SQL IMPORT

For the syntax to use when declaring a cursor for use by SQL IMPORT see the notes for SQL DECLARE CURSOR.

Converting Oracle RDBMS Data Types into Oracle OLAP DML Data Types

Table 22–2, " RDBMS Data Type Conversion to OLAP DML Data Types" shows which Oracle RDBMS data types the SQL IMPORT command automatically converts into Oracle OLAP data types. You must explicitly convert or cast other data types in the SELECT statement within the SQL DECLARE CURSOR command.

Table 22–2 RDBMS Data Type Conversion to OLAP DML Data Types

Oracle RDBMS Data Type	OLAP DML Dimension Type	OLAP DML Variable Data Type
CHAR, NCHAR, NVARCHAR2, VARCHAR2	TEXT [WIDTH n], ID, NTEXT	TEXT, NTEXT
NUMBER	NUMBER, INTEGER, SHORTINTEGER, LONGINTEGER	NUMBER, INTEGER, BOOLEAN, SHORTINTEGER, LONGINTEGER, DECIMAL, SHORTDECIMAL
CLOB (only within SQL FETCH and SQL SELECT statements)	TEXT	TEXT
NCLOB (only within SQL FETCH and SQL SELECT statements)	NTEXT	NTEXT
DATE	-	DATE, DATETIME

Boolean data

You can use Boolean variables as input and target analytic workspace objects for OLAP SQL commands. In input host variables, Oracle OLAP treats Boolean values as integers with a value of 1 (TRUE) or 0 (FALSE).

As target analytic workspace objects, Boolean variables can receive values from any numeric (or bit) column in a relational table.

Date data

When importing text data into a DATE variable, the current setting of the DATEORDER option is used to interpret the value. For example, a text value of 12-08-96 could be interpreted as December 8, 1996, or August 12, 1996, depending on the setting of DATEORDER.

Unusable data types

You cannot transfer data with the following data types: RAW, LONG RAW, ROWID, UROWID, BLOB, and BFILE.

Examples

Example 22–12 Simple Import

The following program fragment shows the basic steps of declaring a cursor and importing the data. Values from the `Prod_ID` and `Prod_Name` columns of the `Products` relational table in the Sales -History (`sh`) database are fetched into the `prod_id` dimension and `prod_label` analytic workspace variable. The `prod_label` variable is dimensioned by `prod_id`.

```
SQL DECLARE productcur CURSOR FOR SELECT Prod_ID, Prod_Name FROM sh.Products
SQL OPEN productdur
SQL IMPORT productcur INTO :prod_id, :prod_label
SQL CLOSE productcur
SQL CLEANUP
```

SQL OPEN

The SQL OPEN command activates an explicitly-declared SQL cursor. When the cursor is opened, SQL examines any input host variables used in the definition of the specified cursor, determines the cursor's result set, and leaves the cursor in the open state for use by SQL FETCH or SQL IMPORT. The cursor is positioned before the first row of the result set.

Syntax

SQL OPEN *cursor*

Arguments

cursor
The name of a cursor previously declared in the same program. You cannot use ampersand substitution.

Notes

Related OLAP DML Commands

You use the SQL OPEN command in combination with other SQL commands to copy data from relational tables into analytic workspace objects as outlined in "Copying Relational Data into Analytic Workspace Objects" on page 22-4.

Examples

Opening a Cursor

The following program fragment declares and opens a cursor named `geolabels`.

```
SQL DECLARE geolabels CURSOR FOR -
   SELECT Store_ID, Store_Name, City FROM Stores
IF SQLCODE NE 0
   THEN SIGNAL dclerror 'SQLERRM'
SQL OPEN geolabels
IF SQLCODE NE 0
   THEN SIGNAL operror 'SQLERRM'
```

SQL PREPARE

The SQL PREPARE command precompiles a SQL statement for later execution using SQL EXECUTE. Typically, you use SQL PREPARE in programs to optimize the processing of SQL statements that will be executed repeatedly, particularly those involving input host variables, such as INSERT, UPDATE, and DELETE.

Syntax

SQL PREPARE *statement-name* FROM *sql-statement* [*insert-options*]

Arguments

statement-name
A name that you assign to the executable code produced from *sql-statement*. You can redefine *statement-name* just by issuing another SQL PREPARE command.

sql-statement
The SQL statement that you want to precompile for more efficient execution. It cannot contain ampersand substitution or variables that are undefined when the program is compiled.

insert-options
The following options are optional when *sql-statement* is an `INSERT` statement:

DIRECT=YES|NO specifies if the insert is a direct-path `INSERT`. This option must be the first option specified right aver the *values* phrase of the `INSERT` statement. Setting this option to `YES` specifies that the insert will be a direct-path `INSERT`. Direct-path `INSERT` enhances performance during `INSERT` operations and is similar to the functionality of Oracle's direct-path loader utility, SQL*Loader. The default value is `NO` which specifies a normal `INSERT`.

NOLOG=YES|NO specifies if logging occurs. Setting this option to `YES` specifies that the redo information is not recorded in the redo log files which makes load-time faster. The default value is `NO` which specifies logging mode.

PARTITION=*(sub)partition-name* specifies that only the segments related to the named partition or subpartition are locked. When you specify this option, another session can insert data to unrelated segments in the same table. When you do not specify this option (the default), other sessions cannot insert data into the same table.

Notes

Restrictions

The SQL PREPARE and SQL EXECUTE commands can only be used within the same OLAP DML program.

Improved Performance Using Direct-Path INSERT

When performing a direct-path `INSERT`, data is written directly into data files, bypassing the buffer cache, free space in the existing data is not reused, and the inserted data is appended after existing data in the table

Restrictions When Using Direct-Path INSERT

Direct-path `INSERT` is subject to a number of restrictions. When executing a direct-path `INSERT` using the OLAP DML, transactions in the session issuing the direct-path `INSERT` must be committed for the `INSERT` to execute successfully. (You can use the SQL or OLAP DML COMMIT to commit transactions.)

Additionally, the general restrictions that apply to using direct-path `INSERT` in SQL apply to preparing a direct-path`INSERT` using the OLAP DML PREPARE command:

- The target table cannot be index organized or clustered.
- The target table cannot contain object type or LOB columns.
- The target table cannot have any triggers or referential integrity constraints defined on it.

For more information on restrictions when using a direct-path `INSERT`, see the discussion of the `INSERT` statement in *Oracle Database SQL Reference*.

Data Type Conversions

Table 22–3, "Automatic Data Type Conversion During Direct-Path Insertion" on page 22-40 shows the automatic data type conversion performed during direct-path insertion.

Table 22–3 Automatic Data Type Conversion During Direct-Path Insertion

Oracle RDBMS	Oracle OLAP DML
CHAR(*n*), VARCHAR(*n*)	TEXT
LONG	TEXT with WIDE option
CHAR(8), VARCHAR(8)	ID
DATE	DATE
NUMBER(*x*,*x*)	DECIMAL (SHORTDECIMAL)
INTEGER (or NUMBER(38)	INTEGER (SHORTINTEGER)
NUMBER(1)	BOOLEAN

Date Data

When inserting text data from Oracle OLAP into a column with a DATE data type, you must use the default date format of `DD MMM YY`. You can use slashes (`/`) or hyphens (`-`) as separators, as well as spaces. When the data is in a different format, you can use the Oracle `TO_DATE` function in the SQL `INSERT` command.

Inserting Large Text Values into a Table

To insert more than 2K bytes of text data from Oracle OLAP into a CLOB or NCLOB column, use the WIDE keyword before the name of the input host variable. When the input host variable is `TEXT`, then the target data type is `CLOB`. When the input host variable is `NTEXT`, then the target data type is `NCLOB`.

The following is the syntax of an input host variable with the WIDE keyword. See Example 22–15, "Using the WIDE Keyword" on page 22-42 for an example.

```
:WIDE input-host-variable
```

See Example 22–15, "Using the WIDE Keyword" on page 22-42 for an example.

Note that the target table must conform to these guidelines:

- Any number and combination of CLOB and NCLOB columns
- No LONG columns

The RDBMS imposes some restrictions on large data types. Oracle OLAP will not signal an error when you violate these restrictions. However, you might get unexpected results. Refer to the *Oracle Application Developer's Guide* for restrictions on large data types.

Calculating the Number of Characters

You can calculate the number of characters that will be sent to the database from an input host variable by using the following formula.

```
NUMCHARS(variable) + 2 * (NUMLINES(variable) - 1)
```

For example, the following statement shows the number of characters that will be sent using `bigvar` as the input host variable.

```
SHOW NUMCHARS(bigvar) + 2 * (NUMLINES(bigvar) -1)
```

This formula counts the extra carriage return and line feed characters that Oracle OLAP inserts between lines when passing the text to the database.

Examples

Example 22–13 Preparing a FOR Loop

To automatically add all the sales people from the `salesperson` dimension to the relational table, you could write a program and put the SQL `INSERT` command in a FOR loop.

```
FOR salesperson
   SQL INSERT INTO Sales VALUES (:Salesperson, :Dollars)
```

When a statement includes input host variables and will be executed repeatedly, such as in a FOR loop, you can make the statements more efficient by "preparing" the SQL statement first. The `INSERT` statement becomes part of a PREPARE statement.

```
SQL PREPARE s1 FROM INSERT INTO Sales VALUES -
   (:Salesperson, :Dollars)
FOR Salesperson
   DO
      SQL EXECUTE s1
      IF SQLCODE NE 0
      THEN BREAK
   DOEND
```

Example 22–14 Updating a Table

The next example shows a simple update of a table using data stored in an analytic workspace. The `market` dimension is limited to one value at a time in the FOR loop. The SQL phrase `WHERE S.Market=:market` specifies that the sales value in the row for that market is the value that is changed.

```
FOR market
   SQL UPDATE Mkt SET Sales=:Mkt.Sales WHERE S.Market=:market
```

Like the `INSERT` statement in the previous example, an `UPDATE` statement should be used in a PREPARE statement and executed in an ACROSS command or FOR loop.

```
SQL PREPARE s2 FROM UPDATE mkt -
   SET Sales=:mkt.sales WHERE s.market=:market
ACROSS market DO 'SQL EXECUTE s1'
```

Example 22–15 Using the WIDE Keyword

In both of the following statements, `WIDE` indicates that the target value is `CLOB` when `var1` is `TEXT`, or `NCLOB` when `var1` is `NTEXT`.

```
SQL INSERT INTO CLOB_TEST values (:dim1 :WIDE var1)
SQL UPDATE CLOB_TEXT SET clob_col = :WIDE var1 WHERE key = 1
```

SQL PROCEDURE

The SQL PROCEDURE command executes procedures stored in the RDBMS.

Note: You can also create SQL stored procedures using the OLAP DML. See:

- "Creating SQL Procedures using the OLAP DML" on page 22-43
- Example 22–16, "Creating a Stored Procedure" on page 22-44

Syntax

SQL PROCEDURE *procedure-name* (*parameters*)

where *parameters* is one or more of the following, separated by commas:

sql-parameter
:*dml-parameter*

Arguments

procedure-name
The name of the SQL stored procedure.

sql-parameter
The name of a variable in the RDBMS.

:*dml-parameter*
A host variable name preceded by a colon. Host variables are OLAP DML expressions such as OLAP DML variables. See "SQL Terminology" on page 22-5 and "Input Host Variables" on page 22-6 for more information on host variables.

Notes

Creating SQL Procedures using the OLAP DML

To create a stored procedure using the OLAP DML, issue an OLAP DML a SQL statement with a SQL `CREATE PROCEDURE` statement as its argument. The syntax for coding `CREATE PROCEDURE` as an argument within an OLAP DML a SQL statement is slightly different than the syntax for coding `CREATE PROCEDURE` in

SQL proper. When coded as an arguments to an OLAP DML statements, use a tilde (~) instead of a semicolon as a terminator, and two colons instead of one in an assignment statement. See Example 22–16, "Creating a Stored Procedure" on page 22-44.

Restrictions When Calling SQL Procedures using the OLAP DML

A stored procedure called using an OLAP DML SQL PROCEDURE statement cannot contain output variables or transactions.

Examples

Example 22–16 Creating a Stored Procedure

The following example shows the syntax for creating a procedure named new_products.

```
SQL CREATE OR REPLACE PROCEDURE new_products -
   (id CHAR, name CHAR, cost NUMBER) AS -
      price NUMBER~ -
   BEGIN -
      price ::= cost * 2.5~ -
      INSERT INTO products -
         VALUES(id, name, price)~ -
   END~
```

Example 22–17 Executing a Stored Procedure

The following FOR loop executes a SQL stored procedure named new_products and inserts data stored in dimensions and variables into a relational table. In this example, prod is an Oracle OLAP dimension, and labels.p and cost.p are variables dimensioned by prod.

```
FOR prod
   DO
      SQL PROCEDURE new_products(:prod, :labels.p, :cost.p)
      IF SQLCODE NE 0
         THEN BREAK
   DOEND
```

SQL SELECT

The SQL SELECT command uses an implicit cursor to copy data from relational tables into analytic workspace objects.

Syntax

SQL SELECT *expressions* FROM *tables* -

[WHERE *predicates*] [GROUP BY *expressions*] -

[ORDER BY *expressions*] [HAVING *predicates*] -

INTO :*targets*... [THEN *action-statements*...]

where:

targets is one or more of the following:

[MATCH] *dimension*|*surrogate*

APPEND [*position*] *dimension*

ASSIGN *surrogate*

variable|*qualified data reference*|*relation*|*composite*

position is one of the following:

Arguments

SELECT *expressions* FROM *tables*-
[WHERE *predicates*] [GROUP BY *expressions*] -
[ORDER BY *expressions*] [HAVING *predicates*]
A SQL SELECT statement that identifies the data you want to associate with the cursor. For the syntax of an SQL SELECT statement, refer to the *Oracle Database SQL Reference.*

targets
Identifies the analytic workspace objects in which you want to store data that is retrieved from a relational table. This list of target analytic workspace objects must correspond in number and data type with the list of table columns specified in the SELECT statement. A target can be a variable, a qualified data reference, a relation, a dimension, or a composite.

Important: The order in which you specify the analytic workspace objects effects dimension status. For each dimension value, Oracle OLAP temporarily limits the status of the dimension to the fetched value. Values are assigned to subsequent analytic workspace objects according to this temporary status. See "Conjoints as Target Analytic Workspace Objects" on page 22-22 and "Composites as Target Analytic Workspace Objects" on page 22-23.

A target must be preceded by a colon. When the target is a dimension, it can include the MATCH and APPEND keywords to specify dimension handling; in this case, the colon precedes the keywords.

[MATCH] dimension
[MATCH] surrogate
Oracle OLAP does not perform dimension maintenance on the *target* dimension or surrogate. It uses the incoming values to align data that is being fetched into dimensioned objects. When a value from the relational database does not match any value in the dimension or surrogate, an error is signaled. (Default)

APPEND [*position*] *dimension*
Oracle OLAP performs dimension maintenance on the *target* dimension, adding new values to the dimension. It uses both old and new dimension values to align data being fetched into dimensioned objects. By default, new values are added to the end of a dimension or surrogate. The *position* can also be used to control how dimension values are processed in action statements.

ASSIGN *surrogate*
Assigns the values to the specified surrogate.

THEN *action-statements*
You may optionally include a THEN clause to specify any number of *action-statements* to be performed each time a row of data is fetched and assigned to analytic workspace objects. An *action-statement* can be one of the following:

- *assignment-statement*
- IF statement
- SELECT-statement
- ACROSS statement: *action-statement*
- *<action-statement-group>*

Refer to the SQL IMPORT command for a complete description of the syntax of *action-statement*.

Notes

Related OLAP DML Commands

You use the SQL SELECT command to copy data from relational tables into analytic workspace objects using an implicit cursor. You can also use copy the data using an explicit cursor using the OLAP DML commands outlined in "Copying Relational Data into Analytic Workspace Objects" on page 22-4.

General Restrictions

The following restrictions apply to the SQL SELECT command cannot contain ampersand substitution.

Optimizing Copies

When copying values from relational tables into a multidimensional input variable, list the columns that correspond to the dimensions in an ORDER BY clause in the *select-statement* argument of the SQL SELECT command, with the slowest-varying dimension first. This will optimize performance.

Ambiguous WHERE Clauses

The *select-statement* argument of a SQL SELECT command can include a WHERE clause. Since both OLAP DML syntax and SQL syntax allow you to use AND and OR, you should construct the clause clearly so that Oracle OLAP can identify the end of an input host variable. For example, the following WHERE clause is ambiguous, because the first host variable could be either ":MARKET AND PRDCODE" or simply ":MARKET."

```
... SELECT ... WHERE mktcode = :market AND prdcode = :product
```

Use the following construction instead.

```
... SELECT ... WHERE :market = mktcode AND :product = prdcode
```

You can also use parenthesis to clarify the syntax, particularly when using a SQL operator that is unknown in Oracle OLAP.

```
... SELECT ... WHERE (mktcode = :market) AND (prdcode LIKE :product)
```

Converting Oracle RDBMS Data Types into Oracle OLAP Data Types

Table 22–2, " RDBMS Data Type Conversion to OLAP DML Data Types" on page 22-35 shows which Oracle RDBMS data types can be automatically converted into Oracle OLAP data types. You must explicitly convert or cast other data types in the SELECT statement.

Examples

Example 22–18 Simple select

For example, assume that there is a relational table named `sales` with the following description.

```
PROD_ID                     NOT NULL NUMBER(6)
CUST_ID                     NOT NULL NUMBER
TIME_ID                     NOT NULL DATE
CHANNEL_ID                  NOT NULL CHAR(1)
PROMO_ID                    NOT NULL NUMBER(6)
QUANTITY_SOLD               NOT NULL NUMBER(3)
AMOUNT_SOLD                 NOT NULL NUMBER(10,2)
```

Assume also that your analytic workspace contains the following definitions for corresponding analytic workspace objects.

```
DEFINE aw_prod_id DIMENSION NUMBER (6)
DEFINE aw_cust_id DIMENSION NUMBER (6)
DEFINE aw_date DIMENSION TEXT
DEFINE aw_channel_id DIMENSION TEXT
DEFINE aw_promo_id DIMENSION NUMBER (6)
DEFINE aw_sales_dims COMPOSITE <aw_prod_id aw_cust_id -
     aw_channel_id aw_promo_id>
DEFINE aw_sales_quantity_sold VARIABLE NUMBER (3) <aw_date aw_sales_dims -
     <aw_prod_id aw_cust_id aw_date aw_channel_id aw_promo_id>>
DEFINE aw_sales_amount_sold VARIABLE NUMBER (10,2) <aw_date aw_sales_dims -
     <aw_prod_id aw_cust_id aw_date aw_channel_id aw_promo_id>>
```

To copy the data for product 415 from the `sales` table into the analytic workspace objects, you execute the following statement in the OLAP worksheet.

```
SQL SELECT prod_id cust_id time_id channel_id promo_id quantity_sold -
amount_sold WHERE prod_id = 415 -
INTO :aw_prod_id, :aw_cust_id, :aw_date,  -
:aw_channel_id, :aw_promo_id, :aw_sales_quantity_sold, :aw_sales_amount_sold
```

SQLBLOCKMAX

The SQLBLOCKMAX option controls the maximum number of records retrieved from an Oracle relational database at one time. This option provides a means of fine-tuning the performance of data fetches.

Data type

INTEGER

Syntax

SQLBLOCKMAX = *records*

Arguments

records
An integer that identifies the number of records you want fetched at one time. While you can set SQLBLOCKMAX to any integer, no appreciable change in performance results in setting it over 100. The default is `10` records.

Notes

Opening Cursors

Only cursors opened after SQLBLOCKMAX is reset will use the new block size.

Number of Records

When a program typically opens a cursor, reads one record, and closes the cursor, you should set SQLBLOCKMAX to `1`. Otherwise, the SQL FETCH command retrieves 10 records and discards 9 of them. The same is true for other routine fetches of less than 10 records.

Block Size

When your program is fetching small records, you can increase SQLBLOCKMAX to reduce the number of blocks required for the fetch. Oracle OLAP fetches the data into a 64K buffer. The block size in bytes is the number of records multiplied by the size of the records. When the block size exceeds the 64K limit imposed by the buffer, Oracle OLAP automatically reduces the number of records fetched. See Example 22–19, "Defining a Cursor with SQLBLOCKMAX" on page 22-50.

Examples

Example 22–19 Defining a Cursor with SQLBLOCKMAX

The following program fragment defines a cursor for fetching 50-byte records from a relational database. The new block size easily fits into Oracle OLAP's 64K buffer (`50` bytes `* 100 = 50k` block size).

```
SQLBLOCKMAX = 100
SQL DECLARE CURSOR c1 FOR SELECT * FROM mydata
SQL OPEN c1
```

SQLCODE

(Read-only) The SQLCODE option holds the value returned by the Oracle RDBMS after the most recently attempted SQL operation.

Return Value

INTEGER. `0` after a successful operation, `-1` after an error, or `100` after all requested rows have been fetched.

Syntax

```
SQLCODE
```

Notes

Signalling Errors

Oracle OLAP does not signal an error when SQLCODE becomes nonzero. Therefore, your program must test the value of SQLCODE and take the appropriate action. Since each SQL operation sets SQLCODE, you must test for errors after each operation to avoid missing an error condition.

Specific Error Codes

You can write programs that look for a specific error code. For example, the most common warning code is `100`, which indicates that the cursor reached the end of its table selection and the FETCH statement is complete.

Error Messages

After an error, the SQLERRM option typically contains an error message.

Examples

Example 22–20 Using SQLCODE When Fetching Data

The following program fragment includes a WHILE loop that tests for the value of SQLCODE and stops trying to fetch data when the end of the cursor's active set is reached.

```
WHILE SQLCODE EQ 0
   SQL FETCH cursor1 INTO :employee, :title
```

SQLERRM

(Read-only) After the database reports an error and SQLCODE has a nonzero value, the SQLERRM option usually contains text that explains the problem.

Data type

TEXT

Syntax

SQLERRM

Notes

Oracle Relational Manager

You can set the SQLMESSAGES option to `YES` to send the value of SQLERRM to the current output file automatically.

Examples

Example 22–21 Displaying Error Messages

The following statements attempt to create a table and check for error messages afterward.

```
SQL CREATE TABLE Products -
   (Prod_ID CHAR(8) -
   Prod_Name VARCHAR(30) -
   Suggested_Price DECIMAL(10,2))
IF SQLCODE NE 0
   SHOW SQLERRM
```

Example 22–22 Sample Error Message

The following statement is incomplete and does not provide sufficient information to create a table.

```
SQL CREATE TABLE Products
```

The Oracle RDBMS returns an error message such as the following.

```
ORA-00906: Missing left parenthesis.
```

SQLMESSAGES

The SQLMESSAGES option controls whether error messages are sent to the current output file.

Data type

BOOLEAN

Syntax

```
SQLMESSAGES = {YES|NO}
```

Arguments

YES
Error messages are sent to the current output file.

NO
Error messages are only stored as values of SQLERRM. (Default)

Notes

Typical Usage
You will want to set SQLMESSAGES to `YES` while you are developing an application so that you can diagnose errors quickly. When your application is in use, you will probably want it to capture and handle errors in a different manner with SQLMESSAGES set to `NO`.

SQRT

The SQRT command computes the square root of an expression.

Return Value

DECIMAL

Syntax

SQRT(*expression*)

Arguments

expression
The numeric expression whose square root is to be computed.

Notes

Negative Expressions

When *expression* is negative and ROOTOFNEGATIVE is set to `NO`, an error occurs. When *expression* is negative and ROOTOFNEGATIVE is set to `YES`, SQRT returns the value `NA`.

Examples

Example 22–23 Calculating a Square Root

This example calculates the square root of `144`. The statement

```
SHOW SQRT(144)
```

produces the following result.

```
12.00
```

STARTOF

The STARTOF function returns the starting date of a time period in a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR.

Important: You can only use this function with dimensions of type DAY, WEEK, MONTH, QUARTER, or YEAR.You can*not* use this function for time dimensions that are implemented as hierarchical dimensions of type TEXT.

Return Value

DATE

Syntax

STARTOF(*dwmqy-dimension*)

Arguments

dwmqy-dimension
A dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR. When you have explicitly defined your own relation between dimensions of this type, you can use the name of this time relation here.

Notes

How STARTOF Works

STARTOF returns the first date of the time period that is first in the current status list of the dimension.

Phased or Multiple Periods

STARTOF is particularly useful when the dimension has a phase that differs from the default or when the time periods are formed from multiple weeks or years. For example, when the dimension has four-week time periods, the STARTOF function identifies the starting date of a particular four-week period.

Format of the Date

When you display the result returned by STARTOF, the date is formatted according to the date template in the DATEFORMAT option. When the day of the week or the name of the month is used in the date template, the day names specified in the DAYNAMES option and the month names specified in the MONTHNAMES option are used. You can use the result returned by STARTOF anywhere that a DATE value is expected.

DATE-to-TEXT Conversion

You can also use the result where a text value is expected. Oracle OLAP automatically converts the date to a text value, using the current template in the DATEFORMAT option to format the text value. When you want to override the current DATEFORMAT template, you can convert the date result to text by using the CONVERT function with a date-format argument.

Retrieving the Last Valid Date of a Time Period

The ENDOF function, which returns the last date of a time period.

Examples

Example 22–24 Finding the Fiscal Year Starting Date

The following statements define a year dimension (called FYEAR, for a fiscal year that ends in June), specify how the year will be formatted, add dimension values for fiscal years 1996 through 1998, and produce a report of the starting date of each fiscal year.

```
DEFINE fyear DIMENSION YEAR ENDING June
VNF 'FY<ff>'
MAINTAIN fyear ADD '30JUN96' '30JUN98'
REPORT W 14 STARTOF(fyear)
```

These statements produce the following output.

```
FYEAR          STARTOF(FYEAR)
---------      --------------
FY96           01JUL95
FY97           01JUL96
FY98           01JUL97
```

STATALL

The STATALL function indicates whether default status is currently in effect for a given dimension. That is, STATALL returns `YES` when STATLIST would return ALL. Otherwise, STATALL returns `NO`.

Return Value

BOOLEAN

Syntax

STATALL(*dimension*)

Arguments

dimension
A text expression whose value is the name of a dimension or dimension surrogate.

Notes

STATALL Compared to STATLIST
STATALL provides an alternative to running theSTATLIST program in order to determine whether or not the status of a specified dimension is ALL.

Examples

Example 22–25 Using STATALL

With the following statement, you can see whether the status of the MONTH dimension is ALL.

```
SHOW STATALL(month)
```

The return value is either `YES` or `NO`.

STATFIRST

The STATFIRST function returns the first value in the current status list of a dimension or a dimension surrogate, or in a valueset.

Return Value

The data type returned by STATFIRST is either the data type of the dimension or dimension surrogate value or an integer that indicates its position in the default status list of the dimension.The dimension value returned by STATFIRST is converted to a number or a text value, as appropriate to the context. See Example 22–26, "Assigning value of STATFIRST to Variables of Different Types" on page 22-60.

Syntax

STATFIRST(*dimension*)

Arguments

dimension
A text expression whose value is the name of a dimension, a dimension surrogate, or a valueset.

Examples

Example 22–26 Assigning value of STATFIRST to Variables of Different Types

The following statements

```
DEFINE textvar TEXT
textvar = STATFIRST(month)
SHOW textvar
```

produce this output.

```
Jun95
```

In contrast, these statements

```
DEFINE intvar INTEGER
intvar = STATFIRST(month)
SHOW INTVAR
```

produce this output.

```
6
```

Example 22–27 STATFIRST with KEEP

The following line from a program uses STATFIRST to limit `month` to all values in the status up to a value that has been stored previously in a variable called `onemonth`. The keyword KEEP means the new status is always a subset of the old status.

```
LIMIT month KEEP STATFIRST(month) TO onemonth
```

STATFIRST is used here, rather than a particular `month` value, so that the limit can work on any status list.

STATLAST

The STATLAST function returns the last value in the current status list of a dimension or a dimension surrogate, or in a valueset.

Return Value

The data type returned by STATLAST is either the data type of the dimension or dimension surrogate value or an integer that indicates its position in the default status list of the dimension. See "Automatic Data Conversion of Returned Dimension Values" on page 22-62.

Syntax

STATLAST(*dimension*)

Arguments

dimension
A text expression whose value is the name of a dimension, a dimension surrogate, or a valueset.

Notes

Automatic Data Conversion of Returned Dimension Values

The dimension value returned by STATLAST is converted to a number or a text value, as appropriate to the context. Suppose, for example, that `jun95` is the sixth `month` value but the last value in the current status list. The value of `STATLAST(month)` can be assigned either to a text variable or a numeric variable.

The following statements

```
DEFINE textvar TEXT
TEXTVAR = statlast(MONTH)
SHOW textvar
```

produce this output.

```
Jun95
```

In contrast, these statements

```
DEFINE intvar INTEGER
INTVAR = STATLAST(month)
SHOW INTVAR
```

produce this output.

```
6
```

Examples

Example 22–28 Setting Status with STATLAST

The following line from a program uses STATLAST to limit `month` to the values in the status, beginning with a month that has been stored previously in a variable called `onemonth`, and ending with the last value in the status.

```
LIMIT month KEEP onemonth TO STATLAST(month)
```

STATLAST is used here, rather than a particular `month` value, so that the limit can work on any status list.

STATLEN

The STATLEN function returns the number of values in the current status list of a dimension or a dimension surrogate, or in a valueset.

Return Value

INTEGER

Syntax

STATLEN(*dimension*)

Arguments

dimension
A text expression whose value is the name of a dimension, a dimension surrogate, or a valueset.

Examples

Example 22–29 Counting Months in Status

The following statement sends to the current outfile the number of months in the current status list of the `month` dimension.

```
SHOW STATLEN(month)
```

STATLIST

The STATLIST function returns a list of all values in the current status list of a dimension or dimension surrogate, or in a valueset. You can format the list to a specified width.

Return Value

STATLIST returns a list of TEXT values that contains either the dimension or dimension surrogate values themselves (for example, `Jan95`) or numbers (for example, `6`) that represent the positions of the values in the *default* status list.

The returned values are in the form *value* TO *value*, for example, `Jan96 TO Jun96`. When default status is in effect, it displays `ALL`. When the current status list or the valueset is empty, it displays `NULL`.

Syntax

STATLIST(*dimension* [*keyword*] [*width*])

Arguments

dimension
A text expression whose value is the name of a dimension, a dimension surrogate, or a valueset.

keyword
A keyword from Table 22–4, " Keywords for STATLIST". The keywords allow you to specify the form in which you want the values in the current status list to appear.

Table 22–4 Keywords for STATLIST

Keyword	Description
INTEGER	Specifies that STATLIST should return the list of values in the current status of a dimension in the form of the integer positions of those values in the default status list of the dimension.
TEXT	Specifies that STATLIST should return the list of values in the current status of a dimension in the form of the value names of those values (Default).

width
An optional integer or integer expression that specifies the width of the list in characters. When no width is specified, STATLIST uses the current value of the LSIZE option. LSIZE has a default value of `80`.

Notes

Compared to STATUS

The STATLIST function is employed by the STATUS command, which summarizes the status of a dimension. Use STATLIST rather than STATUS when you want to control the width or placement of the display.

Examples

Example 22–30 Producing a Status List with ROW

This example lists months in which total sales exceed $3,000,000.

These statements

```
LIMIT month TO TOTAL(sales, month) GE 3000000
ROW W 40 'Months with total sales over $3,000,000: '-
   W 40 STATLIST(month, 40)
```

produce the following output.

```
Months with total sales over $3,000,000: Jun95 TO Sep95, May96 TO Sep96
```

Example 22–31 Producing a Status List with SHOW

The following STATLIST command formats dimension values to a 20-character width.

```
LIMIT month TO 'Jan95' 'Mar95' 'May95' 'Jul96' 'Sep96' 'Nov96'
SHOW STATLIST(month 20)
```

These statements produce this output.

```
Jan95, Mar95, May95,
Jul96, Sep96, Nov96
```

This statement lists dimension positions.

```
SHOW STATLIST(month INTEGER 20)
```

This is the output.

```
1, 3, 5, 19, 21, 23
```

STATMAX

The STATMAX function returns the latest value in the current status list of a dimension or a dimension surrogate, or in a valueset.

Return Value

The data type returned by STATMAX is either the data type of the dimension or dimension surrogate value or an integer that indicates its position in the default status list of the dimension or surrogate. See "Automatic Conversion of Values Returned by STATMAX" on page 22-68.

Syntax

STATMAX(*dimension*)

Arguments

dimension
A text expression whose value is the name of a dimension, dimension surrogate, or valueset.

Notes

Automatic Conversion of Values Returned by STATMAX

The value that STATMAX returns is converted to a number or a text value as appropriate to the context. For example, suppose that the status of `month` is limited to `Jun95` to `Dec95` and that `Dec95` is the twelfth month in the default status list. The value of `STATMAX(month)` can be assigned either to a text variable or a numeric variable.

The following statements

```
DEFINE textvar TEXT
textvar = STATMAX(month)
SHOW textvar
```

produce this output.

```
Dec95
```

In contrast, these statements

```
DEFINE intvar INTEGER
intvar = STATMAX(month)
SHOW intvar
```

produce this output.

```
12
```

Examples

Example 22–32 STATMAX Used in a Title

The following statements from a program use STATMAX to determine the latest of the 10 months with the highest total sales.

```
LIMIT month TO BOTTOM 10 BASEDON TOTAL(sales, month)
SHOW JOINCHARS(STATMAX(month) ' is the latest month -
  of the ten months with the lowest sales.')
SHOW JOINCHARS('the months range from ' STATMIN(month) ' to '-
  STATMAX(month))
```

These statements produce the following sales report.

```
Dec96 is the latest month of the ten months with the lowest sales.
The months range from Jan95 to Dec96
```

When you used STATLAST instead of STATMAX, you could have produced a different value, because the LIMIT command arranged the `month` values by increasing sales rather than chronologically.

STATMIN

The STATMIN function returns the earliest value in the current status list of a dimension or a dimension surrogate, or in a valueset.

Return Value

Either a dimension or surrogate value or an integer that indicates the position of the value in the default status list of the dimension or surrogate. The return value varies depending on the *dimension* argument and the object receiving the return value. See "Automatic Data Type Conversion of Values Returned by STATMIN" on page 22-70.

Syntax

STATMIN(*dimension*)

Arguments

dimension
A text expression whose value is the name of a dimension, dimension surrogate, or valueset.

Notes

Automatic Data Type Conversion of Values Returned by STATMIN

The dimension value that STATMIN returns is converted, if necessary, to a number or a text value. For example, suppose the status of `month` is limited to `Jun95` to `Dec95`, and `Jun95` is the sixth `month` value in the default status list. The value of `STATMIN(month)` can be assigned either to a text variable, a numeric variable, or DATE variable.

The following statements

```
DEFINE textvar TEXT
textvar = STATMIN(month)
SHOW textvar
```

produce this output.

```
Jun95
```

In contrast, these statements

```
DEFINE intvar INTEGER
intvar = STATMIN(month)
SHOW intvar
```

produce this output.

```
6
```

Examples

Example 22–33 Using STATMIN in a Title

The following statements from a program use STATMIN to determine the earliest of the 10 months with the highest total sales.

```
LIMIT month TO TOP 10 BASEDON TOTAL(sales, month)
SHOW JOINCHARS(STATMIN(month) ' is the earliest of the -
  ten months with the highest sales.')
SHOW JOINCHARS( 'The months range from ' statmin(month) ' TO '-
  statmax(month) )
```

The preceding statements produce the following sales report.

```
May95 is the earliest of the ten months with the highest sales.
The months range from May95 to Sep96
```

Example 22–34 Comparing to STATFIRST

In the following example, you can see the difference between STATMIN and STATFIRST, which returns the first value in the current status list.

Assume that you issue the following statements.

```
LIMIT month TO TOP 10 BASEDON TOTAL(sales, month)
REPORT WIDTH 20 TOTAL(sales, month)
```

When the proceeding statements execute, the following report is produced.

```
MONTH          TOTAL(SALES, MONTH)
-------------- --------------------
Jul96                  3,647,085.39
Jun96                  3,458,438.30
Jul95                  3,414,210.05
Aug96                  3,246,601.97
Jun95                  3,228,824.80
Sep96                  3,215,883.93
May96                  3,112,854.59
Aug95                  3,044,694.29
Sep95                  3,006,242.58
May95                  2,908,539.45
```

Notice that the `month` values in this report are arranged by decreasing sales rather than chronologically, and this is now the order in which they occur in the status list:

- STATMIN gives the *chronologically* first value in the status (though it is positionally last) as illustrated in the following statement and output.

  ```
  SHOW STATMIN(month)
  May95
  ```

- STATFIRST gives the value that is *positionally* first in the status (though it is chronologically eighth) as illustrated in the following statement and output.

  ```
  SHOW STATFIRST(month)
  Jul96
  ```

STATRANK

The STATRANK function returns the position of a dimension or dimension surrogate value in the current status list or in a valueset.

Return Value

INTEGER

Syntax

STATRANK(*dimension* [*value*])

Arguments

dimension
A text expression whose value is the name of a dimension, dimension surrogate, or valueset.

value
The value you want to check, which is an appropriate data type for *dimension.* For example, *value* can be a text expression for an ID or TEXT dimension, an integer for an INTEGER dimension, a date for a time dimension, or a combination of values enclosed by angle brackets for conjoint or concat dimensions. The value of a text expression must have the same capitalization as the actual dimension value. When you use a text expression, it must be a single-line value.

When you specify the value of a conjoint dimension, be sure to enclose the value in angle brackets, and separate the base dimension values with a comma and space. When you specify the value of a concat dimension, be sure to enclose the value in angle brackets, and separate the base dimension name from the value with a colon and space.

When you do not specify *value,* STATRANK returns the position of the current value. When you specify the name of a valid dimension value that is not in the current status list or in the valueset, STATRANK returns `NA`.

Examples

Example 22–35 Using STATRANK to Identify Position Numbers

Suppose you want to produce a report of the top five months by total sales, displayed in order as a numbered list. You can use STATRANK to number each month. Assume that you have written a report program with the following defintion and contents.

```
DEFINE sales.rpt PROGRAM
PROGRAM
LIMIT month TO TOP 5 BASEDON TOTAL(sales, month)
SHOW 'Top five months by total sales:'
for month
    ROW WIDTH 4 JOINCHARS(STATRANK(month) '.') WIDTH 5 month
END
```

The report program produces the following output.

```
Top five months by total sales:
1.   Jul96
2.   Jun96
3.   Jul95
4.   Aug96
5.   Jun95
```

After executing the `sales.rpt` program, you can use the SHOW command with the STATRANK function to learn the position of a particular month within the top five months by total sales.

The following statement

```
SHOW STATRANK(month Jun96)
```

produces this output.

```
2
```

Example 22–36 Using STATRANK When the Dimension Is a Conjoint Dimension

When the *dimension* that you specify is a conjoint dimension, then the entire *value* must be enclosed in single quotes.

For example, suppose the analytic workspace already has a `region` dimension and a `product` dimension. The `region` dimension values include `East`, `Central`, and `West`. The `product` dimension values include `Tents`, `Canoes`, and `Racquets`.

The following statements define a conjoint dimension, and add values to it.

```
DEFINE reg.prod DIMENSION <region product>
MAINTAIN reg.prod ADD <'East', 'Tents'> <'West', 'Canoes'>
```

To specify base positions, use a statement such as the following. Because the position of `East` in the `region` dimension is `1` and the position of `Tents` in the `product` dimension is `1`, the following statement returns the position of the corresponding `reg.prod` value.

```
SHOW STATRANK(reg.prod '<1, 1>')

1
```

To specify base text values, use a statement such as the following.

```
SHOW STATRANK(reg.prod '<\'East\', \'Tents\'>')

1
```

Example 22–37 Using STATRANK When the Dimension Is a Concat Dimension

When the *dimension* that you specify is a concat dimension, then the entire *value* must be enclosed in single quotes. The following statement defines a concat dimension named `reg.prod.ccdim` that has as its base dimensions `region` and `product`.

```
DEFINE reg.prod.ccdim DIMENSION CONCAT(region product)
```

A report of `reg.prod.ccdim` returns the following.

```
REG.PROD.CCDIM
----------------------
<Region: East>
<Region: Central>
<Region: West>
<Product: Tents>
<Product: Canoes>
<Product: Racquets>
```

To specify a base dimension position, use a statement such as the following. Because the position of `racquets` in the `product` dimension is `3`, the statement returns the position in `reg.prod.ccdim` of the `<product: Racquets>` value.

```
SHOW STATRANK(reg.prod.ccdim '<product: 3>')

6
```

To specify base dimension text values, use a statement such as the following.

```
SHOW STATRANK(reg.prod.ccdim '<product: Tents>')

4
```

STATUS

The STATUS program sends to the current outfile the status of one or more dimensions, dimension surrogates, or valuesets, or the status of all dimensions in an analytic workspace.

When you specify one or more dimension, dimension surrogate, or valueset names, Oracle OLAP produces the status of only those objects. When you use the AW keyword and specify the name of an attached analytic workspace, Oracle OLAP produces the status of every dimension in that analytic workspace. When you do not specify any argument, STATUS produces the current status of all the dimensions (not dimension surrogates or valuesets) in the current analytic workspace. However, STATUS does not display the status of the NAME dimension unless you specify `STATUS NAME`.

Return Value

TEXT

Syntax

STATUS *name*... | AW [*workspace-name*]

Arguments

name

The name of a dimension or valueset in the analytic workspace. You can also specify the name of a dimensioned analytic workspace object, such as a variable, formula, relation, or named composite. In this case, the status of each dimension of *name* is produced, unless the dimension is included in an unnamed composite.

AW [*workspace-name*]

Specifies that STATUS should produce the status of every dimension in *workspace-*name; *workspace-name* is the name of an analytic workspace.

Notes

STATUS Output

When all values of a dimension are in the current status or in a valueset, in the original order, STATUS displays `ALL`. STATUS shortens any series of three or more values in their original order to *value-1* TO *value-n*. In the case of the dimension NAME, however, STATUS does not shorten a series of three or more values.

Empty STATUS

When a dimension, dimension surrogate, or valueset has no values (for example, a recently defined object for which you have not yet supplied values), STATUS produces `NULL` for that dimension, dimension surrogate, or valueset. When you are in an analytic workspace in which no objects have been defined, STATUS produces the message, `There are no dimensions in your current analytic workspace.`

Examples

Example 22–38 Discovering the Current Status of Certain Dimensions

Use STATUS to produce the current status of the dimensions `month` and `district`.

The following statement

```
STATUS month district
```

produces this output.

```
The current status of MONTH is:
Jan95 TO Dec96
The current status of DISTRICT is:
Boston, Chicago, Denver
```

Example 22–39 Discovering the Status of the Dimensions of a Variable

Use STATUS to produce the current status of all the dimensions of the variable `sales`.

The following statement

```
STATUS sales
```

produces this output.

```
The current status of MONTH is:
Jan95 TO Dec96
The current status of PRODUCT is:
ALL
The current status of DISTRICT is:
Boston, Chicago, Denver
```

STATVAL

The STATVAL function returns the dimension value that corresponds to a specified position in the current status list of a dimension or a dimension surrogate, or in a valueset.

Return Value

The data type returned by STATVAL is either the data type of the dimension or dimension surrogate value or an integer that indicates its position in the default status list of the dimension. The dimension value that STATVAL returns is converted to a number or a text value, as appropriate to the context. To ensure that STATVAL returns an integer value, specify the INTEGER keyword. See Example 22–41, "Ensuring that STATVAL Returns an INTEGER" on page 22-81.

Syntax

STATVAL(*dimension position* [INTEGER])

Arguments

dimension
A text expression whose value is the name of a dimension, a dimension surrogate, or a valueset.

position
An integer or integer expression that specifies the position in the current status list of a dimension or a valueset. When you specify a position that has no values, STATVAL returns `NA`.

INTEGER
Specifies that STATVAL must return an integer that represents the position of the dimension value in the default status list.

Notes

STATVAL in a FOR Loop

In a FOR loop over a dimension, the status is limited to a single dimension value for each iteration of the loop. Therefore, STATVAL has a value only for position `1`. For other positions, STATVAL returns `NA`.

Examples

Example 22–40 STAVAL with Qualified Data References

Suppose you want to know the sales figures for the month ranked fifth among the 10 months with the highest total sales. After limiting `month` to the `TOP 10`, use STATVAL in a qualified data reference to produce sales figures for the month ranked fifth.

```
LIMIT month TO TOP 10 BASEDON TOTAL(sales, month)
REPORT month
```

These statements produce the following report.

```
MONTH
--------------
Jul96
Jun96
Jul95
Aug96
Jun95
Sep96
May96
Aug95
Sep95
MAY95
```

Using STATVAL in the following REPORT command produces a different report.

```
REPORT W 8 DOWN district HEADING -
   JOINCHARS('Sales: 5th of Top Ten - ' STATVAL(month 5)) -
   sales(month STATVAL(month 5))
```

This is the report produced by the preceding statement.

```
         ------------Sales: 5th of Top Ten - Jun95-------------
         -----------------------PRODUCT------------------------
DISTRICT   Tents      Canoes    Racquets  Sportswear  Footwear
-------- ---------- ---------- ---------- ---------- ----------
Boston    88,996.35 147,412.44  90,840.60  75,206.30 144,162.66
Atlanta  110,765.24 106,327.17 109,695.31 155,652.78 146,364.99
Chicago   70,908.96 108,039.05 100,030.29 104,900.66 148,386.81
Dallas   128,692.56  71,899.23 176,452.58 164,823.10  32,421.25
Denver    91,717.46  99,099.20 140,961.37  99,951.60  70,149.77
Seattle  113,806.48 143,037.62  54,926.87  57,739.03  75,457.04
```

Notice that the qualified data reference in the following statement means "sales for the fifth month in the *default* status of `month`."

```
sales(month 5)
```

While the qualified data reference in the followng statement means "sales for the fifth month in the *current* status of `month`."

```
sales(month STATVAL(month 5))
```

The following statements show the different values that are returned for a qualified data reference of `month` and for STATVAL with `month` as an argument.

```
SHOW month(month 5)
SHOW STATVAL(month 5)
```

The preceding statements produce the following output.

```
May95
Jun95
```

Example 22–41 Ensuring that STATVAL Returns an INTEGER

Depending on the context, STATVAL may return an `INTEGER` value without your specifying the INTEGER keyword.

The following statements

```
LIMIT month TO 'Jun95' TO 'Dec95'
SHOW STATVAL(month 3)
```

produce this output.

```
Aug95
```

With the INTEGER keyword,

```
SHOW STATVAL(month 3 INTEGER)
```

the following output is produced.

```
8
```

23

STDDEV to TRACKPRG

This chapter contains the following OLAP DML statements:

STDDEV

The STDDEV function calculates the standard deviation of the values of an expression.

Return Value

DECIMAL

Syntax

STDDEV(*expression* [[STATUS] *dimensions*])

Arguments

expression
The numeric expression whose standard deviation is to be calculated.

STATUS
Can be specified when one or more of the dimensions of the result of the function are not dimensions of the expression. (See the description of the *dimensions* argumentw.) When you specify the STATUS keyword when this is not the case, then an error results.

In cases where one or more of the dimensions of the result of the function are not dimensions of the expression, the STATUS keyword may be *required* in order for the function to be processed successfully, or the STATUS keyword may provide a performance enhancement. See "The STATUS Keyword" on page 23-4.

dimensions
The dimensions of the result. By default, STDDEV returns a single value. When you indicate one or more dimensions for the results, STDDEV calculates a standard deviation along the specified dimension(s) and returns an array of values. Each dimension must be either a dimension of *expression* or related to one of its dimensions. When it is a related dimension, you can specify the name of the relation instead of the dimension name. This enables you to choose the relation to use when there is more than one.

Notes

NA Values

STDDEV is affected by the NASKIP option. When NASKIP is set to `YES` (the default), STDDEV ignores `NA` values and returns the standard deviation of the values that are not `NA`. When NASKIP is set to `NO`, STDDEV returns `NA` when any value in the calculation is `NA`. When all data values for a calculation are `NA`, STDDEV returns `NA` for either setting of NASKIP.

Calculating over a Time Dimension

When *expression* is dimensioned by a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR, you can specify any other dimensions of type DAY, WEEK, MONTH, QUARTER, or YEAR as a related *dimension.* Oracle OLAP uses the implicit relation between the dimensions. To control the mapping of one dimensions of type DAY, WEEK, MONTH, QUARTER, or YEAR to another (for example, from weeks to months), you can define an explicit relation between the two dimensions and specify the name of the relation as the *dimension* argument to the STDDEV function.

For each time period in the related dimensions of type DAY, WEEK, MONTH, QUARTER, and YEAR, Oracle OLAP calculates the standard deviation of the data values of the source time periods that end in the target time period. This method is used regardless of which dimension has the more aggregate time periods. To control the way in which data is aggregated or allocated between the periods of two dimensions of type DAY, WEEK, MONTH, QUARTER, and YEAR, you can use the TCONVERT function.

The STATUS Keyword

When one or more of the dimensions of the result of the function are not dimensions of the expression, Oracle OLAP creates a temporary variable to use while processing the function. When you specify the STATUS keyword, then Oracle OLAP uses the current status instead of the default status of the related dimensions for calculating the size of this temporary variable.

When the size of the temporary variable for the results of the function would exceed 2 gigabytes, you *must* specify the STATUS keyword in order for Oracle OLAP to successfully execute the function. When the dimensions of the expression are limited to a few values and are physically fragmented, you can specify the STATUS keyword to improve the performance of the function.

When you use STDDEV with the STATUS keyword in an expression that requires going outside of the status for results (for example, with the LEAD or LAG functions or with a qualified data reference), the results outside of the status will be returned as NA.

Examples

Example 23–1 Calculating the Standard Deviation of Monthly Sales

This example calculates the average number of tents sold during the first three months of 1996, along with the standard deviation from that average.

```
LIMIT district TO ALL
LIMIT month TO 'Jan96' TO 'Mar96'
LIMIT product TO 'Tents'
REPORT HEADING 'Average' AVERAGE(units month) -
   HEADING 'Stddev' STDDEV(JAN96             262.33      49.32
FEB96             247.83      57.37
MAR96  units month)
```

These statements produce the following output.

```
MONTH          Average     Stddev
-------------- ---------- ----------
Jan96             262.33      49.32
Feb96             247.83      57.37
Mar96             320.50      68.17
```

STDHDR

The STDHDR program generates the standard Oracle OLAP heading at the top of every page of report output. The heading output is sent to the current outfile.

Syntax

```
STDHDR
```

Notes

The Standard Heading

The standard running page heading consists of two lines. The first line includes the date and time on the left and the page number on the right. The second line is blank.

Setting LSIZE

LSIZE must be set to a value of at least `29` before you use STDHDR. Otherwise the heading will not look right. A value less than `26` produces an error. The default for LSIZE is `80`.

Creating a Custom Heading

When PAGING is set to `YES`, Oracle OLAP automatically inserts the standard heading at the top of each page of output. To get a different heading you must write a program that produces the heading and set the PAGEPRG option to the name of that program. To return to the standard heading, set PAGEPRG to `'STDHDR'`. (See PAGEPRG for more information.)

Using STDHDR in a Heading Program

When you use PAGEPRG to specify a heading program, you can still use the standard heading in your custom heading by executing STDHDR as part of your program. Generally, you place STDHDR before the statements that produce your customized heading. See Example 23–2, "Creating a Custom Heading for a Report" on page 23-7.

The STDHDR Program

The STDHDR program uses the HEADING and BITAND commands, as follows.

```
HEADING L W 8 <CONVERT(TODAY, TEXT,'<DD><MTXTL><YY>') TOD> -
        R W LSIZE-25 'Page ' L W 6 D 0 PAGENUM
BLANK
```

Examples

Example 23–2 Creating a Custom Heading for a Report

Suppose you would like each page of your report to include the standard header and also the customized title "Annual Sales Report." To accomplish this, define a small PAGEPRG program called report.head.

```
DEFINE report.head PROGRAM
PROGRAM
CALL STDHDR
HEADING WIDTH LSIZE CENTER 'Annual Sales Report'
BLANK
END
```

In your report program, set PAGING to `YES`, and specify the preceding program to execute after every page break by setting the PAGEPRG option to `'REPORT.HEAD'` (see the PAGEPRG option for further information). When you run the report, each page will contain the following combination of the standard heading and your custom heading.

```
18Jan97  15:05:16                                   PAGE 1

                   Annual Sales Report
```

SUBSTR

The SUBSTR function returns a portion of string, beginning at a specified character position, and a specified number of characters long. SUBSTR calculates lengths using characters as defined by the input character set.

To retrieve a portion of string based on bytes, use SUBSTRB.

Return Value

The return value is the same data type as string.

Syntax

SUBSTR (*string* , *position* [, *substring_length*])

Arguments

string
A text expression that is the base string from which the substring is created.

position
The position at which the first character of the returned string begins.

- When *position* is `0` (zero), then it is treated as `1`.
- When *position* is positive, then the function counts from the beginning of *string* to find the first character.
- When *position* is negative, then the function counts backward from the end of *string*.

substring_length
The number of characters in the returned string. When you do not specify a value for this argument, then the function returns all characters to the end of *string*. When you specify a value that is less than `1`, the function returns `NA`.

Examples

The following example returns several specified substrings of "abcdefg".

```
SHOW SUBSTR('abcdefg',3,4)
cdef

SHOW SUBSTR('abcdefg',-5,4)
cdef
```

SUBSTRB

The SUBSTRB function returns a portion of string, beginning at a specified byte position, and a specified number of bytes long.

To retrieve a portion of string based on characters, use SUBSTR.

Return Value

The return value is the same data type as string.

Syntax

SUBSTRB (*string* , *position* [, *substring_length*])

Arguments

string
A text expression that is the base string from which the substring is created.

position
The position at which the first byte of the returned string begins.

- When *position* is `0` (zero), then it is treated as `1`.
- When *position* is positive, then the function counts from the beginning of *string* to find the first byte.
- When *position* is negative, then the function counts backward from the end of *string*.

substring_length
The number of bytes in the returned string. When you do not specify a value for this argument, then the function returns all bytes to the end of *string*. When you specify a value that is less than `1`, the function returns `NA`.

Examples

Assume a double-byte database character set.

```
SHOW SUBSTRB('abcdefg',5,4.2)
cd
```

SUBTOTAL

The SUBTOTAL function returns the value of one of the subtotals accumulated in a report. You normally use the SUBTOTAL function in a ROW command to include a subtotal or grand total in the report. Since Oracle OLAP maintains 32 running totals for each column, you can include up to 32 levels of subtotals

Note: In the REPORT command, use the GRANDTOTALS and SUBTOTALS keywords to include rows of grand totals and subtotals.

Return Value

DECIMAL

Syntax

SUBTOTAL(*n*)

Arguments

n

An integer value that indicates the level of a running total for each numeric column in a report. For example, a "Total" may be a level 1 subtotal and a "Grand Total" may be a level 2 subtotal. Because it is possible to have up to 32 levels of running totals in a column, *n* must be an integer between 1 and 32. SUBTOTAL returns the value of this subtotal for the current column and then resets the value of subtotal *n* to zero.

Notes

Resetting Subtotals Automatically

When you use the SUBTOTAL function in a ROW command to include a subtotal of the current column, the subtotal at that level is reset to zero.

Resetting Subtotals with ZEROTOTAL

When you use the ROW command to produce a report, you can use the ZEROTOTAL command to reset any subtotal of any column to zero. Normally, you should do this at the beginning of a report program to make sure all totals begin at zero.

Referring to Subtotals

The numbers by which the 32 subtotals are referenced (`1` to `32`) have no intrinsic significance. All the subtotals are the same until you reference them.

NA Values

SUBTOTAL ignores `NA` values. When all values are `NA`, SUBTOTAL returns zero.

Decimal Overflow

When a "decimal overflow" condition occurs while subtotals are being accumulated (that is, an out-of-range value is generated), all subtotals for the affected column are set to `NA` and processing continues when the DECIMALOVERFLOW option is set to `YES`. The subtotals for the column will continue to be `NA` until they are reset by a ZEROTOTAL command. When DECIMALOVERFLOW is `NO`, an error occurs when a decimal overflow condition occurs.

Examples

Example 23–3 Calculating Subtotals and Grand Totals in a Report

In a sales report, suppose you want to show a subtotal for each region. You also want to see a grand total of all sales at the end of the report. You can use `SUBTOTAL(1)` to produce the subtotal for each region. This subtotal is reset to `0` each time you use it, so it provides a separate subtotal for each region. At the end of the report you can use `SUBTOTAL(2)` to produce the grand total. Since you have not yet used it in your report, it holds a total of the sales figures for all regions.

```
LIMIT month TO FIRST 3
LIMIT region TO ALL
ZEROTOTAL ALL
FOR region
   DO
   ROW region
   LIMIT DISTRICT TO region
   FOR district
      DO
      ROW INDENT 5 district ACROSS month: sales
      DOEND
   ROW INDENT 5 'Total' ACROSS month: OVER '-' SUBTOTAL(1)
   BLANK
   DOEND
ROW 'Grand Total' ACROSS month: OVER '=' SUBTOTAL(2)
```

The program produces the following output.

```
East
     Boston     32,153.52  32,536.30  43,062.75
     Atlanta    40,674.20  44,236.55  51,227.06
               ---------- ---------- ----------
     Total      72,827.72  76,772.85  94,289.81
Central
     Chicago    29,098.94  29,010.20  39,540.89
     Dallas     47,747.98  50,166.81  67,075.44
               ---------- ---------- ----------
     Total      76,846.92  79,177.01 106,616.33
West
     Denver     36,494.25  33,658.24  45,303.93
     Seattle    43,568.02  41,191.28  51,547.23
               ---------- ---------- ----------
     Total      80,062.27  74,849.52  96,851.16

               ========== ========== ==========
Grand Total    229,736.91 230,799.38 297,757.30
```

SWITCH

The SWITCH command provides a multipath branch in a program. The specific path taken during program execution depends on the value of the control expression that is specified with SWITCH. You can use SWITCH only within programs.

Syntax

```
SWITCH control-expression
  DO
    CASE case-expression1:
       statement 1.1
       ...
       statement 1.n
       BREAK
    CASE case-expression2:
       statement 2.1
       ...
       statement 2.n
       BREAK
    [DEFAULT:
       statement n.1
       ...
       statement n.n
       BREAK]
  DOEND
```

Arguments

control-expression

The *control-expression* argument determines the case label to which program control is transferred by the SWITCH command. When the SWITCH command is executed, *control-expression* is evaluated and compared with each of the CASE label expressions in the program. When a match is found, control is transferred to that case label. When no match is found, control transfers to the DEFAULT label (if present) or to the statement following the DOEND for SWITCH.

CASE *case-expression1*, CASE *case-expression2*, ...

The CASE labels whose expressions (*case-expression1*, *case-expression2*, ...) specify the different cases you want to handle. When *control-expression* matches *case-expression,* program control is transferred to that CASE label. The CASE label expressions are evaluated at the time the program is run, in the order they appear, until a match is found.

The DEFAULT label is optional. It identifies a special case to which control should be transferred when none of the *case-expressions* matches the *control-expression.* When you omit DEFAULT, and no match is found, control is transferred to the statement that follows the DOEND for SWITCH.

All the CASE labels (including DEFAULT) for a SWITCH command must be included within a DO/DOEND bracket immediately following the SWITCH command. Because *case-expression* is a label, it must be followed by a colon (:). The statements to be executed in a given case must follow the label. Normally, the last statement in a case should be BREAK, which transfers control from SWITCH to the statement that follows the DOEND for SWITCH.

When you omit BREAK (or RETURN, SIGNAL, and so on) at the end of a case, the program will go on to execute the statements for the next case as well. Normally, you do not want this to happen. However, when you plan to execute the same statements for two cases, you can use this to your advantage by placing both CASE labels before the statements.

Notes

Control- and Case-Expressions

The SWITCH *control-expression* can have any data type, as can the *case-expressions.* The various *case-expressions* can have different data types. When you specify the name of a dimension (as a literal, non-quoted text expression) as the *control-expression* or *case-expression*, Oracle OLAP uses the first value in the dimension's current status list, not the dimension name, as it searches for a match. When the dimension has no values in the status list, Oracle OLAP uses the value `NA`. An `NA` *control-expression* will match the first `NA` *case-expression.*

Ampersand Substitution

Avoid using ampersand substitution in a SWITCH *control-expression* or in a CASE label *case-expression.* Ampersands will produce unpredictable, and usually undesirable, results.

Multiple SWITCH Commands

You can include more than one SWITCH command in a program. You can also nest SWITCH commands. When a program contains multiple SWITCH commands, each can have its own DEFAULT label, even when the SWITCH commands are nested.

Transferring Control

While BREAK is commonly used to transfer program control within a SWITCH command, it is not the only such statement you can use. You can also use statements such as CONTINUE, GOTO, RETURN, and SIGNAL. Keep in mind that you can use CONTINUE only when the SWITCH command is within a FOR or WHILE loop. See also the entries for these statements and for DO ... DOEND.

Examples

Example 23–4 Multipath Branching Using SWITCH in a Program

The following program lines produce one of several types of reports. Before the SWITCH command, the program determines which type of report the user wants and places the value `Market` or `Finance` in the variable `userchoice`. The program switches to the case label that matches that name and produces the report.

When the report finishes, the BREAK command transfers control to the cleanup section after the DOEND.

```
SWITCH userchoice
    DO
         CASE 'Market':
            ...
            BREAK
         CASE 'Finance':
            ...
            BREAK
         DEFAULT:
            ...
            BREAK
    DOEND
cleanup:
...
```

SYSDATE

The SYSDATE function returns the current date and time. The format of the date is controlled by the NLS_DATE_FORMAT option. The default datetime format (`DD-MM-RR`) does not display the time.

Return Value

DATETIME

Syntax

SYSDATE

Examples

Example 23–5 Displaying the Current Date

The following statement:

```
SHOW SYSDATE
```

displays the current date:

```
08-Sep-00
```

SYSINFO

The SYSINFO function provides information about the Oracle user ID for the current session.

Return Value

TEXT

Syntax

SYSINFO (*keyword*)

where *keyword* is one of the following:

```
USER
ROLES
PROFILES
HOSTNAME
OSUSER
INSTNAME
PID
PROGNAME
CHOSTNAME
COSUSER
TERMNAME
```

Arguments

USER
Returns a TEXT value that indicates the user ID under which the Oracle Database session is running.

ROLES
Returns a multiline TEXT value that lists the roles that apply to the user ID of the session.

PROFILES
Returns a multiline TEXT value that lists the profiles that apply to the user ID of the session.

OSUSER
Returns TEXT value that indicates the operating system username under which the Oracle Database server is running.

INSTNAME
Returns a TEXT value that is the instance name of the Oracle Database server.

PID
Returns a TEXT value that is the operating system id number of your Oracle Database session.

HOSTNAME
Returns a TEXT value that is the hostname of the Oracle Database server.

PROGNAME
Returns a TEXT value that is identifies the client which is connecting to the Oracle Database.

CHOSTNAME
Returns a TEXT value that is the host name of the client.

COSUSER
Returns a TEXT value that is the operating system user name of the client.

TERMNAME
Returns a TEXT value that is the terminal name of the client.

Notes

USERID Option and SYSINFO(USER) Function

The value of USERID is also the value that `SYSINFO(USER)` returns.

Examples

Example 23–6 Obtaining the User ID

You can use the SYSINFO function to obtain the user of the current session.

```
SHOW SYSINFO(USER)
```

produces output like the following.

```
Scott
```

SYSTEM

The SYSTEM function identifies the platform on which Oracle OLAP is running.

Data type

TEXT

Syntax

SYSTEM

Notes

Relevance of the Platform

Because Oracle OLAP is incorporated in the Oracle Database, the operating system on which it is running should not be an important factor in its behavior.

> **Note:** All references to external files are made through directory objects, which are not platform specific

Examples

Example 23–7 Displaying the Platform

Issuing the following SYSTEM statement on Intel NT returns the value `NTX86`.

```
SHOW SYSTEM

NTX86
```

TALLY

The TALLY function counts the number of values of a dimension that correspond to each value of one or more related dimensions.

Return Value

INTEGER

Syntax

TALLY(*dimension* [[STATUS] *related-dimensions*])

Arguments

dimension
A dimension whose values are to be counted. When you specify *related-dimensions,* TALLY counts the number of values of *dimension* that correspond to each value of a single related dimension, or to each combination of values of two or more related dimensions. When you do not specify *related-dimensions,* TALLY counts the number of values in the dimension. Only values in the current status of *dimension* are counted.

STATUS
May be specified when using one or more related dimensions for the results of the function. (See the description of the *related-dimensions* argument.) When you specify the STATUS keyword without specifying *related-dimensions,* Oracle OLAP produces an error.

When you use related dimensions, the STATUS keyword may be required in order for Oracle OLAP to successfully process the function, or the STATUS keyword may provide a performance enhancement. See "TALLY with STATUS" on page 23-23.

related-dimensions
One or more related dimensions for the results. These must be related to *dimension.* Alternatively, you can specify the name of the relation instead of the dimension name. This enables you to choose which relation is used when there is more than one. When no *related-dimensions* are specified, TALLY returns the total number of values in the current status of *dimension.*

Notes

TALLY with NA

TALLY returns NA for any *related-dimension* position that has no *dimension* values corresponding to it.

TALLY with Time Dimensions

When *expression* is dimensioned by a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR, you can specify any other dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR as a *related-dimension.* Oracle OLAP uses the implicit relation between the dimensions. To control the mapping of one DAY, WEEK, MONTH, QUARTER, or YEAR dimension to another (for example, from weeks to months), you can define an explicit relation between the two dimensions and specify the name of the relation as the *related-dimension* argument to the TALLY function.

For each time period in the related dimension, Oracle OLAP tallies all the source time periods that end in the target time period. This method is used regardless of which dimension has the more aggregate time periods.

TALLY with STATUS

When you use TALLY with related dimensions, Oracle OLAP creates a temporary variable to use while processing the function. When you specify the STATUS keyword, then Oracle OLAP uses the current status instead of the default status of the related dimensions for calculating the size of this temporary variable.

When the size of the temporary variable for the results of the function would exceed 2 gigabytes, you *must* specify the STATUS keyword in order for Oracle OLAP to successfully execute the function. When *dimension* is limited to a few values that are physically fragmented, you can specify the STATUS keyword to improve the performance of the function.

When you use TALLY with the STATUS keyword in an expression that requires going outside of the status for results (for example, with the LEAD or LAG functions or with a qualified data reference), the results outside of the status will be returned as NA.

Examples

Example 23–8 Breaking Out TALLY Results

Here you use TALLY to determine how many products are produced by each division. The `division.product` relation records the division to which each product belongs. The following is a report of `division.product`.

```
PRODUCT          DIVISION.PRODUCT
---------------------------------
Tents          Camping
Canoes         Camping
Racquets       Sporting
Sportswear     Clothing
Footwear       Clothing
```

The following statement includes TALLY to present the number of products produced by each division.

```
REPORT HEADING 'Products' TALLY(product, division)
```

The statement produces this report.

```
DIVISION        Products
-------------------------
Camping                2
Sporting               1
Clothing               2
```

TAN

The TAN function calculates the tangent of an angle expression.

Return Value

DECIMAL

Syntax

TAN(*expression*)

Arguments

expression
A numeric expression that contains an angle value, which is specified in radians.

Examples

Example 23–9 Calculating the Tangent of an Angle

This example calculates the tangent of an angle of `1` radian. The statements

```
DECIMALS = 5
SHOW TAN(1)
```

produce the following result.

```
1.55741
```

TANH

The TANH function calculates the hyperbolic tangent of an angle expression.

Return Value

DECIMAL

Syntax

TANH(*expression*)

Arguments

expression
A numeric expression that contains an angle value, which is specified in radians.

Examples

Example 23–10 Calculating the Hyperbolic Tangent of an Angle

This example calculates the hyperbolic tangent of an angle of `1` radian. The statements

```
DECIMALS = 5
SHOW TANH(1)
```

produce the following result.

```
0.76159
```

TCONVERT

The TCONVERT function converts time-series data from one dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR to another dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR. You can specify an aggregation method or an allocation method to use in the conversion.

> **Important:** You can only use this function with dimensions of type DAY, WEEK, MONTH, QUARTER, or YEAR.You can*not* use this function for time dimensions that are implemented as hierarchical dimensions of type TEXT.

Return Value

The value returned by the TCONVERT function depends on the type of conversion you specify and the type of the dimension being converted.

Syntax

TCONVERT(*expression time-dimension method* [*method*])

where:

method is one of following syntax:

SUM|AVERAGE|LAST [<u>BY PERIOD</u>|BY DAY] [STATUS|NOSTATUS]

SPLIT|REPEAT|INTERPOLATE [<u>BY PERIOD</u>|BY DAY]

Arguments

expression
An expression whose values you want to convert. *Expression* must be dimensioned by a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR. This dimension is referred to as the *source* dimension. Usually *expression* is numeric, but with some conversion methods you can also convert text data. See "Converting Text Data" on page 23-33.

time-dimension
The DAY, WEEK, MONTH, QUARTER, or YEAR dimension to which you want to convert the *expression.* This dimension is referred to as the *target* dimension.

method

The method to use for converting data from the source dimension to the target dimension. You can specify an aggregation method or an allocation method:

- Aggregation methods are SUM, AVERAGE, and LAST. They are typically used to convert data from smaller time periods to larger time periods (for example, months to years).
- Allocation methods are SPLIT, REPEAT, and INTERPOLATE. They are typically used to convert data from larger to smaller time periods (for example, years to quarters). The allocation methods all use the full default status of the source dimension to determine the periods that contribute to the allocation.

Except for a case in which the source dimension and target dimension have overlapping periods of equal length (as with a calendar year and a fiscal year), you can specify both an aggregation method and an allocation method. See "Compatible Aggregation and Allocation Methods" on page 23-31 and "Using Both Aggregation and Allocation" on page 23-32.

For all methods, results are calculated for the values in the current status of the target dimension.

The results you obtain depend on the method you specify and on whether you convert data between dimensions with periods of equal length or unequal length. See "Using Both Aggregation and Allocation" on page 23-32, "Overlapping Periods of Equal Length" on page 23-32, and "Substituting a Compatible Method" on page 23-33.

SUM [BY PERIOD]

Aggregates data to a target period by totaling the data of the contributing source periods. For each target period, SUM PERIOD returns the total for all the source periods that end in the target period. SUM uses the implicit relation between the source and target dimensions.

SUM BY DAY

Weights each source value according to the portion of target days it represents. For each target period, SUM BY DAY multiplies each contributing source period value by a weighting factor that has this form where *source-days-in-target* is the Number of *source-period days* that actually fall in target period and total-days-in-period is the total number of days in source period:

source-days-in-target / total-days-in-period

SUM BY DAY then returns the total of these weighted source values. When you use SUM BY DAY, the value of an individual source period may be apportioned across adjacent target periods.

For example, suppose you convert weekly data to monthly data. When three days of a week fall in January and four fall in February, then SUM BY DAY adds `3/7` of the data for that week to the January total and `4/7` to the February total. In contrast, SUM BY PERIOD adds the entire data value for the week to the February total (since the week ends in February).

As another example, suppose you want to convert calendar year data to a fiscal year ending in June. Calendar year 1996 (`Cal96`) is the only calendar year that ends in fiscal year 1997 (`Fy97`). The SUM BY PERIOD method assigns the value for `Cal96` to `Fy97`. In contrast, SUM BY DAY apportions the `Cal96` value to the fiscal years `Fy96` and `Fy97`, according to the number of calendar days that fall in each fiscal year. Of the 366 days of `Cal96`, 182 days (January 1 - June 30) fall in `Fy96` and 184 days (July 1 - December 31) fall in `Fy97`. Therefore, for the `CAL96` data, SUM BY DAY uses a weighting factor of `182/366` for `Fy96` and a factor of `184/366` for `Fy97`.

AVERAGE [BY PERIOD]
Aggregates data to a target period by averaging the data of the contributing source periods. For each target period, AVERAGE BY PERIOD adds up the data from all the source periods that end within the target period and divides this total by the number of source periods. AVERAGE BY PERIOD uses the implicit relation between the two time dimensions.

AVERAGE BY DAY
Weights the value of each contributing source period by the portion of target days it represents. For each target period, AVERAGE BY DAY multiplies the value of each source period by the number of days of that source period that actually fall within the target period. The average is then calculated by adding these weighted source values and dividing by the total number of days in the target period. When you use AVERAGE BY DAY, the value of a single source period may be apportioned across adjacent target periods.

LAST [BY PERIOD]
For each target period, LAST BY PERIOD returns the data value from the last source period that ends within the target period. It uses the implicit relation between the source and target dimensions.

LAST BY DAY
Has the same effect as LAST BY PERIOD, provided you are converting data from smaller periods to larger periods. See "Substituting a Compatible Method" on page 23-33.

STATUS
Indicates that the current status of the source dimension is used. It is the default for the SUM and AVERAGE methods.

NOSTATUS
Indicates that the full default status of the source dimension is used. It is the default for the LAST method.

SPLIT [BY PERIOD]
Allocates data to target periods by splitting the data from the source periods. SPLIT BY PERIOD divides a source value evenly among the target periods that end in that source period. SPLIT BY PERIOD uses the implicit relation between the two DAY, WEEK, MONTH, QUARTER, or YEAR dimensions.

SPLIT BY DAY
Weights each source value according to the portion of target days it represents. For each target period, SPLIT BY DAY multiplies each contributing source period value by a weighting factor that has this form where *target-days-in-source* is the Number of target-period days that actually fall in source period and *total-period-days* is the total number of days in source period:

target-days-in-source / total-period-days

SPLIT BY DAY then returns the total of these weighted source values. When you use SPLIT BY DAY, the value of an individual source period may be apportioned across adjacent target periods.

REPEAT
For each target period, REPEAT returns the value of a source period. The target periods are the periods that end within the source period. REPEAT uses the implicit relation between the source and target dimensions. REPEAT BY DAY has the same effect as REPEAT BY PERIOD, provided you are converting data from larger time

periods to smaller time periods. See "Substituting a Compatible Method" on page 23-33.

INTERPOLATE [BY PERIOD]
The INTERPOLATE method allocates data to target periods by first calculating the difference between the values of the current and previous source periods, and then splitting the result incrementally over the target periods. INTERPOLATE divides the difference between the current and previous source period values by the number of target periods that end in the source period, and it increments each target period by this amount.

INTERPOLATE BY DAY
For each target period, adds the value of the previous source period to a value that is calculated as follows where *end-days* is the number of days from end of previous source period to end of current target period and *period-days* is the total number of days in current source period:

(*end-days* / *period-days*) * (*current-source-value* - *previous-source-value*)

When a target period has days that fall in more than one source period, a similar calculation is made for each source period.

Notes

Dimensions of the Result
The results returned by TCONVERT are dimensioned by the target DAY, WEEK, MONTH, QUARTER, or YEAR dimension and by all of *expression* dimensions that are not DAY, WEEK, MONTH, QUARTER, or YEAR dimensions.

Status Used with Allocation
The STATUS and NOSTATUS keywords have no effect with the allocation methods. The allocation methods always use the full default status of the source dimension to determine the contributing periods.

Compatible Aggregation and Allocation Methods
Except for a case in which the source dimension and the target dimension have overlapping periods of equal length, you can specify both an aggregation method and an allocation method. However, the two methods must be compatible.

Table 23–1, " Compatible Aggregation and Allocation Methods" shows the compatible methods.

Table 23–1 Compatible Aggregation and Allocation Methods

Aggregation	Compatible Allocation
SUM	SPLIT
AVERAGE	REPEAT
LAST	INTERPOLATE

When you specify both an aggregation method and an allocation method, you can specify BY PERIOD or BY DAY with either method. When you specify BY PERIOD (explicitly or by default) for one method and BY DAY for the other method, BY DAY takes precedence.

Using Both Aggregation and Allocation

When you specify both an aggregation method and a compatible allocation method, Oracle OLAP handles this as follows:

- When you convert data from smaller periods to larger periods, Oracle OLAP uses the aggregation method (with BY DAY, if specified for either method).
- When you convert data from larger periods to smaller periods, Oracle OLAP uses the allocation method (with BY DAY, if specified for either method).
- When you convert data between dimensions that have non-overlapping periods of equal length, such as a quarter ending in March and a quarter ending in June, the results of the aggregation and allocation methods will be identical.

Overlapping Periods of Equal Length

When you convert data between two dimensions of type DAY, WEEK, MONTH, QUARTER, or YEAR that have overlapping periods of equal length, such as a calendar year and a fiscal year, or a quarter ending in March and a quarter ending in April, you must specify either an aggregation method or allocation method, but not both. For these dimensions, the compatible aggregation and allocation methods may yield different results.

For example, when you convert data from a calendar year dimension to a fiscal year dimension that ends in June, the SUM and SPLIT methods will return different results:

- The SUM method totals up the data from the source periods that end in the target period. Since the calendar year 1996 ends in fiscal year 1997, the SUM method assigns the value for calendar year 1996 to fiscal year 1997.
- The SPLIT method allocates a source data value to the target periods that end in the source period. Since the fiscal year 1996 ends in calendar year 1996, the SPLIT method assigns the value for calendar year 1996 to fiscal year 1996.

Substituting a Compatible Method

When you specify a single conversion method, and you use an aggregation method to convert data from a larger period to a smaller period (for example, from months to weeks) Oracle OLAP automatically uses the compatible allocation method in place of the specified aggregation method. Similarly, when you use an allocation method to convert data from a smaller period to a larger period, Oracle OLAP automatically uses the compatible aggregation method. See "Compatible Aggregation and Allocation Methods" on page 23-31.

Data Type of the Result

When possible, TCONVERT returns results that have the same data type as *expression.* When *expression* is DECIMAL, the results are always DECIMAL. When *expression* is INTEGER, the results are INTEGER when the required calculations do not involve division. For example, when two dimensions are *aligned* (that is, they have the same phase and are based on the same periods, such as a calendar year dimension and a quarter dimension ending in December), the result is INTEGER when you use the REPEAT method to convert an INTEGER *expression* from larger periods to smaller periods. Similarly, the result is INTEGER when you use the SUM or LAST method to convert the *expression* from smaller to larger periods.

Converting Text Data

You can also use TCONVERT to convert the values of a text expression when no numeric calculations are needed for the conversion. For aligned dimensions, for example, you can use the LAST method to convert text values from smaller periods to larger periods, and you can use the REPEAT method to convert text values from larger periods to smaller periods. You can also use the LAST and REPEAT methods to convert text data between dimensions that have periods of equal length. When you attempt to convert a text expression with a method that requires numeric calculations, you will receive an error message.

Methods for Financial Data

When you work with financial data, you can use an appropriate conversion method for each type of data. Table 23–2, " Examples of Conversion Methods for Different Types of Financial Data" gives some examples.

Table 23–2 Examples of Conversion Methods for Different Types of Financial Data

Type of Financial Data	Conversion	Conversion Method
Revenue or expenses	Month to year	SUM
Stock quotations	Day to quarter	AVERAGE
Balance sheet items	Month to quarter	LAST
Quarterly tax payment	Year to quarter	SPLIT BY PERIOD
Money supply	Year to quarter	INTERPOLATE

The Effect of NASKIP

TCONVERT is affected by the NASKIP option. When NASKIP is set to `NO`, TCONVERT returns an `NA` value for any target period that receives contributions from a source period with an `NA` value.

Examples

Example 23–11 Splitting Data Across Quarters

This example shows the effects of using the SPLIT method and the SPLIT BY DAY method to allocate an annual budget revenue figure of $120,000 across the quarters of the year 1996. An existing `year` dimension is the source dimension and an existing `quarter` dimension is the target dimension.

The following statements

```
DEFINE budget.revenue DECIMAL <year>
budget.revenue(year 'Yr96') = 120000
LIMIT quarter TO year 'Yr96'
REPORT W 12 HEADING 'Split Evenly' -
   TCONVERT(budget.revenue quarter SPLIT) -
   W 12 HEADING 'Split by Day' -
   TCONVERT(budget.revenue quarter Split by day)
```

produce this report.

```
QUARTER        Split Evenly Split by Day
-------------- ------------ ------------
Q1.96             30,000.00    29,836.07
Q2.96             30,000.00    29,836.07
Q3.96             30,000.00    30,163.93
Q4.96             30,000.00    30,163.93
```

Example 23–12 Aggregating Weekly Data to Monthly Using TCONVERT

This example aggregates weekly data to monthly data. First, define a week dimension named `week` and add weeks that include the dates January 1, 1996 and June 30, 1996 (Oracle OLAP automatically adds the intervening weeks).

```
DEFINE week DIMENSION WEEK
MAINTAIN week ADD '01Jan96' '30Jun96'
```

Next, define a variable named `weekvar`, dimensioned by `week`, and assign a value of `7` to each week.

```
DEFINE weekvar DECIMAL <week>
weekvar = 7
```

The following statements show that December 31, 1995 is the beginning date of the first week for which `weekvar` contains non-`NA` data and that July 6, 1996 is the ending date of the final week for which `weekvar` contains non-`NA` data.

```
SHOW BEGINDATE(weekvar)
SHOW ENDDATE(weekvar)
```

The statements produce this output.

```
31Dec95
06Jul96
```

With these beginning and ending dates, when the data is converted to monthly data, it will be aggregated over the months `Dec95` through `Jul96`. The following statements show the effects of using the SUM method and the SUM BY DAY method to convert the weekly `weekvar` data to monthly data.

```
LIMIT month TO 'Jan96' TO 'Jul96'
REPORT HEADING 'Sum' TCONVERT(weekvar month SUM) -
   HEADING 'Sum by Day' -
   TCONVERT(weekvar month SUM BY day)
```

These statements produce the following report.

```
MONTH                 Sum    Sum by Day
-------------- ---------- ----------
Jan96               28.00      31.00
Feb96               28.00      29.00
Mar96               35.00      31.00
Apr96               28.00      30.00
May96               28.00      31.00
Jun96               35.00      30.00
Jul96                7.00       6.00
```

TEMPSTAT

The TEMPSTAT command limits the dimension you are looping over, inside a FOR loop or inside a loop that is generated by the REPORTcommand. Status is restored after the statement following TEMPSTAT. When a DO ... DOEND phrase follows TEMPSTAT, status is restored when the matched DOEND or a BREAK or GOTO command is encountered. You can use TEMPSTAT only within programs.

Syntax

TEMPSTAT *dimension...*

  statement block

Arguments

***dimension*(s)**
One or more dimensions whose status you would like to temporarily change inside a FOR loop or an automatic loop that is generated by the REPORTstatement.

statement block
One or more statements that change the status of the dimension. To execute more than one statement under the temporary status, enclose them between DO ... DOEND brackets.

Notes

Nesting

You can nest TEMPSTAT commands, one within another, and you can repeat the same dimension within the nested TEMPSTAT commands.

Placement of TEMPSTAT

When you want to be able to change the status of a dimension while REPORT is looping over it, you must place the TEMPSTAT command inside that REPORT loop rather than before the REPORT command. For example, suppose you have written a user-defined function called `monthly_sales`, which changes the status of `month`, and `monthly_sales` is part of a REPORT command that is looping over `month`. In this case the TEMPSTAT command must be inside the `monthly_sales` function in order for a status change to take place. This is true even when the REPORT command is given within TEMPSTAT DO/DOEND brackets within a FOR loop over MONTH.

POP and POPLEVEL Commands

Within the DO/DOEND brackets of a TEMPSTAT statement block, you cannot use the POP command to pop a dimension that is protected by TEMPSTAT on the block -- unless the matching PUSH command is also within the block.

Similarly, within the DO/DOEND brackets of a TEMPSTAT statement block, you cannot use the POPLEVEL command to pop a dimension that is protected by TEMPSTAT on the block -- unless one of two conditions is met: the matching PUSHLEVEL command is also within the block, or the only pushes on the dimension since the PUSHLEVEL command was given are also within the block.

Use Only LIMIT and CONTEXT Commands

Within the DO/DOEND brackets of a TEMPSTAT command, the only way to change the status of a dimension within a loop over that dimension is with the LIMIT or CONTEXT APPLY commands. (See LIMIT command and CONTEXT command for details.) You cannot change the status of the dimension using POP or POPLEVEL. You also cannot perform any operations that would add values to the dimension, because adding values also changes the status of the dimension to ALL. For example, MAINTAIN ADD, FILEREAD APPEND, and IMPORT (with new values in the EIF file) add values to a dimension.

Examples

TEMPSTAT in a FOR Loop

The following program excerpt uses the TEMPSTAT command to limit the `market` dimension within the `FOR market` loop.

```
FOR market
DO
 TEMPSTAT market
  DO
   LIMIT market TO CHILDREN USING market.market
   REPORT market
  DOEND
DOEND
```

TEXTFILL

The TEXTFILL function reformats a text value to fit compactly into lines of a specified width, regardless of its current format. TEXTFILL is commonly used to reformat text with an unnecessarily ragged right margin or with a bad line width.

Return Value

TEXT or NTEXT

Syntax

TEXTFILL(*text-expression*, *width*)

Arguments

text-expression

A text expression to be reformatted to the specified width, regardless of the current format of the data. When you specify a TEXT expression, the return value is TEXT. When you specify an NTEXT expression, the return value is NTEXT.

width

The desired width of the reformatted data, entered as an integer value from `1` to `132`.

Notes

How TEXTFILL Works

TEXTFILL joins lines of text while reformatting, whereas ROW and REPORT reformat without joining lines. See Example 23–13, "The Effects of TEXTFILL on ROW" on page 23-40.

Width Greater Than the Column Width

In a structured report, TEXTFILL reformats *text-expression* to the width you specify, as long as that width is less than the width of the report column. When *width* is greater than the column width, it is ignored by TEXTFILL, and the expression is reformatted to the width of the column.

How Words are Handled

TEXTFILL fits as many words of *text-expression* as it can onto one line, placing just one space between words and removing extra spaces between words. When a word is longer than *width,* TEXTFILL breaks it across two or more lines. In this case there may be extra spaces at the end of lines.

Permanent Reformatting

Rather than repeatedly reformatting a specific text variable, you can permanently format it by assigning the result of the TEXTFILL function to the same text variable, as shown in the following example.

```
textvar = TEXTFILL(textvar 12)
```

Examples

Example 23–13 The Effects of TEXTFILL on ROW

The following example shows the effect of TEXTFILL on a ROW command, using the nicely formatted text variable `textvar`.

The statement

```
SHOW textvar
```

produces the following output.

```
You can use the following options to control the format of
   your display.

   BMARGIN    Controls the bottom margin.
   COLWIDTH   Controls column width.
   COMMAS     Controls the use of commas in numbers.
   DECIMALS   Controls number of decimal places in numbers.
   LSIZE      Controls the maximum length of a line.
   NASPELL    Controls the spelling of NA values in output.
```

The ROW command

```
ROW W 50 textvar
```

produces the following output.

```
You can use the following options to control the
format of your
display.
BMARGIN       Controls the bottom margin.
COLWIDTH      Controls column width.
COMMAS        Controls the use of commas in
numbers.
DECIMALS      Controls the number of decimal
places in numbers.
LSIZE         Controls the maximum length of a
line.
NASPELL       Controls the spelling of NA values
in output.
```

By contrast, the ROW command with TEXTFILL

```
ROW W 50 TEXTFILL(textvar, 50)
```

produces the following output.

```
You can use the following options to control the
format of your display. BMARGIN Controls the
bottom margin. COLWIDTH Controls column width.
COMMAS Controls the use of commas in numbers.
DECIMALS Controls the number of decimal places in
numbers. LSIZE Controls the maximum length of a
line. NASPELL Controls the spelling of NA values
in output.
```

THIS_AW

(Read-only)The THIS_AW option is the value of the workspace name that Oracle OLAP uses when it replaces occurrences of the THIS_AW keyword to create a qualified object name.

Data type

TEXT

Syntax

```
THIS_AW
```

THOUSANDSCHAR

(Read-only) The THOUSANDSCHAR option is the value specified for the NLS_NUMERIC_CHARACTERS option discussed in NLS Options on page 18-54.

Data type

ID

Syntax

THOUSANDSCHAR

Notes

Format for Numeric Input

The value of THOUSANDSCHAR only affects the way Oracle OLAP formats numbers in output. It does not affect the way numbers should be formatted for input.

The Decimal Marker

The DECIMALCHAR option lets you check the value of the decimal marker.

Examples

Example 23–14 Displaying the Decimal and Thousands Markers

The following statements show the DECIMALCHAR and THOUSANDSCHAR values. Assume that you issue the following statements.

```
SHOW THOUSANDSCHAR
SHOW DECIMALCHAR
```

Assume that a comma is displayed as the marker for THOUSANDSCHAR and that a period is displayed as the marker for DECIMALCHAR. With these values, the following SHOW statement would produce the output shown below it.

```
SHOW TOTAL(sales)
63,181,743.50
```

TMARGIN

The TMARGIN option defines the number of blank lines for the top margin of output pages, above the running page heading. TMARGIN is meaningful only when PAGING is set to `YES` and only for output from statements such as REPORT and DESCRIBE. The TMARGIN option is usually set in the initialization section of report programs.

Data type

INTEGER

Syntax

TMARGIN = *n*

Arguments

n

An integer expression that specifies the number of lines that you want to set aside for the top margin in a report. The default is `2`.

Notes

Producing the Top Margin Lines

The top margin lines are produced before the program that is defined by PAGEPRG, if any, is run.

Output to the Default Outfile

When you set TMARGIN for the default outfile, the new value remains in effect until you reset it, regardless of intervening OUTFILE commands that send output to a file. That is, the value of TMARGIN is automatically saved for the default outfile.

Output to a File

To set TMARGIN for a file, first make the file your current outfile by specifying its name in an OUTFILE command, then set TMARGIN to the desired value. The new value remains in effect until you reset it or until you use an OUTFILE command to direct output to a different outfile. When you direct output to a different outfile, TMARGIN returns to its default value of 2 for the file.

Examples

Example 23–15 Setting the Top Margin of a Report

In this example, you want to save space when you produce a long report, so you set a small top margin of 1 line. Here is the statement that you would include in the initialization section of your report program.

```
TMARGIN = 1
```

TO_CHAR

The TO_CHAR function converts a date, number, or NTEXT expression to a TEXT expression in a specified format. This function is typically used to format output data.

Return Value

TEXT

Syntax

TO_CHAR(*datetime-exp*, [*datetime-fmt*,] [*option setting*])

or

TO_CHAR(*num-exp*, [*num-fmt*,] [*nlsparams*])

or

TO_CHAR(*ntext-exp)*

Arguments

datetime-exp
A DATETIME expression to be converted to TEXT.

datetime-fmt
A text expression that identifies a date format model. This model specifies how the conversion from a DATETIME data type to TEXT should be performed. For information about date format models, see the *Oracle Database SQL Reference* and the *Oracle Database Globalization Support Guide*. The default value of *datetime-fmt* is controlled by the NLS_DATE_FORMAT option.

option setting
An OLAP option (such as NLS_DATE_LANGUAGE) and its new setting, which temporarily overrides the setting currently in effect for the session. Typically, this option identifies the language that you want *datetime-exp* to be translated into. See Example 23–18, "Displaying the Current Date and Time in Spanish" on page 23-49. Do not use options that set other options. See "Specifying Options" on page 23-52.

num-exp
A numeric expression to be converted to TEXT.

num-fmt
A text expression that identifies a number format model. This model specifies how the conversion from a numerical data type (NUMBER, INTEGER, SHORTINTEGER, LONGINTEGER, DECIMAL, SHORTDECIMAL) to TEXT should be performed. For information about number format models, see the *Oracle Database SQL Reference* and the *Oracle Database Globalization Support Guide*.

The default number format model uses the decimal and thousands group markers identified by NLS_NUMERIC_CHARACTERS.

nlsparams
A text expression that specifies the thousands group marker, decimal marker, and currency symbols used in *num-exp*. This expression contains one or more of the following parameters, separated by commas:

NLS_CURRENCY *symbol*

NLS_ISO_CURRENCY *territory*

NLS_NUMERIC_CHARACTERS *dg*

symbol
A text expression that specifies the local currency symbol. It can be no more than 10 characters.

territory
A text expression that identifies the territory whose ISO currency symbol is used.

dg
A text expression composed of two different, single-byte characters for the decimal marker (`d`) and thousands group marker (`g`).

These parameters override the default values specified by the NLS_CURRENCY, NLS_ISO_CURRENCY, and NLS_NUMERIC_CHARACTERS options. (See NLS Options on page 18-54.)

ntext-exp
An NTEXT expression to be converted to TEXT. A conversion from NTEXT to TEXT can result in data loss when the NTEXT value cannot be represented in the database character set.

Notes

Similarity to SQL TO_CHAR Function

The OLAP DML TO_CHAR function has the same functionality as the SQL `TO_CHAR` function. For more information about the SQL `TO_CHAR` function, see *Oracle Database SQL Reference*.

Support for Numerical Data Types

The TO_CHAR function converts INTEGER, SHORTINTEGER, LONGINTEGER, DECIMAL, and SHORTDECIMAL values to NUMBER before converting them to TEXT. Thus, TO_CHAR converts NUMBER values faster than other numerical data types.

Output Date Format

A converted date has the format specified by the NLS_DATE_FORMAT option.

Rounding

All number format models cause the number to be rounded to the specified number of significant digits. Table 23–3, " Possible Effects of Rounding" identifies some of the effects of rounding.

Table 23–3 Possible Effects of Rounding

IF *num-exp*	THEN the return value
has more significant digits to the left of the decimal place than are specified in the format,	appears as pound signs (#).
is a very large positive value that cannot be represented in the specified format,	is a tilde (~).
is a very small negative value that cannot be represented in the specified format,	is a negative sign followed by a tilde (-~).

Specifying Options

Options that set other options should not be used in this statement. For example, do not set NLS_DATE_LANGUAGE or NLS_TERRITORY. Set NLS_DATE_LANGUAGE instead.

While TO_CHAR will save and restore the current setting of the specified option so that it has a new value only for the duration of the statement, TO_CHAR cannot save and restore any side effects of changing that option. For example, NLS_TERRITORY controls the value of NLS_DATE_FORMAT, NLS_NUMERIC_CHARACTERS, NLS_CURRENCY, NLS_CALENDAR, and other options. (See NLS Options on page 18-54.) When you change the value of NLS_TERRITORY in a call to TO_CHAR, all of these options will be reset to their territory-appropriate default values twice: once when NLS_TERRITORY is set to its new value for the duration of the TO_CHAR command, and again when the saved value of NLS_TERRITORY is restored.

Simple Data Type Conversion

For simple data type conversion, use CONVERT.

Examples

Example 23–16 Converting a Date to CHAR

This statement converts today's date and specifies the format.

```
SHOW TO_CHAR(SYSDATE, 'Month DD, YYYY HH24:MI:SS')
```

The specified date format allows the time to be displayed along with the date.

```
November  30, 2000 10:01:29
```

Example 23–17 Converting a Numerical Value to Text

This statement converts a number to text and specifies a space as the decimal marker and a period as the thousands group marker.

```
SHOW TO_CHAR(1013.50, NA, NLS_NUMERIC_CHARACTERS ' .')
```

The value 1013.50 now appears like this:

```
1.013 50
```

Example 23–18 Displaying the Current Date and Time in Spanish

The following statements set the default language to Spanish and specify a new date format.

```
NLS_DATE_LANGUAGE = 'spanish'
NLS_DATE_FORMAT = 'Day: Month dd, yyyy HH:MI:SS am'
```

The following statement:

```
SHOW TO_CHAR(SYSDATE)
```

displays the current date and time in Spanish:

```
Viernes  : Diciembre  01, 2000 08:21:17 AM
```

The NLS_DATE_LANGUAGE option changes the language for the duration of the statement. The following statement:

```
SHOW TO_CHAR(SYSDATE, NA, NLS_DATE_LANGUAGE 'german')
```

Displays the date and time in German:

```
Freitag   : Dezember  01, 2000 08:26:00 AM
```

TO_DATE

The TO_DATE function converts a formatted TEXT or NTEXT expression to a DATETIME value. This function is typically used to convert the formatted date output of one application (which includes information such as month, day, and year in any order and any language, and separators such as slashes, dashes, or spaces) so that it can be used as input to another application.

Return Value

DATETIME

Syntax

TO_DATE(*text-exp*, [*fmt*,] [*option setting*])

Arguments

text-exp
The text expression that contains a date to be converted. The expression can have the TEXT or NTEXT data type. A conversion from NTEXT can result in an incorrect result when the NTEXT value cannot be interpreted as a date.

fmt
A text expression that identifies a date format model. This model specifies how the conversion from text to DATE should be performed. For information about date format models, see the *Oracle Database SQL Reference* and the *Oracle Database Globalization Support Guide*.

The default value of *fmt* is the value of NLS_DATE_FORMAT.

option setting
An OLAP option (such as NLS_DATE_LANGUAGE) and its new setting, which temporarily overrides the setting currently in effect for the session. Typically, this option identifies the language of *text-exp* when it is different from the session language. See Example 23–20, "Specifying a Default Language and a Date Format" on page 23-53. Do not use options that set other options. See "Specifying Options" on page 23-52.

Notes

Similarity to SQL TO_DATE Function

The OLAP DML TO_DATE function has the same functionality as the SQL `TO_DATE` function. For more information about the SQL `TO_DATE` function, see *Oracle Database SQL Reference*.

Capitalization

Capital letters in words, abbreviation, or Roman numerals in a format element produce corresponding capitalization in the return value. For example, the format element `DAY` produces `MONDAY`, `Day` produces `Monday`, and `day` produces `monday`.

Output Format

The date value generated by TO_DATE has the format specified by the NLS_DATE_FORMAT option.

Default Date Format Values

The values of some format elements are determined by the value of the NLS_TERRITORY option. The language used for months and days is controlled by NLS_DATE_LANGUAGE.

Specifying Options

Options that set other options should not be used in this statement. For example, do not set NLS_LANGUAGE or NLS_TERRITORY. Set NLS_DATE_LANGUAGE instead. (See NLS Options on page 18-54 for more information on these options.)

While TO_DATE will save and restore the current setting of the specified option so that it has a new value only for the duration of the statement, TO_DATE cannot save and restore any side effects of changing that option. For example, NLS_TERRITORY controls the value of NLS_DATE_FORMAT, NLS_NUMERIC_CHARACTERS, NLS_CURRENCY, NLS_CALENDAR , and other options. When you change the value of NLS_TERRITORY in a call to TO_DATE, all of these options will be reset to their territory-appropriate default values twice: once when NLS_TERRITORY is set to its new value for the duration of the TO_DATE command, and again when the saved value of NLS_TERRITORY is restored.

Unrecognized Dates

When TO_DATE cannot construct a value with a valid DATE value using *fmt*, it returns an error. For example, when an alphanumeric character appears in *text-exp* where *fmt* indicates a punctuation character, then an error results.

Simple Data Type Conversion

To convert dates with minimal formatting requirements, use CONVERT.

Examples

Example 23–19 Converting Text Values to DATE Values

The following statement converts `January 15, 2002, 11:00 A.M.` to the default date format of `15JAN02`, and stores that value in a DATE variable named `bonusdate`.

```
bonusdate = TO_DATE('January 15, 2002, 11:00 A.M.', -
   'Month dd, YYYY, HH:MI A.M.')
```

Example 23–20 Specifying a Default Language and a Date Format

The following statements set the default language to Spanish and specify a new date format. The NLS_DATE_LANGUAGE option, when used in the TO_DATE function, allows the American month name to be translated.

```
NLS_DATE_FORMAT = 'Day: Month dd, yyyy HH:MI:SS am'
NLS_DATE_LANGUAGE = 'spanish'
SHOW TO_DATE('November 15, 2001', 'Month dd, yyyy', -
   NLS_DATE_LANGUAGE 'american')
```

The date is translated from American to Spanish and displayed in the new date format.

```
Jueves   : Noviembre  15, 2001 12:00:00 AM
```

TO_NCHAR

The TO_NCHAR function converts a TEXT expression, date, or number to NTEXT in a specified format. This function is typically used to format output data.

Return Value

NTEXT

Syntax

TO_NCHAR(*text-exp)*

or

TO_NCHAR(*datetime-exp*, [*datetime-fmt*,] [*option setting*]

or

TO_NCHAR(*num-exp*, [*num-fmt*,] [*nlsparams*]

Arguments

text-exp
A TEXT expression to be converted to NTEXT.

datetime-exp
A DATETIME expression to be converted to NTEXT.

datetime-fmt
A text expression that identifies a date format model. This model specifies how the conversion from a DATETIME data type to NTEXT should be performed. For information about date format models, see the *Oracle Database SQL Reference* and the *Oracle Database Globalization Support Guide*. The default value of *datetime-fmt* is controlled by the NLS_DATE_FORMAT option.

option setting
An OLAP option (such as NLS_DATE_LANGUAGE) and its new setting, which temporarily overrides the setting currently in effect for the session. Typically, this option identifies the language that you want *datetime-exp* to be translated into. See Example 23–23, "Specifying the Default Language and a Date Format" on

page 23-57. Do not use options that set other options. See "Specifying Options" on page 23-56.

num-exp
A numeric expression to be converted to NTEXT.

num-fmt
A text expression that identifies a number format model. This model specifies how the conversion from a numerical data type (NUMBER, INTEGER, SHORTINTEGER, LONGINTEGER, DECIMAL, SHORTDECIMAL) to TEXT should be performed. For information about number format models, see the *Oracle Database SQL Reference* and the *Oracle Database Globalization Support Guide*.

The default number format model uses the decimal and thousands group markers identified by NLS_NUMERIC_CHARACTERS option.

nlsparams
A text expression that specifies the thousands group marker, decimal marker, and currency symbols used in *num-exp*. This expression contains one or more of the following parameters, separated by commas:

NLS_CURRENCY *symbol*

NLS_ISO_CURRENCY *territory*

NLS_NUMERIC_CHARACTERS *dg*

symbol
A text expression that specifies the local currency symbol. It can be no more than 10 characters.

territory
A text expression that identifies the territory whose ISO currency symbol is used.

dg
A text expression composed of two different, single-byte characters for the decimal marker (`d`) and thousands group marker (`g`).

These parameters override the default values specified by the NLS_CURRENCY, NLS_ISO_CURRENCY, and NLS_NUMERIC_CHARACTERS options. (See NLS Options on page 18-54.)

Notes

NTEXT Always UTF8 Unicode

The return value of the TO_NCHAR function has the NTEXT data type, which is always in UTF8 Unicode. This encoding might be different from the NCHAR character set of the database, which can be UTF16.

Similarity to SQL TO_NCHAR Function

The OLAP DML TO_NCHAR function has functionality similar to that of the SQL `TO_NCHAR` function. For more information about the SQL `TO_NCHAR` function, see *Oracle Database SQL Reference*.

Support for Numerical Data Types

The TO_NCHAR function converts INTEGER, SHORTINTEGER, LONGINTEGER, DECIMAL, and SHORTDECIMAL values to NUMBER before converting them to NTEXT. Thus, TO_NCHAR converts NUMBER values faster than other numerical data types.

Output Date Format

A converted date has the format specified by the NLS_DATE_FORMAT option. (See NLS Options on page 18-54.)

Rounding

All number format models cause the number to be rounded to the specified number of significant digits. Table 23–3, " Possible Effects of Rounding" on page 23-48 identifies some of the effects of rounding.

Specifying Options

Options that set other options should not be used in this command. For example, do not set NLS_LANGUAGE or NLS_TERRITORY. Set NLS_DATE_LANGUAGE instead.

While TO_NCHAR will save and restore the current setting of the specified option so that it has a new value only for the duration of the command, TO_NCHAR cannot save and restore any side effects of changing that option. For example, NLS_TERRITORY controls the value of NLS_DATE_FORMATE, NLS_NUMERIC_CHARACTERS, NLS_CURRENCY, NLS_CALENDAR, and other options. When you change the value of NLS_TERRITORY in a call to TO_NCHAR, all of these options will be reset to their territory-appropriate default values twice: once when NLS_TERRITORY is set to its new value for the duration of the

TO_NCHAR command, and again when the saved value of NLS_TERRITORY is restored.

Simple Data Type Conversion

For simple data type conversion, use CONVERT.

Examples

Example 23–21 Date Conversion

This statement converts today's date and specifies the format.

```
SHOW TO_NCHAR(SYSDATE, 'Month DD, YYYY HH24:MI:SS')
```

The specified date format allows the time to be displayed along with the date.

```
November  30, 2000 10:01:29
```

Example 23–22 Converting Numerical Data to NTEXT Data

This statement converts a number to NTEXT and specifies a space as the decimal marker and a period as the thousands group marker.

```
SHOW TO_NCHAR(1013.50, NA, NLS_NUMERIC_CHARACTERS ' .')
```

The value 1013.50 now appears like this:

```
1.013 50
```

Example 23–23 Specifying the Default Language and a Date Format

The following statements set the default language to Spanish and specify a new date format.

```
NLS_DATE_LANGUAGE = 'spanish'
NLS_DATE_FORMAT = 'Day: Month dd, yyyy HH:MI:SS am'
```

The following statement:

```
SHOW TO_NCHAR(SYSDATE)
```

Displays the current date and time in Spanish:

```
Viernes   : Diciembre  01, 2000 08:21:17 AM
```

The NLS_DATE_LANGUAGE option changes the language for the duration of the statement. The following statement

```
SHOW TO_NCHAR(SYSDATE, NA, NLS_DATE_LANGUAGE 'german')
```

displays the date and time in German:

```
Freitag   : Dezember  01, 2000 08:26:00 AM
```

TO_NUMBER

The TO_NUMBER function converts a formatted TEXT or NTEXT expression to a number. This function is typically used to convert the formatted numerical output of one application (which includes currency symbols, decimal markers, thousands group markers, and so forth) so that it can be used as input to another application.

Return Value

NUMBER. Negative return values contain a leading negative sign, and positive values contain a leading space, unless the format model contains the `MI`, `S`, or `PR` format elements.

Syntax

TO_NUMBER(*text-exp*, [*fmt*,] [*nlsparams*])

Arguments

text-exp
A text expression that contains a number to be converted. The expression can have the TEXT or NTEXT data type. A conversion from NTEXT can result in an incorrect result when the NTEXT value cannot be interpreted as a number.

fmt
A text expression that identifies a number format model. This model specifies how the conversion to NUMBER should be performed. For information about number format models, see the *Oracle Database SQL Reference* and the *Oracle Database Globalization Support Guide*.

The default number format identifies a period (`.`) as the decimal marker and does not recognize any other symbol.

nlsparams

A text expression that specifies the thousands group marker, decimal marker, and currency symbols used in *text-exp*. This expression contains one or more of the following parameters, separated by commas:

NLS_CURRENCY *symbol*

NLS_ISO_CURRENCY *territory*

NLS_NUMERIC_CHARACTERS *dg*

symbol

A text expression that specifies the local currency symbol. It can be no more than 10 characters.

territory

A text expression that identifies the territory whose ISO currency symbol is used.

dg

A text expression composed of two different, single-byte characters for the decimal marker (d) and thousands group marker (g).

These parameters override the default values specified by th NLS_CURRENCY, NLS_ISO_CURRENCY, and NLS_NUMERIC_CHARACTERS options. Refer to NLS Options on page 18-54 for additional information.

Notes

Similarity to SQL TO_NUMBER Function

The OLAP DML TO_NUMBER function has the same functionality as the SQL TO_NUMBER function. For more information about the SQL TO_NUMBER function, see *Oracle Database SQL Reference*.

Default Number Format Values

The values of some formats are determined by the value of NLS_TERRITORY. (See NLS Options on page 18-54.) .

Rounding

All number format models cause the number to be rounded to the specified number of significant digits. Table 23–3, " Possible Effects of Rounding" on page 23-48 identifies some of the effects of rounding.

Simple Data Type Conversion

To convert text with minimal formatting requirements, use CONVERT.

Examples

Example 23–24 Converting Text Data to Decimal Data

The following statements convert a text string to a DECIMAL data type by identifying the local currency symbol (`L`), the thousands group separator (`G`) and the decimal marker (`D`). The NLS_NUMERIC_CHARACTERS option identifies the characters used for the `G` and `D` format, since they are different from the current setting for the session.

```
DEFINE money VARIABLE DECIMAL
money = TO_NUMBER('$94 567,00', 'L999G999D00', NLS_NUMERIC_CHARACTERS ', ')
SHOW money
```

The output of this statement is:

```
94,567.00
```

TOD

The TOD function returns the current time of day in the form hh:mm:ss using a 24-hour format.

Return Value

ID

Syntax

TOD

Examples

Example 23–25 Displaying the Current Time

The following statement sends the current time of day to the current outfile.

```
show tod
```

This statement produces the following output.

```
17:30:46
```

TODAY

The TODAY function returns the current date as a DATE value.

Return Value

DATE

Syntax

TODAY

Notes

Format of the Date

When you display the result returned by TODAY, the value has the format specified by the date template in the DATEFORMAT option. When the day of the week or the name of the month is used in the date template, TODAY uses the day names specified in the DAYNAMES option and the month names specified in the MONTHNAMES option. You can use the result returned by TODAY anywhere that a DATE value is expected.

DATE-to-TEXT Conversion

You can also use the result where a text value is expected. TODAY automatically converts the date to a text value, using the current template in the DATEFORMAT option to format the text value. When you want to override the current DATEFORMAT template, you can convert the date result to text by using the CONVERT function with a date-format argument.

Examples

Example 23–26   Displaying Today's Date

The following statements send the current date in DATE format to the current outfile.

```
DATEFORMAT = '<wtextl> <mtextl> <d>, <yyyy>'
SHOW TODAY
```

When the current date is January 15, 1996, then these statements produce the following output.

```
Monday January 15, 1996
```

Example 23–27 Calculating a Date Using the TODAY Function

The following statement calculates the date 60 days from today.

```
SHOW TODAY + 60
```

When the current date is January 15, 1996, then this statement produces the following output.

```
Friday March 15, 1996
```

TOTAL

The TOTAL function calculates the total of the values of an expression.

Return Value

The data type of the expression. It can be INTEGER, LONGINT, or DECIMAL.

Syntax

TOTAL(*expression* [[STATUS] *dimensions*])

Arguments

expression
The expression to be totalled.

STATUS
Can be specified when one or more of the dimensions of the result of the function are not dimensions of the expression. (See the description of the *dimensions* argument.) When you specify the STATUS keyword when this is not the case, Oracle OLAP produces an error.

When one or more of the dimensions of the result of the function are not dimensions of the expression, Oracle OLAP creates a temporary variable to use while processing the function. When you specify the STATUS keyword, Oracle OLAP uses the current status instead of the default status of the related dimensions for calculating the size of this temporary variable. In this situation, the STATUS keyword might be *required* in order for Oracle OLAP to process the function successfully, or the STATUS keyword might provide a performance enhancement:

- When the size of the temporary variable for the results of the function would exceed the maximum size for an Oracle OLAP variable, you *must* specify the STATUS keyword in order for Oracle OLAP to execute the function successfully.
- When the dimensions of the expression are limited to a few values and are physically fragmented, you can specify the STATUS keyword to improve the performance of the function.

When you use TOTAL with the STATUS keyword for an expression that requires going outside of the status for results (for example, with the LEAD or LAG functions or with a qualified data reference), the results outside of the status will be returned as NA.

dimensions

The dimensions of the result. By default, TOTAL returns a single value. When you indicate one or more dimensions for the results, TOTAL calculates a total for each value of the dimensions that are specified and returns an array of values. Each dimension must be either a dimension of *expression* or related to one of its dimensions. When it is a related dimension, you can specify the name of the relation instead of the dimension name. This enables you to choose which relation is used when there is more than one.

Notes

NA Values

TOTAL is affected by the NASKIP option. When NASKIP is set to YES (the default), TOTAL ignores NA values and returns the sum of the values that are not NA. When NASKIP is set to NO, TOTAL returns NA when any value in the calculation is NA. When all data values for a calculation are NA, TOTAL returns NA for either setting of NASKIP.

Aggregating to Higher Levels

When you specify related *dimensions,* TOTAL adds the values of an array across one or more of its dimensions to obtain an array with fewer dimensions. Because of this, TOTAL is useful for aggregating data from a lower level of detail to a higher level.

Totaling over a Time Dimension

When *expression* is dimensioned by a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR, you can specify any other DAY, WEEK, MONTH, QUARTER, or YEAR dimension as a related *dimension.* Oracle OLAP uses the implicit relation between the dimensions. To control the mapping of one DAY, WEEK, MONTH, QUARTER, or YEAR dimension to another (for example, from weeks to months), you can define an explicit relation between the two dimensions and specify the name of the relation as the *dimension* argument to the TOTAL function.

For each time period in the related dimension, Oracle OLAP totals the data for all the source time periods that end in the target time period. This method is used regardless of which dimension has the more aggregate time periods. To control the way in which data is aggregated or allocated between the periods of two time dimensions, you can use the TCONVERT function.

Multiple Relations in a TOTAL Function

When you break out the total by a related dimension, you are changing the dimensionality of the expression, so Oracle OLAP expects values based on this new dimensionality. It chooses the relation that holds values of that dimension.

When there is more than one relation that holds values of the expected dimension, Oracle OLAP uses the one that was defined first. When there is no relation in which the related dimension is the one expected, Oracle OLAP looks for a relation that is dimensioned by the expected dimension.

For example, assume that there are two relations between `district` and `region`, as follows.

```
DEFINE REGION.DISTRICT RELATION REGION <DISTRICT>
LD The region each district belongs to

DEFINE DISTRICT.REGION RELATION DISTRICT <REGION>
LD The primary district in each region
```

When a workspace had the two relations described earlier and you specified the following TOTAL function, Oracle OLAP would use the relation `region.district` by default, because it holds values of the specified dimension.

```
REPORT TOTAL(sales region)
```

Examples

Example 23–28 Totaling Sales over All Months

Suppose you would like to see the total sportswear sales for all months for each district. Use the TOTAL function to calculate the total sales. To see a total for each district, specify `district` as the dimension of the results.

```
LIMIT product TO 'Sportswear'
REPORT W 15 HEADING 'Total Sales' TOTAL(sales district)
```

The preceding statements produce the following output.

```
DISTRICT         Total Sales
-------------- ---------------
Boston            1,659,609.90
Atlanta           3,628,616.62
Chicago           2,296,631.81
Dallas            3,893,829.30
Denver            2,133,425.29
Seattle           1,298,215.59
```

TRACEFILEUNIT

(Read-only) The TRACEFILEUNIT option records the unit number of the Oracle trace file. This is a writable output file that collects information about the activity in the Oracle session.

Syntax

TRACEFILEUNIT

Notes

Use of the TRACEFILEUNIT Value

The unit number stored in the TRACEFILEUNIT option can be useful because you might want to set the POUTFILEUNIT option to its value. You might also set the OUTFILE or DBGOUTFILE options to this number.

Examples

Example 23–29 Setting POUTFILEUNIT to the Oracle Trace File

The following code sets the POUTFILEUNIT option to the value of TRACEFILEUNIT option.

```
POUTFILE = TRACEFILEUNIT
```

TRACKPRG

The TRACKPRG command tracks the performance cost of every program that runs while you have tracking turned on. To get meaningful information from TRACKPRG, your session must be the only one running in Oracle OLAP. Furthermore, the accuracy of the results of TRACKPRG decreases as more processes are started on the host computer.

You turn TRACKPRG on, run the programs you want to track, and use TRACKPRG again to obtain the results. Each time each program is executed, TRACKPRG stores its cost data as one entry in its tracking list. When you execute another program, a new entry is added to the list, which is maintained in Oracle OLAP memory (free storage).

A program or line of code is considered to have a high performance cost when it takes a long time to execute. Use TRACKPRG to identify programs that have relatively high costs and then use the MONITOR command to identify the time-consuming lines within those programs. When you wish, you can use both commands simultaneously.

Syntax

TRACKPRG {ON|OFF|*file*|INIT}

where *file* is:

FILE [APPEND] [*file-id*]

Arguments

ON
Starts looking for programs to be run so it can gather their timing data in a tracking list. (Continues the current tracking process without interruption when tracking is already on, or resumes with a gap when tracking is off.)

OFF
Stops tracking programs and freezes any timing data currently in the tracking list. This lets you immediately, or later in your session, send the list to the current outfile or to a text file.

FILE
Specifies where to send the tracking list. TRACKPRG FILE has no effect on the tracking list, so you can send the same list repeatedly to different destinations.

APPEND
Specifies that Oracle OLAP adds the tracking list to the contents of the file indicated by *file-id* instead of replacing it.

file-id
Specifies the file to which Oracle OLAP sends the data. When you specify *file-id*, Oracle OLAP sends to the named text file. When you omit *file-id*, Oracle OLAP sends the timing data currently in the tracking list to the current outfile.

INIT
Discards the timing data in the current tracking list and releases the Oracle OLAP memory that was used for that list (useful when you want the memory for other purposes). Also, when tracking is on, resumes waiting for you to run programs so it can gather their data into a completely new tracking list.

Notes

Single Execution

Each entry (that is, line) in the tracking list focuses on a single execution of a single program.

Depth of the Call

Each entry records the depth of the call, if any, to the current program; that is, how many program calls it has taken to get to the program reported on the current line. In TRACKPRG output, the depth of the call is indicated by the indentation of the program name. For each indented program, TRACKPRG also records the name of the program that called it at the end of the entry.

Types of Timing Data

In each entry, TRACKPRG records two types of timing data:

- Exclusive cost -- The time spent in this program, excluding the time spent on any programs that are called by this one.
- Inclusive cost -- The time spent in this program, including the time spent on any programs that are called by this one.

This gives you the option of generating a report on both types of cost.

Entry Sections

In TRACKPRG output, each entry (line) is divided into the following four sections:

- Program name, in character columns 1 through 38
- Exclusive time, in columns 39 through 49
- Inclusive time, in columns 50 through 60
- Name of calling program, in columns 61 through 77

Here is a sample of TRACKPRG output (for the MAIN program) with column numbers included for reference.

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890

MAIN                                      39.6198425 225.551453
 COMM                                      43.793808  185.93161 MAIN
  _C.SYS.INFO                             .112533569 .112533569 COMM
  _C.SYS.INFO                             .087173462 .087173462 COMM
  _C.MAIN                                 61.414505  141.938095 COMM
   _C.CON                                 66.7147064 80.5235901 _C.MAIN
    _C.SYS.DORETURN                       .032287598 .032287598 _C.CON
```

TRACKREPORT Program

When you want to use Oracle OLAP reporting capabilities to produce a report from the timing data in the text file that is created by TRACKPRG, you can use the TRACKREPORT program. It has the following syntax.

TRACKREPORT *textfile-id*

The *textfile-id* argument is the file id of the text file created by TRACKPRG from which you want to generate a report. TRACKREPORT uses the FILEREAD command to read the data into an Oracle OLAP variable, and then it uses Oracle OLAP reporting capabilities to produce a report like the following sample.

```
                     Exclusive  Inclusive  Number of
    Program name        cost       cost      calls
____________________ __________ __________ __________

COMM                 43.793808   185.93161          1
MAIN                 39.6198425 225.551453          1
_C.CON               66.7147064 80.5235901          1
_C.ENV.PUTOPTS       1.15296936 1.15296936          1
_C.ENV.XLATEIN       6.32765198 6.32765198          1
_C.MAIN               61.414505 141.938095          1
_C.SYS.DORETURN      .032287598 .032287598          1
_C.SYS.INFO          .289932251 .289932251          3
_C.SYS.NOF10         .038269043 .038269043          1
_CONNECT              5.3609314 6.16748047          1
_CONNNONE            .806549072 .806549072          1
```

When you want to further process the data from a TRACKPRG file, you can write your own program using the `TRACKREPORT` program as a model.

Excluded Subprograms

When you do not want separate performance data on all the subprograms called by the program you are timing, you can, within the overall program, turn tracking off before calling any subprograms you want to exclude and then turn it back on before calling any you want to include. You can do this repeatedly. Remember, however, that the time taken by any excluded subprograms is assigned to the total "exclusive" time for the overall program as well as to its "inclusive" time, since TRACKPRG has not individually tracked the excluded subprograms.

Very Small Programs

You might not be able to reproduce the results exactly for very small programs. When the CPU interrupts processing to do other tasks, that time is a greater percentage of the total execution time.

Unit of Measure

The MONITOR and TRACKPRG commands use milliseconds as the unit for recording execution time. The execution time does not include time spent on I/O and time spent waiting for the next statement.

Examples

Example 23–30 Collecting Timing Data USING TRCKPRG

In this example, timing data on the `mybjt` program and all the programs it calls is collected in a file called `mybjttim.dat`.

```
TRACKPRG ON
mybjt
TRACKPRG OFF
TRACKPRG FILE mybjttim.dat
TRACKPRG INIT
TRACKREPORT mybjttim.dat
```

Example 23–31 Using the INIT Keyword and TRACKREPORT

In this example, tracking is turned on to collect timing data about the execution of `prog1` and the data is sent to a file named `prog1.trk`. Then, the INIT keyword is used to discard the existing tracking list so the data for a second program can be collected and sent to a file. Throughout the procedure, tracking remains on. Finally, after tracking is turned off and the INIT keyword is used to release the memory that was used for the tracking list, the `TRACKREPORT` program is called to produce two reports generated from the data stored in the two files.

```
TRACKPRG ON
prog1
TRACKPRG FILE prog1.trk
TRACKPRG INIT
prog2
TRACKPRG FILE prog2.trk
TRACKPRG OFF
TRACKPRG INIT
TRACKREPORT prog1.trk
TRACKREPORT prog2.trk
```

24
TRAP to ZSPELL

This chapter contains the following OLAP DML statements:

- USERID
- USETRIGGERS
- VALSPERPAGE
- VALUES
- VARCACHE
- VARIABLE
- VINTSCHED
- VNF
- VPMTSCHED
- WEEKDAYSNEWYEAR
- WEEKOF
- WHILE
- WIDTH_BUCKET
- WKSDATA
- YESSPELL
- YRABSTART
- YYOF
- ZEROROW
- ZEROTOTAL
- ZSPELL

TRAP

Within an OLAP DML program, the TRAP command causes program execution to branch to a label when an error occurs in a program or when the user interrupts the program. When execution branches to the trap label, that label is deactivated.

The label should be no longer than eight characters. It must start with a letter, dot, or underscore, and the remaining characters must be letters, numbers, dots, or underscores.

Syntax

TRAP {OFF|ON *errorlabel* [NOPRINT|PRINT]}

Arguments

OFF
Deactivates the trap label. Since only one trap label can be active at a time, you do not supply *errorlabel* when setting `TRAP OFF`. When you try to include a label with OFF, an error occurs.

ON *errorlabel*
Activates the trap label (*errorlabel*). When TRAP is active, any error in the program will cause execution to branch to *errorlabel.*

errorlabel
The name of a label elsewhere in the program constructed following the "Guidelines for Constructing a Label" on page 14-7. Execution of the program branches to the line directly following the specified label.

Note that *errorlabel*, as specified in ON, must *not* be followed by a colon. However, the actual label elsewhere in the program must end with a colon.

NOPRINT
PRINT
Indicates whether to suppress output of the error message. NOPRINT suppresses the message. PRINT (default) means that the error message is sent to the current outfile before execution branches to the trap label. With the OFF keyword, NOPRINT and PRINT are meaningless and produce an error.

Notes

Activating a Trap Label

To activate a trap label, include a TRAP command at the beginning of your program and specify a trap label in it. Then include this label later in your program.

Missing Label

When an actual trap label that corresponds to *errorlabel* does not exist elsewhere in the same program, execution stops with an error.

Automatic Deactivation

When an error occurs in a program that contains a trap label, execution branches to the label and the trap is deactivated. You do not have to execute an explicit `TRAP OFF` command. Thus, an error occurring after execution has branched to the label will not cause execution to branch to the same label again.

ERRORNAME and ERRORTEXT

In the statements that follow the trap label, you can check the name of the error that has occurred by using the ERRORNAME option, which contains the name of the first error occurring in the program. You can also check the error message for that error by using the ERRORTEXT option (see the entries for ERRORNAME and ERRORTEXT).

To find out what the value of ERRORNAME will be for specific error conditions, you can check the dimension `_MSGID`, which is supplied as a part of Oracle OLAP. The error messages are contained in the variable `_MSGTEXT`, which is dimensioned by `_MSGID`. To see this list, execute the following statement.

```
REPORT W 60 _MSGTEXT
```

Passing an Error to a Calling Program

To pass an error to a calling program, you can use one of two methods. The method you use depends on when you want the error message to be produced. With the first method, Oracle OLAP produces the message immediately and then the error condition is passed through the chain of programs. With the second method, Oracle OLAP passes the error through the chain of programs first and then produces the message. See "Passing an Error: Method One" on page 24-5 and "Passing an Error: Method Two" on page 24-5 for details.

With both methods, the appropriate error handling happens in each program in the chain, and at some point Oracle OLAP sends an error message to the current outfile.

Passing an Error: Method One

Using this method, Oracle OLAP produces the message immediately and then the error condition is passed through the chain of programs.

Use a TRAP command with the (default) PRINT option. When an error occurs, Oracle OLAP produces an error message, and execution branches to the trap label. After the trap label, perform whatever cleanup you want, and then execute the following statement.

```
SIGNAL PRGERR
```

This creates an error condition that is passed up to the program from which the current program was run. However, PRGERR does not produce an error message. PRGERR sets the ERRORNAME option to a blank value.

When the calling program contains a trap label, execution branches to the label. When each of the programs in a sequence of nested programs uses TRAP and SIGNAL in this way, you can pass the error condition up through the entire sequence of programs.

Passing an Error: Method Two

Using this method, Oracle OLAP passes the error through the chain of programs first and then produces the message.

Use a TRAP command with the NOPRINT option. When an error occurs, execution branches to the trap label, but the error message is suppressed. After the trap label, perform whatever cleanup you want, then execute the following statement.

```
SIGNAL ERRORNAME ERRORTEXT
```

The options ERRORNAME and ERRORTEXT contain the name and message of the original error, so this SIGNAL command reproduces the original error. The error is then passed up to the program from which the current program was run.

When the calling program also contains a trap label, execution branches to its label. When each of the programs in a sequence of nested programs uses `TRAP...NOPRINT` and `SIGNAL ERRORNAME ERRORTEXT` in this way, you can pass the error condition up through the entire sequence of programs. Oracle OLAP produces the error message at the end of the chain.

When you reach a level where you want to handle the error and continue the application, omit the SIGNAL command. You can display your own message with the SHOW command.

Examples

Example 24–1 Trapping a Program Error

The following program fragment uses the TRAP command to direct control to a label where options and dimension status are set back to the values they had before the program was executed and an error is signaled.

```
PUSH month DECIMALS LSIZE PAGESIZE
TRAP ON haderror NOPRINT
LIMIT month TO LAST 1
   ...
POP month DECIMALS LSIZE PAGESIZE
RETURN

haderror:
POP month DECIMALS LSIZE PAGESIZE
SIGNAL ERRORNAME ERRORTEXT
```

Example 24–2 Producing a Program Error Message Immediately

To produce the error message immediately, use a TRAP command in each nested program, but do not use the NOPRINT keyword. When an error occurs, an error message is produced immediately, and execution branches to the trap label.

At the trap label, perform whatever error-handling commands you want and restore the environment. Then execute a SIGNAL statement that includes the PRGERR keyword.

```
SIGNAL PRGERR
```

When you use the PRGERR keyword in the SIGNAL statement, no error message is produced, and the name PRGERR is not stored in ERRORNAME. The SIGNAL command signals an error condition that is passed up to the program from which the current program was run. When the calling program contains a trap label, then execution branches to that label.

When each program in a chain of nested programs uses the TRAP and SIGNAL commands in this way, you can pass the error condition up through the entire chain. Each program has commands like these.

```
TRAP ON error
   ...       "Body of program and normal exit commands
RETURN
error:
   ...       "Error-handling and exit commands
SIGNAL PRGERR
```

Example 24–3 Producing a Program Error Message at the End of the Chain

To produce the error message at the end of a chain of nested programs, use a TRAP statement that includes the NOPRINT keyword. When an error occurs in a nested program, execution branches to the trap label, but the error message is suppressed.

At the trap label, perform whatever error-handling commands you want and restore the environment. Then execute the following SIGNAL command.

```
SIGNAL ERRORNAME ERRORTEXT
```

The preceding SIGNAL statement contains includes ERRORNAME and ERRORTEXT within it. The ERRORNAME option contains the name of the original error, and the ERRORTEXT option contains the error message for the original error. When the calling program contains a trap label, then execution branches to that label.Consequently, the SIGNAL statement passes the original error name and error text to the calling program.

When each program in a chain of nested programs uses the TRAP and SIGNAL commands in this way, the original error message is produced at the end of the chain. Each program has commands like the following.

```
TRAP ON error NOPRINT
   ...       "Body of program and normal exit commands
RETURN
error:
   ...       "Error-handling and exit commands
SIGNAL ERRORNAME ERRORTEXT
```

TRIGGER command

The TRIGGER command associates a previously-created program to an object and identifies the object event that automatically executes the program; or a disassociates a trigger program from the object.

In order to assign a trigger program to an object, the object must be the one most recently defined or considered during the current session. When it is not, you must first use a CONSIDER command to make it the current definition.

> **See also:** "Trigger Programs" on page 1-14 for a general discussion, and the following statements for more specific information:
>
> - TRIGGER function, DESCRIBE command, and OBJ function that retrieve information about triggers.
> - TRIGGER_AW, TRIGGER_DEFINE, TRIGGER_AFTER_UPDATE, and TRIGGER_BEFORE_UPDATE which are trigger programs that you do not have to identify using the TRIGGER command.
> - USETRIGGERS option that you can use to disable all triggers.

Syntax

TRIGGER {*event-name* [*program-name*] }... | {DELETE *event-name*}... | DELETE ALL

where *event-name* is one of the following:

```
MAINTAIN
DELETE
PROPERTY
ASSIGN
BEFORE_UPDATE
AFTER_UPDATE
```

You can use the same keyword many times in a single TRIGGER statement; however, in this case, Oracle OLAP ignores all but the last occurrence of the keyword. See "Multiple Occurrences of the Same Keyword" on page 24-10, for details.

Arguments

MAINTAIN

Specifies that the trigger for the program is a Maintain event. A Maintain event is the execution of the MAINTAIN statement. As outlined in Table 24–5, "Subevents for the MAINTAIN Event" on page 24-30, the Maintain event has several subevents that correspond to the major keywords of the MAINTAIN command. Exactly when a program triggered by a Maintain event is executed is dependent on the Maintain subevent that triggered the program and the object type for which the Maintain event is defined:.

- Programs triggered by Maintain Add and Maintain Merge events on dimensions and composites are executed *after* the entire MAINTAIN statement executes.
- Programs triggered by Maintain Add and Maintain Merge events on dimension surrogates are executed multiple times—once *after*r each value is added or merged.
- Programs triggered by other Maintain subevents are executed *before* the MAINTAIN statement is executed.

DELETE

Specifies that the trigger for the program is a Delete event. A Delete event is a DELETE statement for the object. Oracle OLAP executes the specified program immediately before a DELETE statement deletes the object.

PROPERTY

Specifies that Oracle OLAP executes the specified program in response to a Property event. A Property event is the execution of a PROPERTY statement to create, modify, or delete an object property. A program that is triggered by a Property event is executed before the statement that triggered it.

ASSIGN

Specifies that Oracle OLAP executes the specified program in response to a Assign event. An Assign event is executed when SET assigns values to variable, relation, worksheet object, or a formula. A program that is triggered by SET is executed each time Oracle OLAP assigns a value to the object for which the event was defined. Thus, a program triggered by an Assign event is often executed over and over again as the assignment statements loops through a object assigning values.

UPDATE
When the object has been acquired using ACQUIRE in an analytic workspace that is attached in multiwriter mode, specifies that Oracle OLAP executes the specified program immediately after the object is updated.

> **Tip:** To specify processing when the entire analytic workspace is updated, create a TRIGGER_AFTER_UPDATE or TRIGGER_BEFORE_UPDATE program.

program-name
The name of the trigger program. When omitted for an event, the event does not trigger an action.

DELETE *event-name*
Deletes the triggers for the specified object events. Oracle OLAP disassociates the trigger program from the specified object event.

DELETE ALL
Deletes all of the triggers for the specified object. Oracle OLAP disassociates the trigger program from all events for object.

Notes

Multiple Occurrences of the Same Keyword

You can use all of the keywords in a single TRIGGER statement. However, if you use the same keyword twice in a TRIGGER statement, then Oracle OLAP recognized the last occurrence of the keyword; other occurrences are ignored.

For example, assume that you code the following TRIGGER statement.

```
TRIGGER PROPERTY progname1 PROPERTY progname2 PROPERTY progname3
```

When executing this TRIGGER statement, Oracle OLAP executes `progname3` immediately before a property of the object is created, modified, or deleted; Oracle OLAP does *not* execute `progname1` or `progname2`.

No Support for Recursive Triggers

Oracle OLAP does not support recursive triggers. You must set the USETRIGGERS option to `NO` before you issue the same DML statement *within* a trigger program that triggered the program itself. For example, assume that you have written a program named `TRIGGER_MAINTAIN_ADD` that is triggered by MAINTAIN ADD statements. Within the `TRIGGER_MAINTAIN_ADD` program, you must set the USETRIGGERS option to `NO` before you issue a MAINTAIN statement.

Characteristics of Trigger Programs

Trigger programs have certain characteristics depending on the statement that triggers them. Some trigger programs execute before the triggering statement executes; some after. Oracle OLAP passes arguments to programs triggered by some statements, but not others. Oracle OLAP does not change dimension status before most trigger programs execute, but does change dimension status before some MAINTAIN statements trigger program execution. In most cases, you can give a trigger program any name that you choose, but some events require a program with a specific name.

Table 24–1, " Trigger Program Characteristics" on page 24-12 lists the OLAP DML statements that trigger programs, the required name of the program (if any), whether or not Oracle OLAP uses values returned by the program, and whether or not Oracle OLAP passes arguments to the program.

Keep the following points in mind when designing trigger programs:

- Triggers that execute *before* the DML statement—For trigger programs that execute before the triggering OLAP DML statement executes, you can define the trigger program as a user-defined function that returns a `BOOLEAN` value. The value returned by the program determines whether or not Oracle OLAP executes the statement that triggered the execution of the trigger program. When the program returns `FALSE`, Oracle OLAP does not execute the triggering statement; when it returns `TRUE` or `NA`, the triggering statement executes.

- Arguments passed to trigger programs—Oracle OLAP passes arguments to some trigger programs. These programs are identified in Table 24–1, " Trigger Program Characteristics" on page 24-12. Descriptions of these arguments are provided in Table 24–2, " Arguments Passed to Trigger Programs" on page 24-13. Use the ARGUMENT command to declare these arguments in your program. Use VARIABLE to define program variables for the values. Use the WKSDATA function to retrieve the data type of an argument with a `WORKSHEET` data type.

- Assign trigger programs—Oracle OLAP executes a program triggered by an Assign event each time it assigns a value to the object for which the event was defined. Thus, a program triggered by an Assign event is often executed over and over again as the assignment statements loops through a object assigning values. With each execution, the value to be assigned is passed as argument1 to the Assign trigger program. (See Table 24–2, " Arguments Passed to Trigger

Programs" on page 24-13 for more information and Example 24–8, "An ASSIGN Trigger on a Variable" on page 24-17 for an example.) Within the Assign trigger program, you can use aTRIGGERASSIGN command to assign a different value than that specified by the assignment statement that triggered the execution of the Assign trigger program.

You can *only* assign values to a formula when the formula has an Assign trigger defined for it. When you assign a value to a formula with an Assign event, Oracle OLAP executes the trigger program for the event for assigned value and passes the assigned value to the trigger program. The Assign trigger does `not` change the definition of the formula itself. See Example 24–10, "An ASSIGN Trigger on a Formula" on page 24-24 for an example of an Assign trigger on a formula.

- Maintain trigger programs and dimension status —In some cases, Oracle OLAP changes the status of the dimension being maintained when a Maintain event triggers the execution of a program. See Table 24–3, "How Programs Triggered by Maintain Events Effect Dimension Status" on page 24-14 for details.
- Maintain triggers and dimension surrogates—Maintain triggers for dimension surrogates are different than Maintain triggers for other objects. You can only successfully issue a MAINTAIN statement against a dimension surrogate, when the dimension surrogate has a Maintain trigger. Issuing a MAINTAIN statement for a surrogate dimension that does *not* have a Maintain trigger, returns an error. Also, for Maintain Add and Maintain Merge triggers, whether or not an argument is passed to the program depends on the object on which the trigger is defined:
    - For dimension surrogates with a Maintain trigger, Oracle OLAP executes the trigger program one time for each value added or merged and passes that value into the program.
    - For other objects with a Maintain trigger, Oracle OLAP executes the trigger program only once after the MAINTAIN statement executes and no values are passed into the program

Table 24–1 Trigger Program Characteristics

Triggering Statement (event)	Program Name	Return Values	Passed Arguments
= command (SET)	No required name	No	Yes
AW command	TRIGGER_AW	No	No
DEFINE	TRIGGER_DEFINE	No	No

Table 24–1 (Cont.) Trigger Program Characteristics

Triggering Statement (event)	Program Name	Return Values	Passed Arguments
MAINTAIN ADD	No required name	No	No
MAINTAIN DELETE (*not* ALL)	No required name	Yes	No
MAINTAIN DELETE ALL	No required name	Yes	No
MAINTAIN MERGE	No required name	No	No
MAINTAIN MOVE	No required name	Yes	Yes
MAINTAIN RENAME	No required name	Yes	Yes
PROPERTY	No required name	Yes	Yes
UPDATE (Update AW)	TRIGGER_AFTER_UPDATE	No	No
UPDATE (Update AW)	TRIGGER_BEFORE_UPDATE	Yes	No
UPDATE (Update Multi)	No required name	No	No

Table 24–2 Arguments Passed to Trigger Programs

Event	Argument1	Argument2
Property	When the PROPERTY statement is assigning a property to an object, the name of the property. When the PROPERTY statement is deleting one or more properties, the literal DELETE. (TEXT data type)	When the value of *argument1* is DELETE, the name of the property or the literal ALL. In all other cases, the name of the property. (WORKSHEET data type)
Assignment	The value that you want to assign. When you know the data type of the object to which the value is assigned, specify that data type for the argument. When you do not know the actual data type, specify WORKSHEET as the data type of the argument.	None. Oracle OLAP passes only one argument to the program.
Maintain Add		(Dimension surrogates only) The value added. (WORKSHEET data type)
Maintain Rename	The dimension value that you want to rename. (TEXT data type)	The new name of the dimension member. (WORKSHEET data type)

Table 24–2 (Cont.) Arguments Passed to Trigger Programs

Event	Argument1	Argument2
Maintain Merge		(Dimension surrogates only) The value merged. (`WORKSHEET` data type)
Maintain Move	The position of the dimension value that you want to move. (`TEXT` data type)	The literal `BEFORE` or `AFTER`. (`WORKSHEET` data type)

Table 24–3 How Programs Triggered by Maintain Events Effect Dimension Status

Event Subevent	Dimension Status Before Program Execution
Maintain Add	Status set to dimension values just added.
Maintain Delete	Status set to dimension values about to be deleted.
Maintain Delete All	Current status is not changed.
Maintain Merge	Status set to dimension values just merged.
Maintain Move	Status set to dimension values about to be moved.
Maintain Rename	Current status is not changed.

Examples

Example 24–4 Creating Triggers

Assume that your analytic workspace contains a `TEXT` dimension named `city` and that you want to create programs that will automatically execute when a MAINTAIN statement executes against `city` or when a property is created or deleted for city. To create these triggers, you issue the following statements.

```
"Define the trigger programs
DEFINE trigger_maintain_move_city PROGRAM BOOLEAN
DEFINE trigger_property_city PROGRAM BOOLEAN
"Associate the trigger programs to events for the city dimension
CONSIDER city
TRIGGER PROPERTY trigger_property_city
TRIGGER MAINTAIN rigger_maintain_move_city
```

Example 24–5 Describing Triggers

Assume that you have created the triggers for `city` as described in Example 24–4, "Creating Triggers" on page 24-14. Later you want to see the description of the triggers, to do so you cannot merely issue a DESCRIBE statement for your analytic workspace. Instead, you must issue a FULLDSC statement.

```
DEFINE CITY DIMENSION TEXT
TRIGGER MAINTAIN RIGGER_MAINTAIN_MOVE_CITY -
        PROPERTY TRIGGER_PROPERTY_CITY

DEFINE TRIGGER_MAINTAIN_MOVE_CITY PROGRAM BOOLEAN

DEFINE TRIGGER_PROPERTY_CITY PROGRAM BOOLEAN
```

Example 24–6 Deleting Triggers

Assume that you have created the triggers described in Example 24–4, "Creating Triggers" on page 24-14. Now you want to delete the MAINTAIN trigger for `city`. To delete this trigger you issue the following statements.

```
CONSIDER city
TRIGGER DELETE MAINTAIN
```

When you issue a FULLDSC statement, you confirm that the MAINTAIN trigger for `city` has been deleted although the `trigger_maintain_move_city` program remains.

```
DEFINE CITY DIMENSION TEXT
TRIGGER PROPERTY TRIGGER_PROPERTY_CITY

DEFINE TRIGGER_MAINTAIN_MOVE_CITY PROGRAM BOOLEAN

DEFINE TRIGGER_PROPERTY_CITY PROGRAM BOOLEAN
```

To actually delete the `trigger_maintain_move_city` program you need to issue the following statement.

```
DELETE TRIGGER_MAINTAIN_MOVE_CITY
```

Example 24–7 A MAINTAIN Trigger Program

Assume that you have a dimension with the following definition in your analytic workspace.

```
DEFINE CITY DIMENSION TEXT
```

To create a Maintain trigger for `city`, you take the following steps:

1. Define the trigger program as a user-defined function. It can have any name that you want. The following statement defines a program named `trigger_maintain_city`.

   ```
   DEFINE trigger_maintain_city PROGRAM BOOLEAN
   ```

2. Specify the content of the program.

   ```
   PROGRAM
   SHOW JOINCHARS ('calltype = ' CALLTYPE)
   SHOW JOINCHARS ('triggering event = ' TRIGGER(EVENT))
   SHOW JOINCHARS ('triggering subevent = ' TRIGGER(SUBEVENT))
   RETURN TRUE
   END
   ```

3. Issue a TRIGGER command to associate the trigger program with the `city` dimension as a program to be executed when a Maintain event occurs. Remember to use a CONSIDER statement to make the definition for `city` the current definition.

   ```
   CONSIDER city
   TRIGGER MAINTAIN TRIGGER_MAINTAIN_CITY
   ```

When you issue a FULLDSC statement to see a full description of your analytic workspace, you can see the definition of `city` (including its Maintain trigger) and the `trigger_maintain_city` program.

```
DEFINE CITY DIMENSION TEXT
TRIGGER MAINTAIN TRIGGER_MAINTAIN_CITY

DEFINE TRIGGER_MAINTAIN_CITY PROGRAM BOOLEAN
PROGRAM
SHOW JOINCHARS ('calltype = ' CALLTYPE)
SHOW JOINCHARS ('triggering event = ' TRIGGER(EVENT))
SHOW JOINCHARS ('triggering subevent = ' TRIGGER(SUBEVENT))
RETURN TRUE
END
```

As illustrated in the following statements and output, when you issue MAINTAIN statements for city, the `trigger_maintain_city` program executes.

```
MAINTAIN city ADD 'Boston' 'Houston' 'Dallas'

calltype = TRIGGER
triggering event = MAINTAIN
triggering subevent = ADD

REPORT city

CITY
--------------
Boston
Houston
Dallas

MAINTAIN city MOVE 'Dallas' to 2

calltype = TRIGGER
triggering event = MAINTAIN
triggering subevent = MOVE

REPORT city

CITY
--------------
Boston
Dallas
Houston
```

Example 24–8 An ASSIGN Trigger on a Variable

Assume. that your analytic workspace contains objects with the following definitions.

```
DEFINE geog DIMENSION TEXT
DEFINE sales VARIABLE DECIMAL <geog>
DEFINE percent_sales VARIABLE INTEGER <geog>
```

The sales variable contains the values shown below. The percent_sales variable is empty.

```
GEOG              SALES
-------------- ----------
North America        0.59
Europe               9.35
Asia                   NA
```

Assume that you want specialized processing of values when you assign values to percent_sales. To handle this processing automatically, you can create a Assign trigger program for percent_sales by taking the following steps:

1. Create a trigger program that will execute each time you assign values to percent_sales.

   ```
   DEFINE TRIGGER_EQ PROGRAM BOOLEAN
   PROGRAM
   ARGUMENT datavalue WORKSHEET
   show 'description of triggering object = '
   DESCRIBE &TRIGGER(NAME)
   SHOW JOINCHARS ('calltype = ' CALLTYPE)
   SHOW JOINCHARS ('triggering event = ' TRIGGER(EVENT))
   SHOW JOINCHARS ('triggering subevent = ' TRIGGER(SUBEVENT))
   SHOW JOINCHARS ('value being assigned = ' datavalue)
   SHOW '  '
   END
   ```

2. Add an assign trigger to percent_sales using the TRIGGER command. Remember to first issue a CONSIDER command to make the definition for the e percent_sales variable the current definition.

   ```
   CONSIDER percent_sales
   TRIGGER ASSIGN TRIGGER_EQ
   ```

3. Assign values to percent_sales.

   ```
   percent_sales = (sales/TOTAL(sales))*100
   ```

Assigning values to `percent_sales` triggers the execution of the `trigger_eq` program and produces the following output lines.

```
description of triggering object =
DEFINE PERCENT_SALES VARIABLE INTEGER <GEOG>
TRIGGER ASSIGN TRIGGER_EQ
calltype = TRIGGER
triggering event = ASSIGN
triggering subevent =
value being assigned = 6
argument 2 =

description of triggering object =
DEFINE PERCENT_SALES VARIABLE INTEGER <GEOG>
TRIGGER ASSIGN TRIGGER_EQ
calltype = TRIGGER
triggering event = ASSIGN
triggering subevent =
value being assigned = 94
argument 2 =

description of triggering object =
DEFINE PERCENT_SALES VARIABLE INTEGER <GEOG>
TRIGGER ASSIGN TRIGGER_EQ
calltype = TRIGGER
triggering event = ASSIGN
triggering subevent =
value being assigned =
argument 2 =
```

Note: From the output you can see that Oracle OLAP called the `trigger_eq` program three times—each time it assigned a value to `percent_sales`.

4. When you issue REPORT commands for `sales` and `percent_sales` you can see the result of the calculations. The `percent_sales` variable contains values that are the percent of sales for each continent.

```
GEOG                      SALES
-------------- --------------------
North America                  0.59
Europe                         9.35
Asia                             NA

GEOG              PERCENT_SALES
-------------- --------------------
North America                     6
Europe                           94
Asia                             NA
```

Example 24–9 Setting Values in an ASSIGN Trigger Program

Assume that you have the following objects in your analytic workspace.

```
DEFINE GEOGRAPHY DIMENSION TEXT WIDTH 12
LD Geography Dimension Values

DEFINE PRODUCT DIMENSION TEXT WIDTH 12
LD Product Dimension Values

DEFINE TIME DIMENSION TEXT WIDTH 12
LD Time Dimension Values

DEFINE CHANNEL DIMENSION TEXT WIDTH 12
LD Channel Dimension Values

DEFINE F.MARGIN FORMULA DECIMAL <CHANNEL GEOGRAPHY PRODUCT TIME>
LD Margin
EQ f.sales-f.costs


DEFINE F.COSTS VARIABLE SHORT <GEOGRAPHY PRODUCT CHANNEL TIME>
LD Costs

DEFINE F.SALES VARIABLE SHORT <GEOGRAPHY PRODUCT CHANNEL TIME>
LD Sales
```

Note that `f.costs`, `f.sales`, and `f.margin` all have the same dimensions.

Now you add an Assign trigger to `f.margin` that will execute a program named `t.margin`. The definition of `f.margin` is modified to the following definition.

```
DEFINE F.MARGIN FORMULA DECIMAL <CHANNEL GEOGRAPHY PRODUCT TIME>
LD Margin
TRIGGER ASSIGN T.MARGIN
EQ f.sales-f.costs
```

Now you actually write the `t.margin` program. When an expression is assigned to the `f.margin` formula, the program uses this value to compute new values for `f.costs` and `f.sales`.

```
DEFINE T.MARGIN PROGRAM
PROGRAM
ARG newVal DECIMAL       " The value passed to the program by the Assign trigger
VARIABLE t.valDiff DECIMAL     " Difference between newVal and old value
VARIABLE t.costInc DECIMAL     " Amount the difference makes to costs
"show the value of newVal
SHOW 'newVal = ' NONL
SHOW newVal
" Compute the difference between the current value and the new one
t.valDiff = newVal - f.margin
" Now increase costs proportional to their existing amounts
t.costInc = (newVal - f.margin) * (f.costs/f.sales)
" Adjust the values of sales and costs to get the new value
SET1 f.costs = f.costs + t.costInc

SET1 f.sales =  f.sales + t.valDiff + t.costInc

SHOW geography NONL
SHOW ' ' NONL
SHOW product NONL
SHOW ' ' NONL
SHOW channel NONL
SHOW ' ' NONL
SHOW time NONL
SHOW ' f.costs = 'NONL
SHOW f.costs NONL
SHOW ' f.sales = 'NONL
SHOW f.sales
END
```

Now assume that you issue the following LIMIT statements to identify a subset of data and issue a REPORT statement to report on the values of f.margin.

```
LIMIT t0.hierdim TO 'STANDARD'
LIMIT time TO t0.levelrel EQ 'L2'
LIMIT geography TO FIRST 1
LIMIT channel TO FIRST 1
LIMIT product TO FIRST 5
REPORT DOWN time ACROSS product: f.margin

GEOGRAPHY: WORLD
CHANNEL: TOTALCHANNEL
               -----------------------F.MARGIN-----------------------
               -----------------------PRODUCT------------------------
TIME           TOTALPROD   AUDIODIV  PORTAUDIO    PORTCD     PORTST
-------------- ---------- ---------- ---------- ---------- ----------
Q1.96          54,713,974 29,603,546  5,379,661  2,480,914  1,615,708
Q2.96          63,919,784 34,594,087  6,331,848  2,869,265  1,931,785
Q3.96          58,303,490 31,543,152  5,792,725  2,616,515  1,795,701
Q4.96          71,197,892 38,383,878  7,059,581  3,163,804  2,232,880
Q1.97          55,489,723 29,989,262  5,368,237  2,491,475  1,607,344
Q2.97          41,687,908 22,532,979  4,070,725  1,855,992  1,245,161
```

Now you issue the following assignment statement that increase the value of `f.margin` by 10% and report it

```
f.margin = f.margin * 1.1
```

The execution of this assignment statement triggers the execution of the Assign trigger program named `t.margin`. The output of that program follows.

```
newVal = 60,185,371.40
WORLD TOTALPROD TOTALCHANNEL Q1.96 f.costs = 1,298,474.00 f.sales = 61,483,840.00
newVal = 32,563,900.67
WORLD AUDIODIV TOTALCHANNEL Q1.96 f.costs = 664,226.90 f.sales = 33,228,130.00
newVal = 5,917,626.67
WORLD PORTAUDIO TOTALCHANNEL Q1.96 f.costs = 97,976.04 f.sales = 6,015,603.00
newVal = 2,729,005.43
WORLD PORTCD TOTALCHANNEL Q1.96 f.costs = 34,301.53 f.sales = 2,763,307.00
newVal = 1,777,278.95
WORLD PORTST TOTALCHANNEL Q1.96 f.costs = 25,160.72 f.sales = 1,802,440.00
newVal = 70,311,762.13
WORLD TOTALPROD TOTALCHANNEL Q2.96 f.costs = 1,504,051.00 f.sales = 71,815,820.00
newVal = 38,053,495.70
WORLD AUDIODIV TOTALCHANNEL Q2.96 f.costs = 768,788.10 f.sales = 38,822,280.00
newVal = 6,965,032.86
```

```
WORLD PORTAUDIO TOTALCHANNEL Q2.96 f.costs = 114,558.20 f.sales = 7,079,591.00
newVal = 3,156,191.20
WORLD PORTCD TOTALCHANNEL Q2.96 f.costs = 39,256.88 f.sales = 3,195,448.00
newVal = 2,124,963.02
WORLD PORTST TOTALCHANNEL Q2.96 f.costs = 29,780.54 f.sales = 2,154,744.00
newVal = 64,133,838.86
WORLD TOTALPROD TOTALCHANNEL Q3.96 f.costs = 1,350,733.00 f.sales = 65,484,570.00
newVal = 34,697,467.06
WORLD AUDIODIV TOTALCHANNEL Q3.96 f.costs = 691,887.10 f.sales = 35,389,360.00
newVal = 6,371,997.63
WORLD PORTAUDIO TOTALCHANNEL Q3.96 f.costs = 103,203.70 f.sales = 6,475,202.00
newVal = 2,878,166.40
WORLD PORTCD TOTALCHANNEL Q3.96 f.costs = 35,358.18 f.sales = 2,913,525.00
newVal = 1,975,270.68
WORLD PORTST TOTALCHANNEL Q3.96 f.costs = 27,339.77 f.sales = 2,002,611.00
newVal = 78,317,681.06
WORLD TOTALPROD TOTALCHANNEL Q4.96 f.costs = 1,618,915.00 f.sales = 79,936,590.00
newVal = 42,222,265.94
WORLD AUDIODIV TOTALCHANNEL Q4.96 f.costs = 826,923.40 f.sales = 43,049,190.00
newVal = 7,765,539.34
WORLD PORTAUDIO TOTALCHANNEL Q4.96 f.costs = 123,269.50 f.sales = 7,888,809.00
newVal = 3,480,184.35
WORLD PORTCD TOTALCHANNEL Q4.96 f.costs = 41,998.90 f.sales = 3,522,183.00
newVal = 2,456,168.00
WORLD PORTST TOTALCHANNEL Q4.96 f.costs = 33,357.19 f.sales = 2,489,525.00
newVal = 61,038,695.03
WORLD TOTALPROD TOTALCHANNEL Q1.97 f.costs = 1,423,963.00 f.sales = 62,462,660.00
newVal = 32,988,187.65
WORLD AUDIODIV TOTALCHANNEL Q1.97 f.costs = 679,477.80 f.sales = 33,667,660.00
newVal = 5,905,060.56
WORLD PORTAUDIO TOTALCHANNEL Q1.97 f.costs = 158,854.40 f.sales = 6,063,915.00
newVal = 2,740,622.56
WORLD PORTCD TOTALCHANNEL Q1.97 f.costs = 53,144.41 f.sales = 2,793,767.00
newVal = 1,768,078.14
WORLD PORTST TOTALCHANNEL Q1.97 f.costs = 40,784.62 f.sales = 1,808,863.00
newVal = 45,856,698.46
WORLD TOTALPROD TOTALCHANNEL Q2.97 f.costs = 1,070,465.00 f.sales = 46,927,160.00
newVal = 24,786,276.35
WORLD AUDIODIV TOTALCHANNEL Q2.97 f.costs = 512,435.60 f.sales = 25,298,710.00
newVal = 4,477,797.64
WORLD PORTAUDIO TOTALCHANNEL Q2.97 f.costs = 118,791.70 f.sales = 4,596,590.00
newVal = 2,041,591.56
WORLD PORTCD TOTALCHANNEL Q2.97 f.costs = 39,287.77 f.sales = 2,080,879.00
newVal = 1,369,677.57
WORLD PORTST TOTALCHANNEL Q2.97 f.costs = 30,038.08 f.sales = 1,399,716.00
```

Example 24–10 An ASSIGN Trigger on a Formula

The way Oracle OLAP handles assigning values to a formula varies depending on whether or not the formula has an Assign trigger as part of its definition.

Assume your analytic workspace contains objects with the following definitions and values.

```
DEFINE GEOG.D DIMENSION TEXT

DEFINE SALES VARIABLE DECIMAL <GEOG.D>

DEFINE F_MODIFIED_SALES FORMULA DECIMAL <GEOG.D>
EQ sales+20
```

A report of `f_modified_sales` formula displays the following report that contains the values computed by the formula.

```
REPORT f_modified_sales

         -------------F_MODIFIED_SALES--------------
         ------------------GEOG.D-------------------
TIME.D     Boston    Medford   San Diego  Sunnydale
-------- ---------- ---------- ---------- ----------
Jan76          0.00   1,000.00   2,000.00   3,000.00
Feb76      1,000.00   3,000.00   5,000.00   7,000.00
Mar76      2,000.00   5,000.00   8,000.00  11,000.00
76Q1             NA         NA         NA         NA
```

The `f_modified_sales` formula does not presently have an Assign trigger on it. Consequently, as illustrated in the following code, any attempt to assign values to `f_modified_sales` results in an error.

```
f_modified_sales = 3
ORA-34142: You cannot assign values to a FORMULA.
```

To create an Assign trigger on `f_modified_sales` take the following steps:

1. Define the trigger program

```
DEFINE TRIGGER_ASSIGN_MODIFIED_SALES PROGRAM
PROGRAM
ARGUMENT datavalue NUMBER
SHOW 'description of triggering object = '
DESCRIBE &TRIGGER(NAME)
SHOW JOINCHARS ('calltype = ' CALLTYPE)
SHOW JOINCHARS ('triggering event = ' TRIGGER(EVENT))
SHOW JOINCHARS ('value being assigned = ' datavalue)
SHOW '  '
END
```

2. Add the Assign trigger to the definition of the formula using the following statements.

```
CONSIDER f_modified_sales
TRIGGER ASSIGN trigger_assign_modified_sales
```

   Issuing a `FULLDSC f_modified_sales` statement displays the new complete definition for `f_modified_sales`.

```
DEFINE F_MODIFIED_SALES FORMULA DECIMAL <GEOG.D>
TRIGGER ASSIGN TRIGGER_ASSIGN_MODIFIED_SALES
EQ sales+20
```

3. Now when you issue the following statement to assign a value to `f_modified_sales`, an error does not occur. Instead, the trigger program executes 4 times, once for each dimension value of sales.

```
f_modified_sales = 3

description of triggering object =
DEFINE F_MODIFIED_SALES FORMULA DECIMAL <GEOG.D>
TRIGGER ASSIGN TRIGGER_ASSIGN_MODIFIED_SALES
EQ sales-1000
calltype = TRIGGER
triggering event = ASSIGN
value being assigned = 3.00

description of triggering object =
DEFINE F_MODIFIED_SALES FORMULA DECIMAL <GEOG.D>
TRIGGER ASSIGN TRIGGER_ASSIGN_MODIFIED_SALES
EQ sales-1000
calltype = TRIGGER
triggering event = ASSIGN
value being assigned = 3.00

description of triggering object =
DEFINE F_MODIFIED_SALES FORMULA DECIMAL <GEOG.D>
TRIGGER ASSIGN TRIGGER_ASSIGN_MODIFIED_SALES
EQ sales-1000
calltype = TRIGGER
triggering event = ASSIGN
value being assigned = 3.00

description of triggering object =
DEFINE F_MODIFIED_SALES FORMULA DECIMAL <GEOG.D>
TRIGGER ASSIGN TRIGGER_ASSIGN_MODIFIED_SALES
EQ sales-1000
calltype = TRIGGER
triggering event = ASSIGN
value being assigned = 3.00
```

4. However, as issuing a REPORT statement for `f_modified_salesd` illustrates, the values calculated by a simple execution of the formula have not changed.

```
REPORT f_modified_sales

GEOG.D               F_MODIFIED_SALES
------------ -----------------------------------
Boston                                    30.00
Medford                                   32.21
San Diego                                 33.03
Sunnydale                                 38.32
```

TRIGGER function

The TRIGGER function retrieves the event, subevent, or name of the object or analytic workspace that caused the execution of a trigger program (that is, a TRIGGER_DEFINE, TRIGGER_AFTER_UPDATE, or TRIGGER_BEFORE_UPDATE program, or any program identified as a trigger program using the TRIGGER command).

When the current program is a trigger program, the TRIGGER function returns the trigger information for that program. When it is not, the TRIGGER function returns trigger information for the most recently executed trigger program.

See also: "Trigger Programs" on page 1-14 and the following statements:

- TRIGGER command, DESCRIBE command, and OBJ function that retrieve information about triggers.
- TRIGGER_AW, TRIGGER_DEFINE, TRIGGER_AFTER_UPDATE, and TRIGGER_BEFORE_UPDATE which are trigger programs that you do not have to identify using the TRIGGER command.
- USETRIGGERS option that you can use to disable all triggers.

Return Values

```
TEXT
```

Syntax

TRIGGER (NAME | EVENT | SUBEVENT)

Return Values

NAME
For a program identified as a trigger program using the TRIGGER command, returns the object for which the trigger program is association. For a TRIGGER_AW, TRIGGER_DEFINE, TRIGGER_AFTER_UPDATE, or TRIGGER_BEFORE_UPDATE program, returns the name of the analytic workspace that caused the program to execute.

EVENT
Returns the name of the event (DML statement) that triggered the execution of the program.

AW
MAINTAIN
DELETE
DEFINE
PROPERTY
ASSIGN
BEFORE_UPDATE
AFTER_UPDATE

For more information on trigger events, see TRIGGER command and TRIGGER_DEFINE.

SUBEVENT
When the value returned by EVENT is `MAINTAIN`, `AFTER_UPDATE` or `BEFORE_UPDATE`, returns more information on the OLAP DML command that triggered the execution of the program. Valid subevents for `AW` are outlined in Table 24–4, " Subevents for the AW Event" on page 24-29. Valid subevents for `MAINTAIN` are outlined in Table 24–5, "Subevents for the MAINTAIN Event" on page 24-30. Valid subevents for `UPDATE` are outlined in Table 24–6, "Subevents for UPDATE Events" on page 24-30.

Table 24–4 Subevents for the AW Event

Subevent	Description
CREATE	Returned when a AW CREATE statement triggered the execution of the program.
ATTACH	Returned when a AW ATTACH statement triggered the execution of the program.
DELETE	Returned when a AW DELETE statement triggered the execution of the program.
DETACH	Returned when a AW DETACH statement triggered the execution of the program.

Table 24–5 Subevents for the MAINTAIN Event

Subevent	Description
ADD	Returned when a MAINTAIN ADD statement triggered the execution of the program.
DELETE	Returned when any MAINTAIN DELETE statement *except* a MAINTAIN DELETE ALL statement triggered the execution of the program.
DELETE ALL	Returned when a MAINTAIN DELETE ALL statement triggered the execution of the program.
MERGE	Returned when a MAINTAIN MERGE statement triggered the execution of the program.
MOVE	Returned when a MAINTAIN MOVE statement triggered the execution of the program.
RENAME	Returned when a MAINTAIN RENAME statement triggered the execution of the program.

Table 24–6 Subevents for UPDATE Events

Subevent	Description
AW	Returned when an UPDATE statement triggered the execution of a TRIGGER_AFTER_UPDATE or TRIGGER_BEFORE_UPDATE program.
MULTI	Returned when an UPDATE statement triggered the execution of a program identified as a trigger program using the TRIGGER command when an object is acquired in multiwriter mode.

Examples

For examples of using the TRIGGER function, see Example 24–12, "TRIGGER_BEFORE_UPDATE Program" on page 24-35, Example 24–7, "A MAINTAIN Trigger Program" on page 24-15, Example 24–11, "TRIGGER_AFTER_UPDATE Program" on page 24-32, Example 24–13, "A TRIGGER_DEFINE Program" on page 24-37, and Example 24–14, "Assigning an Alternative Value using an Assign Trigger" on page 24-38.

TRIGGER_AFTER_UPDATE

A TRIGGER_AFTER_UPDATE program is a program that you create and that Oracle OLAP checks for by name when an UPDATE statement executes. When the program exists in the same analytic workspace that you are updating, Oracle OLAP executes the program after executing the UPDATE.

Note: The USETRIGGERS option must be set to its default value of `TRUE` for a TRIGGER_AFTER_UPDATE program to execute

See also: A TRIGGER_AFTER_UPDATE program is only one of a number of trigger programs that you can write. You can write other trigger programs as described in TRIGGER command, TRIGGER_BEFORE_UPDATE, TRIGGER_DEFINE, and "Trigger Programs" on page 1-14.

Notes

No Support for Recursive Triggers

Oracle OLAP does not support recursive triggers. You must set the USETRIGGERS option to `NO` before you issue an UPDATE statement within a TRIGGER_AFTER_UPDATE program.

Syntax

To create a program with the name `TRIGGER_AFTER_UPDATE`, follow the guidelines presented in "Trigger Programs" on page 1-14.

Examples

Example 24–11 TRIGGER_AFTER_UPDATE Program

Assume you have defined the following program in your analytic workspace.

```
DEFINE TRIGGER_AFTER_UPDATE PROGRAM
PROGRAM
SHOW JOINCHARS ('calltype = ' CALLTYPE)
SHOW JOINCHARS ('triggering event = ' TRIGGER(EVENT))
SHOW JOINCHARS ('triggering subevent = ' TRIGGER(SUBEVENT))

END
```

When you issue an UPDATE statement the program executes and displays the following output.

```
calltype = TRIGGER
triggering event = AFTER_UPDATE
triggering subevent = AW
```

TRIGGER_AW

A TRIGGER_AW program is a program that you create and that Oracle OLAP checks for by name when an AW command executes. When the program exists in the same analytic workspace that you are updating, Oracle OLAP executes the program and then, depending on the value returned by the program (if any), either does nor does not update the workspace.

Note: The USETRIGGERS option must be set to its default value of `TRUE` for a TRIGGER_AW program to execute

---

See also: A TRIGGER_AW program is only one of a number of trigger programs that you can write. You can write other trigger programs as described in "Trigger Programs" on page 1-14.

Return Value

You can write the program as a function that returns a `BOOLEAN` value. In this case, when the program returns `FALSE`, Oracle OLAP does not execute the UPDATE statement that triggered the execution of the TRIGGER_AW program; when the program returns `TRUE` or `NA`, the AW command executes.

Notes

No Support for Recursive Triggers

Oracle OLAP does not support recursive triggers. You must set the USETRIGGERS option to `NO` before you issue an AW command within a TRIGGER_AW program.

Syntax

To create a program with the name `TRIGGER_AW`, follow the guidelines presented in"Trigger Programs" on page 1-14.

TRIGGER_BEFORE_UPDATE

A TRIGGER_BEFORE_UPDATE program is a program that you create and that Oracle OLAP checks for by name when an UPDATE statement executes. When the program exists in the same analytic workspace that you are updating, Oracle OLAP executes the program and then, depending on the value returned by the program (if any), either does nor does not update the workspace.

Note: The USETRIGGERS option must be set to its default value of `TRUE` for a TRIGGER_BEFORE_UPDATE program to execute

See also: A TRIGGER_BEFORE_UPDATE program is only one of a number of trigger programs that you can write. You can write other trigger programs as described in TRIGGER command, TRIGGER_AFTER_UPDATE, TRIGGER_DEFINE, and "Trigger Programs" on page 1-14.

Return Value

You can write the program as a function that returns a `BOOLEAN` value. In this case, when the program returns `FALSE`, Oracle OLAP does not execute the UPDATE statement that triggered the execution of the TRIGGER_BEFORE_UPDATE program; when the program returns `TRUE` or `NA`, the UPDATE statement executes.

Notes

No Support for Recursive Triggers

Oracle OLAP does not support recursive triggers. You must set the USETRIGGERS option to `NO` before you issue an UPDATE statement within a TRIGGER_BEFORE_UPDATE program.

Syntax

To create a program with the name `TRIGGER_UPDATE`, follow the guidelines presented in"Trigger Programs" on page 1-14.

Examples

Example 24–12 TRIGGER_BEFORE_UPDATE Program

Assume that an analytic workspace named `myaw` has an TRIGGER_BEFORE_UPDATE program with the following definition.

```
DEFINE TRIGGER_BEFORE_UPDATE PROGRAM BOOLEAN
PROGRAM
SHOW JOINCHARS ('calltype = ' CALLTYPE)
SHOW JOINCHARS ('triggering event = ' TRIGGER(EVENT))
SHOW JOINCHARS ('triggering subevent = ' TRIGGER(SUBEVENT))
RETURN TRUE
END
```

Assume that you define a `TEXT` variable named `myvar` and, then, issue an UPDATE statement. The TRIGGER_BEFORE_UPDSATE program executes.

```
calltype = TRIGGER
triggering event = BEFORE_UPDATE
triggering subevent = AW
```

Because the program returned TRUE, the definition for `myvar` exists after you detach and reattach the workspace.

```
AW DETACH myaw
AW ATTACH myaw
DESCRIBE

DEFINE TRIGGER_BEFORE_UPDATE PROGRAM BOOLEAN
PROGRAM
SHOW JOINCHARS ('calltype = ' CALLTYPE)
SHOW JOINCHARS ('triggering event = ' TRIGGER(EVENT))
SHOW JOINCHARS ('triggering subevent = ' TRIGGER(SUBEVENT))
RETURN TRUE
END

DEFINE MYVAR VARIABLE TEXT
```

However, if you modified the program so that it returned `FALSE`, then when you detach and reattach the workspace, not only would the `myvar` definition not in the workspace, the definition for the TRIGGER_BEFORE_UPDATE program would also not be in the workspace.

TRIGGER_DEFINE

A TRIGGER_DEFINE program is a program that you create and that Oracle OLAP checks for by name when a DEFINE statement executes. When the program exists in the same analytic workspace in which you are defining a new object, Oracle OLAP executes the program.

Note: The USETRIGGERS option must be set to its default value of `TRUE` for a TRIGGER_DEFINE program to execute

See also: A TRIGGER_DEFINE program is only one of a number of trigger programs that you can write. You can write other trigger programs as described in TRIGGER command, TRIGGER_AFTER_UPDATE, TRIGGER_BEFORE_UPDATE, and "Trigger Programs" on page 1-14.

Syntax

To create a program with the name `TRIGGER_DEFINE`, follow the guidelines presented in "Trigger Programs" on page 1-14.

Notes

No Support for Recursive Triggers

Oracle OLAP does not support recursive triggers. You must set the USETRIGGERS option to `NO` before you issue a DEFINE statement within a TRIGGER_DEFINE program.

Examples

Example 24–13 A TRIGGER_DEFINE Program

Assume that you have written a TRIGGER_DEFINE program with the following description in your analytic workspace.

```
DEFINE TRIGGER_DEFINE PROGRAM
PROGRAM
SHOW JOINCHARS ('calltype = ' CALLTYPE)
SHOW JOINCHARS ('triggering event = ' TRIGGER(EVENT))
SHOW JOINCHARS ('fully-qualified object name ='TRIGGER(NAME))
SHOW JOINCHARS ('type of object = 'OBJ(TYPE TRIGGER(NAME))
DESCRIBE &TRIGGER(NAME)
END
```

Assume, as shown in the following statements, that you issue a DEFINE VARIABLE statement to define a variable named `myvar`. As shown by the output following the statement, Oracle OLAP defines the variable and executes the TRIGGER_DEFINE program.

```
DEFINE myvar VARIABLE TEXT
calltype = TRIGGER
triggering event = DEFINE
fully-qualified object name =MYAW!MYVAR
type of object = VARIABLE

DEFINE MYVAR VARIABLE TEXT
```

TRIGGERASSIGN

Within a program triggered by an Assign event, assigns a value that is different from the value specified by the assignment statement that triggered the execution of the program.

Note: The USETRIGGERS option must be set to its default value of `TRUE` for this command to execute

See: "Trigger Programs" on page 1-14 and TRIGGER command for more information about creating trigger programs for Assign events.

Data type

The data type of the object to which Oracle OLAP assigns the value.

Syntax

TRIGGERASSIGN *value*

Arguments

value
The value that you want assigned.

Examples

Example 24–14 Assigning an Alternative Value using an Assign Trigger

Assume that you have objects with the following descriptions in your analytic workspace.

```
DEFINE GEOG.D DIMENSION TEXT
DEFINE TIME.D DIMENSION TEXT
DEFINE TIME.PARENTREL RELATION TIME.D <TIME.D>
DEFINE SALES VARIABLE DECIMAL <GEOG.D TIME.D>
DEFINE MODIFIED_SALES VARIABLE DECIMAL <GEOG.D TIME.D>
```

Assume also that you have populated the `sales` variable with the values shown in the following report, but that you have not yet populated the `modified_sales` variable.

```
             -----------------------SALES-----------------------
             ----------------------GEOG.D-----------------------
TIME.D          Boston      Medford     San Diego    Sunnydale
------------ ------------ ------------ ------------ ------------
Jan76            1,000.00     2,000.00     3,000.00     4,000.00
Feb76            2,000.00     4,000.00     6,000.00     8,000.00
Mar76            3,000.00     6,000.00     9,000.00    12,000.00
76Q1                   NA           NA           NA           NA
```

Now you want to assign values to the `modified_sales` variable using various expressions, however, you want to ensure that the values never are less than or equal to 1,000. You can assure this processing by taking the following steps:

1. Create the following program that checks for values less than or equal to 1000 condition.

```
DEFINE TRIGGER_ASSIGN_MODIFIED_SALES PROGRAM
PROGRAM
ARGUMENT datavalue DECIMAL
IF datavalue LE 1000
 THEN TRIGGERASSIGN 1000
show 'description of triggering object = '
DESCRIBE &TRIGGER(NAME)
SHOW JOINCHARS ('calltype = ' CALLTYPE)
SHOW JOINCHARS ('triggering event = ' TRIGGER(EVENT))
SHOW JOINCHARS ('triggering subevent = ' TRIGGER(SUBEVENT))
SHOW JOINCHARS ('value passed to program = ' datavalue)
SHOW '  '
END
```

2. Issue the following statements to add an Assign trigger to the `modified_sales` variable. The `trigger_assign_modified_sales` program is the trigger program.

```
CONSIDER modified_sales
TRIGGER ASSIGN trigger_assign_modified_sales
```

3. Assign values to `modified_sales`.

```
modified_sales = sales - 1000
```

4. This statement triggers the execution of the `trigger_assign_modified_sales` program for each value that Oracle OLAP assigns.

```
description of triggering object =
DEFINE MODIFIED_SALES VARIABLE DECIMAL <GEOG.D TIME.D>
TRIGGER ASSIGN TRIGGER_ASSIGN_MODIFIED_SALES
calltype = TRIGGER
triggering event = ASSIGN
triggering subevent =
value passed to program = 0.00

description of triggering object =
DEFINE MODIFIED_SALES VARIABLE DECIMAL <GEOG.D TIME.D>
TRIGGER ASSIGN TRIGGER_ASSIGN_MODIFIED_SALES
calltype = TRIGGER
triggering event = ASSIGN
triggering subevent =
value passed to program = 1,000.00

description of triggering object =
DEFINE MODIFIED_SALES VARIABLE DECIMAL <GEOG.D TIME.D>
TRIGGER ASSIGN TRIGGER_ASSIGN_MODIFIED_SALES
calltype = TRIGGER
triggering event = ASSIGN
triggering subevent =
value passed to program = 2,000.00

description of triggering object =
DEFINE MODIFIED_SALES VARIABLE DECIMAL <GEOG.D TIME.D>
TRIGGER ASSIGN TRIGGER_ASSIGN_MODIFIED_SALES
calltype = TRIGGER
triggering event = ASSIGN
triggering subevent =
value passed to program = 3,000.00

description of triggering object =
DEFINE MODIFIED_SALES VARIABLE DECIMAL <GEOG.D TIME.D>
TRIGGER ASSIGN TRIGGER_ASSIGN_MODIFIED_SALES
calltype = TRIGGER
triggering event = ASSIGN
triggering subevent =
value passed to program = 1,000.00

description of triggering object =
DEFINE MODIFIED_SALES VARIABLE DECIMAL <GEOG.D TIME.D>
TRIGGER ASSIGN TRIGGER_ASSIGN_MODIFIED_SALES
calltype = TRIGGER
triggering event = ASSIGN
triggering subevent =
value passed to program = 3,000.00

description of triggering object =
DEFINE MODIFIED_SALES VARIABLE DECIMAL <GEOG.D TIME.D>
TRIGGER ASSIGN TRIGGER_ASSIGN_MODIFIED_SALES
calltype = TRIGGER
triggering event = ASSIGN
triggering subevent =
value passed to program = 5,000.00

description of triggering object =
DEFINE MODIFIED_SALES VARIABLE DECIMAL <GEOG.D TIME.D>
TRIGGER ASSIGN TRIGGER_ASSIGN_MODIFIED_SALES
calltype = TRIGGER
triggering event = ASSIGN
triggering subevent =
value passed to program = 7,000.00

description of triggering object =
DEFINE MODIFIED_SALES VARIABLE DECIMAL <GEOG.D TIME.D>
TRIGGER ASSIGN TRIGGER_ASSIGN_MODIFIED_SALES
calltype = TRIGGER
triggering event = ASSIGN
triggering subevent =
value passed to program = 2,000.00

description of triggering object =
DEFINE MODIFIED_SALES VARIABLE DECIMAL <GEOG.D TIME.D>
TRIGGER ASSIGN TRIGGER_ASSIGN_MODIFIED_SALES
calltype = TRIGGER
triggering event = ASSIGN
triggering subevent =
value passed to program = 5,000.00

description of triggering object =
DEFINE MODIFIED_SALES VARIABLE DECIMAL <GEOG.D TIME.D>
TRIGGER ASSIGN TRIGGER_ASSIGN_MODIFIED_SALES
calltype = TRIGGER
triggering event = ASSIGN
triggering subevent =
value passed to program = 8,000.00

description of triggering object =
DEFINE MODIFIED_SALES VARIABLE DECIMAL <GEOG.D TIME.D>
TRIGGER ASSIGN TRIGGER_ASSIGN_MODIFIED_SALES
calltype = TRIGGER
triggering event = ASSIGN
triggering subevent =
value passed to program = 11,000.00

description of triggering object =
DEFINE MODIFIED_SALES VARIABLE DECIMAL <GEOG.D TIME.D>
TRIGGER ASSIGN TRIGGER_ASSIGN_MODIFIED_SALES
calltype = TRIGGER
triggering event = ASSIGN
triggering subevent =
value passed to program =

description of triggering object =
DEFINE MODIFIED_SALES VARIABLE DECIMAL <GEOG.D TIME.D>
TRIGGER ASSIGN TRIGGER_ASSIGN_MODIFIED_SALES
calltype = TRIGGER
triggering event = ASSIGN
triggering subevent =
value passed to program =

description of triggering object =
DEFINE MODIFIED_SALES VARIABLE DECIMAL <GEOG.D TIME.D>
TRIGGER ASSIGN TRIGGER_ASSIGN_MODIFIED_SALES
calltype = TRIGGER
triggering event = ASSIGN
triggering subevent =
value passed to program =

description of triggering object =
DEFINE MODIFIED_SALES VARIABLE DECIMAL <GEOG.D TIME.D>
TRIGGER ASSIGN TRIGGER_ASSIGN_MODIFIED_SALES
calltype = TRIGGER
triggering event = ASSIGN
triggering subevent =
value passed to program =
```

5. The following report of `modified_sales` shows that all values are at least 1,000.

```
             -------------------MODIFIED_SALES-------------------
             -----------------------GEOG.D-----------------------
TIME.D          Boston      Medford     San Diego    Sunnydale
------------ ------------ ------------ ------------ ------------
Jan76            1,000.00     1,000.00     2,000.00     3,000.00
Feb76            1,000.00     3,000.00     5,000.00     7,000.00
Mar76            2,000.00     5,000.00     8,000.00    11,000.00
76Q1                   NA           NA           NA           NA
```

TRIGGERMAXDEPTH

The TRIGGERMAXDEPTH option determines the maximum number of $NATRIGGER property expressions that Oracle OLAP can execute simultaneously.

Data type

INTEGER

Syntax

TRIGGERMAXDEPTH = *n*

Arguments

n

An INTEGER expression that specifies the maximum number of $NATRIGGER property expressions that can execute simultaneously. The default value is `50`.

Notes

About the $NATRIGGER Property

The TRIGGERMAXDEPTH option works in conjunction with the $NATRIGGER property of a variable.

Recursive Triggers

While an $NATRIGGER expression is executing, it cannot be invoked again by a formula, program, or other $NATRIGGER expression that it invokes unless the RECURSIVE option is set to YES. The TRIGGERMAXDEPTH option governs the depth of recursion of $NATRIGGER expressions and prevents infinite recursions or excessively deep recursions, which can cause Oracle OLAP to malfunction.

Examples

Example 24–15 Setting the Maximum Trigger Depth

This example sets the maximum trigger depth, exceeds it, then sets the depth to a higher value. Usually the TRIGGERMAXDEPTH value would be much higher than `2`, which is used in this example. The default value is `50`.

```
DEFINE d1 INTEGER DIMENSION
MAINTAIN d1 ADD 2
DEFINE v1 DECIMAL <d1>
PROPERTY '$NATRIGGER' 'v2 + 1'
DEFINE v2 DECIMAL <d1>
PROPERTY '$NATRIGGER' 'v3 + 1'
DEFINE v3 DECIMAL <d1>
PROPERTY '$NATRIGGER' 'v4 + 1'
DEFINE v4 DECIMAL <d1>
v4(d1 1) = 333.3
RECURSIVE = YES
TRIGGERMAXDEPTH = 2
SHOW v1
```

The preceding statements produce the following output.

```
ERROR: Depth of NA trigger calls exceeds allowable (maximum depth 2)
```

The following statements set the maximum trigger depth to a higher value and show the value of the variable.

```
TRIGGERMAXDEPTH = 3
SHOW v1
```

The preceding statements produce the following output.

```
336.3
```

TRIGGERSTOREOK

The TRIGGERSTOREOK option controls whether you can use $STORETRIGGERVAL properties to specify that `NA` values in an object be permanently replaced by the values specified by a $NATRIGGER property.

Important: The value of the TRIGGERSTOREOK option is only one factor that Oracle OLAP uses to determine what to do with variable data that is the result of $NATRIGGER expression execution. For a discussion of the other factors and their interrelationship, see "How Oracle OLAP Determines Whether to Store or Cache Results of $NATRIGGER" on page 6-21.

Data type

BOOLEAN

Syntax

TRIGGERSTOREOK = {<u>NO</u>|YES}

Arguments

NO
YES
Specifies whether or not `NA` values are permanently replaced with the $NATRIGGER property expression that is set for a variable. The default value is `NO`.

For Oracle OLAP to permanently replace `NA` values for a variable with the valid $NATRIGGER property expression that is set for the variable, you must set both the TRIGGERSTOREOK option and the $STORETRIGGERVAL property for the variable to YES.

Notes

About the $NATRIGGER and STORETRIGGERVAL Properties

The TRIGGERSTOREOK option works in conjunction with the $NATRIGGER and $STORETRIGGERVAL properties of a variable.

Examples

Example 24–16 Replacing NA Values Temporarily

This example replaces the NA values in the cells of a variable temporarily. The following statements define a dimension with three values and define a variable dimensioned by the dimension. They add the $NATRIGGER property to the variable, then put a value in one cell of the variable and leave the other cells empty, so that their values are NA. Finally, they report the values in the cells of the variable.

```
DEFINE d1 INTEGER DIMENSION
MAINTAIN d1 ADD 3
DEFINE v1 DECIMAL <d1>
PROPERTY '$NATRIGGER' '500.0'
v1(d1 1) = 333.3

REPORT v1
```

The preceding statements produce the following output.

```
D1              V1
--------- ----------
        1     333.30
        2     500.00
        3     500.00
```

This statement deletes the $NATRIGGER property from the v1 variable.

```
CONSIDER v1
PROPERTY DELETE '$NATRIGGER'
REPORT v1
```

The preceding statements produce the following output.

```
D1              V1
--------- ----------
        1     333.30
        2         NA
        3         NA
```

Example 24–17 Replacing NA Values Permanently

The following statements add the $NATRIGGER property to the v1 variable that was defined in the previous example and set the TRIGGERSTOREOK option and the $STORETRIGGERVAL properties to YES. They then report the values in the cells of the variable.

```
CONSIDER v1
PROPERTY '$NATRIGGER' '800.0'
TRIGGERSTOREOK = YES
PROPERTY 'STORETRIGGERVAL' YES
REPORT v1
```

The preceding statements produce the following output.

```
D1                 V1
-------------- ----------
             1     333.30
             2     800.00
             3     800.00
```

The following statements delete the $NATRIGGER property from the `v1` variable and report the values in the cells of the variable.

```
CONSIDER v1
PROPERTY DELETE '$NATRIGGER'
REPORT v1
```

The preceding statements produce the following output.

```
D1                 V1
-------------- ----------
             1     333.30
             2     800.00
             3     800.00
```

TRIM

The TRIM function enables you to trim leading or trailing characters (or both) from a character string.

You can also trim leading characters using LTRIM and trailing characters using RTRIM.

Return Value

The data type of the string you are trimming (that is, *trim-source*).

Syntax

TRIM ([{{LEADING|TRAILING|<u>BOTH</u>} [*trim_character*])|*trim_character*} FROM] *trim_source*)

Arguments

trim-character
An expression that specifies the values to be trimmed. This text expression can be any of the text data types. When you do not specify a value, then the default value is a blank space and the function removes leading and trailing blank spaces.

LEADING
Specifies that the function removes any leading characters equal to *trim_character*.

TRAILING
Specifies that the function removes any trailing characters equal to *trim_character*.

BOTH
Specifies that the function removes leading and trailing characters equal to *trim_character*.

trim-source
An expression that is the value to be trimmed. This text expression can be any of the text data types.

TRUNC

The TRUNC function truncates either a number or a date and time value. Because the syntax of the TRUNC function is different depending on the whether it is being used for a number or a date and time value, two separate entries are provided:

- TRUNC (for dates and time)
- TRUNC (for numbers)

TRUNC (for dates and time)

When you specify a date and time value as an argument, the TRUNC function returns the date and time value truncated to a specified date format. When you do not specify a format, the date and time value is truncated to the nearest day.

Return Value

DATETIME

Syntax

TRUNC (*datetime_exp*, *fmt*)

Arguments

datetime-exp
An expression that identifies a date and time number.

fmt
A text expression that specifies one of the format models shown in Table 24–7, " Format Models for TRUNC for Dates and Time". A format model indicates how the date and time number should be truncated.

Table 24–7 Format Models for TRUNC for Dates and Time

Format Model	Description
CC SCC	One greater than the first two digits of a 4-digit year to indicate the next century. For example, 1900 becomes 2000. `S` prefixes BC dates with `-`.
D DAY DY	Starting day of the week (1 to 7). The day of the week that is number 1 is controlled by NLS_TERRITORY (See NLS Options).
DD	Day of month

Examples

Example 24–18 Truncating to the Nearest Year

When the value of the NLS_DATE_FORMAT option is `DD-MON-YY`, then this statement:

```
SHOW TRUNC ('27-OCT-92','YEAR')
```

returns this value:

```
01-JAN-92
```

TRUNC (for numbers)

When you specify a number as an argument, the TRUNC function truncates a number to a specified number of decimal places.

Return Value

DECIMAL

Syntax

TRUNC (*number*, *truncvalue*)

Arguments

number
The number to truncate. The value specified for *number* must be followed by a comma.

truncvalue
An INTEGER value that specifies the number of places to the right or left of the decimal point to which *number* should be truncated. When *truncvalue* is positive, digits to the right of the decimal point are truncated. When it is negative, digits to the left of the decimal point are truncated (that is, made zero). When *truncvalue* is omitted, *number* is truncated to `0` decimal places.

Examples

Example 24–19 Truncating to the Right of the Decimal Point

The following statement

```
SHOW TRUNC (15.79, 1)
```

returns this value

```
15.7
```

Example 24–20 Truncating to the Left of the Decimal Point

The following statement

```
SHOW TRUNC (15.79, -1)
```

returns this value

```
10
```

UNHIDE

The UNHIDE command unhides the text of a program that has been made invisible by using the HIDE command. To use UNHIDE, you must know the seed expression that was used with the HIDE command when the program was hidden.

Syntax

UNHIDE *prog-name seed-exp*

Arguments

prog-name
The name of a program whose text has been made invisible by using the HIDE command. Do not enclose the program name in quotes.

seed-exp
The single-line text expression that was used in the HIDE command when "prog-name" was hidden. The seed expression must be byte-for-byte the same value as you used in the HIDE command. Also, since the seed expression is case-sensitive, specify uppercase and lowercase characters carefully.

Notes

Forgetting the Seed Expression

When you want to use the UNHIDE command on a program but you have forgotten the seed expression, you can call Oracle OLAP Products Technical Support for help in solving your problem. Before calling, make a connection to Oracle OLAP from OLAP Worksheet, and in Oracle OLAP, attach the analytic workspace that contains the hidden program.

Examples

Example 24–21 Unhiding Program Text

The following example unhides the text of a program called `sales_rpt`. The seed expression `crystal` was used when the program was hidden using HIDE.

```
UNHIDE sales_rpt 'crystal'
```

UNIQUELINES

The UNIQUELINES function removes duplicate lines in a multiline TEXT value and sorts the lines in ascending order. The function returns a multiline TEXT value composed of the resulting lines.

Return Value

TEXT or NTEXT

Syntax

UNIQUELINES(*text-expression*)

Arguments

text-expression
A multiline text expression from which UNIQUELINES removes duplicate lines and in which it sorts the remaining lines. When you specify a TEXT expression, the return value is TEXT. When you specify an NTEXT expression, the return value is NTEXT.

Notes

Case Sensitivity

UNIQUELINES is case-sensitive when it checks for duplicates, and it compares all characters, including spaces.

Sort Order

UNIQUELINES sorts the lines in ascending order.

Examples

Example 24–22 Removing Duplicate Text Lines

In the following example, one line is removed from the value of `officelist`, and the lines are sorted.

The statement

```
SHOW officelist
```

produces the following output.

```
MIAMI
Providence
Miami
Baltimore
Saratoga
Baltimore
```

The statement

```
show uniquelines(officelist)
```

produces the following output.

```
Baltimore
Miami
MIAMI
Providence
Saratoga
```

UNRAVEL

The UNRAVEL function is used in conjunction with an assignment statement to copy the values of an expression into the cells of a variable when the dimensions of the expression are not the same as the dimensions of the variable.

An assignment statement created using an assignment statement assigns the values obtained from UNRAVEL by looping over the status of the dimensions of the target variable. The first dimension listed in the variable's definition varies the fastest. UNRAVEL obtains the values of the expression in the same way, looping over the status of the dimensions of the expression with the first dimension varying the fastest. You can alter the order in which UNRAVEL obtains its values by specifying the dimensions over which to loop.

Return Value

The data type returned by UNRAVEL is the data type of the values specified by the expression.

Syntax

UNRAVEL(*expression* [*dimension1*...])

Arguments

expression
The expression whose values are to be copied.

dimension
Specifies one or more dimensions over which to loop; the dimension specified first will vary fastest as the data is unraveled.

Specifying dimensions in UNRAVEL overrides the default looping order, as well as the extent of the unraveling of the expression. By default, unraveling extends through all the dimensions of the expression. However, when you specify some but not all the dimensions of the expression, any dimensions you have not specified do not unravel. Instead, the unraveled values will include only the first value of each of the omitted dimensions.

Notes

Performance Tip for Unraveling Variables Dimensioned by Composites

By default, when UNRAVEL loops over a composite, it sorts the composite values according to the current order of the values in the composite's base dimensions. The task of sorting requires some processing time, so when variables are large, performance can be affected. When your variable is very large, and you are more concerned about performance than about the order in which UNRAVEL output is produced, you can set the SORTCOMPOSITE option to `NO`.

Moving Worksheet Data

One common use of UNRAVEL is to move data from a worksheet to a variable, because the worksheet usually does not have the same dimensions as the variable. See Example 24–23, "Copying Data from a Worksheet to a Variable" on page 24-59.

Filling Extra Target Cells

When there are still more cells in the target for the assignment statement (created using an assignment statement) to fill after it has used the last value from the expression, UNRAVEL starts over at the first value again.

Setting Status

Since the order in which unraveled values are assigned depends on the current status of the dimensions of both the variable and the expression, be sure that the appropriate LIMIT commands have been given so that the cells match up correctly.

Assigning Data Values

See SET for information on how data values are assigned.

Examples

Example 24–23 Copying Data from a Worksheet to a Variable

In an analytic workspace, you have imported some product price data from a spreadsheet into a worksheet. You now want to transfer that data to a variable called newprice. You can produce a report of a worksheet, called pricedata, with these statements.

```
LIMIT wksrow TO 1 TO 6
LIMIT wkscol TO 1 2 3
REPORT pricedata
```

This is the report.

```
               -----------PRICEDATA------------
               -------------WKSCOL-------------
WKSROW              1          2          3
-------------- ---------- ---------- ----------
             1            Jan95      Jan96
             2 Tents          191.39     194.00
             3 Canoes         279.92     300.00
             4 Racquets        83.34      85.00
             5 Sportswear     107.90     110.00
             6 Footwear       183.18     195.00
```

As you can see, row 1 contains month labels, while column 1 contains product labels. The variable newprice is dimensioned by month and product, as shown in its definition.

```
DEFINE newprice VARIABLE DECIMAL <month product>
LD Wholesale Unit Selling Price
```

Even though the worksheet has different dimensions (wkscol and wksrow) than newprice, the data contained in it is well organized for transferring to the variable.

However, you do not want to take data from all the rows and columns in the worksheet, so limit wkscol and wksrow to the rows and columns that contain the price data itself.

```
LIMIT wkscol TO 2 3
LIMIT wksrow TO 2 TO 6
```

Also, you only want to set values into the variable newprice for January 1995 and January 1996. So first limit month to these values, then type the = command using UNRAVEL to move the values from the worksheet to the variable.

```
LIMIT month TO 'Jan95' 'Jan96'
newprice = UNRAVEL(pricedata)
```

You do not have to specify dimensions in the UNRAVEL function because wkscol is the fastest varying dimension. This means that both months will unravel for the first product, then both months for the second product. Since the fastest-varying dimension of the variable is `month`, SET assigns values to the variable in the same order.

A report of `newprice` looks like this.

```
               ------NEWPRICE-------
               --------MONTH--------
PRODUCT          Jan95      Jan96
-------------- ---------- ----------
Tents              191.39     194.00
Canoes             279.92     300.00
Racquets            83.34      85.00
Sportswear         107.90     110.00
Footwear           183.18     195.00
```

UPCASE

The UPCASE function converts all alphabetic characters in a text expression into uppercase. When you specify a TEXT expression, the return value is TEXT. When you specify an NTEXT expression, the return value is NTEXT.

Return Value

TEXT or NTEXT

Syntax

UPCASE(*text-expression*)

Arguments

text-expression
The text expression whose characters are to be converted.

Examples

Example 24–24 Converting Part of a Text Expression to Uppercase

Suppose you get some new data to add to a mailing list. In the existing mailing list, people's names have the first letter capitalized. In the new data, however, the whole name is in lowercase. You can use UPCASE to make the new data correspond to the current data with a statement similar to the following.

```
lastname = JOINCHARS(UPCASE(EXTCHARS(lastname, 1, 1)), -
           EXTCHARS(lastname, 2, NUMCHARS(lastname)))
```

UPDATE

The UPDATE command moves analytic workspace changes from a temporary area to the database table in which the workspace is stored. Typically, you use an UPDATE command when you are finished making changes in a workspace; however, you can also specify UPDATE commands periodically as you go along. Your changes are not saved until you execute a COMMIT command, either from Oracle OLAP or from SQL.

Syntax

UPDATE [MULTI [*aquired_objects*]] [*analytic_workspaces*]

Arguments

When you do not specify any parameters, the command updates all analytic workspaces that are attached in read/write non-exclusive and read/write exclusive modes and all acquired objects (that is, all acquired variables, relations, valuesets, and dimensions) in all analytic workspaces that are attached in multiwriter mode.

acquired_objects

A list of the names of acquired objects, separated by commas, in analytic workspaces attached in multiwriter mode. These objects can be any variable, relation, valueset, or dimension that you have acquired using an ACQUIRE statement.

> **Important:** you cannot update an object when it is dimensioned by an acquired and maintained dimension unless you update that dimension first.

workspaces

A list of names, separated by commas. of one or more workspaces attached in read/write or multiwriter mode.

Notes

Unsaved Changes

When you do not use the UPDATE and COMMIT commands, changes made to an analytic workspace during your session are discarded when you end your Oracle session.

Note: You can detach and reattach a workspace without losing updated changes, even though they are not committed. This is because the detaching and reattaching occur within a single database session

Automatic COMMIT

Many users execute DML statements using SQL*Plus or OLAP Worksheet. Both of these tools automatically execute a COMMIT statement when you end your session.

Triggering Program Execution When UPDATE Executes

Using the TRIGGER command, you can make the UPDATE command an event that automatically executes an OLAP DML program. See "Trigger Programs" on page 1-14 for more information

Shared Workspaces

When you have attached a shared workspace and another user has read/write access, that user's UPDATE and COMMIT commands do not affect your view of the workspace. Your view of the data remains the same as when you attached the workspace. When you want access to the changes, you can detach the workspace and reattach it.

Effect of the ROLLBACK Command

The OLAP DML does not provide a way to issue a SQL `ROLLBACK` statement; however, you could execute one in your session from outside Oracle OLAP (for example, through PL/SQL). When a `ROLLBACK` statement is executed in your session, Oracle OLAP checks to see whether there are uncommitted updates in an attached workspace.

- When there are uncommitted updates (that is, you have made changes and executed an UPDATE command, but you have not subsequently executed a COMMIT command), then Oracle OLAP discards your changes and detaches the workspace.

- When you have no uncommitted updates, then Oracle OLAP takes no action in response to the `ROLLBACK` command. This means that, when you have not issued an UPDATE command since your last COMMIT command, Oracle OLAP takes no action and all your changes remain in the workspace during your session.

When you rollback to a savepoint and there are uncommitted updates that occurred subsequent to the savepoint, Oracle OLAP discards those updates and detaches the workspace. Uncommitted updates that occurred before the savepoint remain in the workspace, and you can see them when you reattach the workspace in the same session.

Examples

Example 24–25 Saving Analytic Workspace Changes

The following statement moves changes in the current workspace session to the database table in which the workspace is stored.

```
UPDATE
```

In order to save the changes in the database, the UPDATE command must be followed by a COMMIT command.

USERID

(Read-only) The USERID option holds the user ID for the current Oracle Database session.

Data type

TEXT

Syntax

USERID

Notes

USERID Option and SYSINFO(USER) Function

The value of USERID is also the value that SYSINFO(USER) returns.

Examples

Example 24–26 Displaying the Session User ID

This statement displays the Oracle user ID associated with the current session.

```
SHOW USERID
```

USETRIGGERS

The USETRIGGERS option determines if a TRIGGER_DEFINE, TRIGGER_AFTER_UPDATE, or TRIGGER_BEFORE_UPDATE program, or any programs identified by the TRIGGER command as triggers execute.

> **Tip:** Oracle OLAP does not support recursive triggers. Set the USETRIGGERS option to `NO` before you issue the same DML statement *within* a trigger program that triggered the program itself. For example, assume that you have written a TRIGGER_DEFINE program. Within the TRIGGER_DEFINE program, you must set the USETRIGGERS option to `NO` before you issue a DEFINE statement

Data type

BOOLEAN

Syntax

USETRIGGERS = {NO|<u>YES</u>}

Arguments

YES
Trigger programs execute. (Default)

NO
Trigger programs do *not* execute.

Examples

Example 24–27 Changing USETRIGGERS to NO

Assume you have just created a new analytic workspace. As illustrated in the following statement, the default value of the USETRIGGERS option is `YES`, but you can set the option to `NO` at any time.

```
SHOW USETRIGGERS
yes


USETRIGGERS = NO
SHOW USETRIGGERS
no
```

VALSPERPAGE

The VALSPERPAGE program calculates the maximum number of values for a variable of a given width that will fit on one page. Pages are units of storage in the workspace.

Syntax

VALSPERPAGE(*n*)

Arguments

n

An INTEGER expression specifying the width of a variable in bytes. This value should be between `1` and `4000`. When you specify a value greater than `4000` or less than `1`, the result is `NA`.

Notes

Large Variables

Oracle OLAP lets you create very large, multidimensional variables. Theoretically, a variable can contain up to `2**63` cells, although this maximum is subject to memory constraints and other factors specific to your system.

Related Statements

AW function and DEFINE VARIABLE command.

Examples

Example 24–28 Calculating the Number of Cells in a Page

The following statement calculates the maximum number of cells available in a single page for a variable with an INTEGER data type. The default width of integers in Oracle OLAP is `4` bytes.

```
SHOW VALSPERPAGE(4)
```

The output of this statement is

```
992
```

VALUES

The VALUES function returns the default status list or the current status list of a dimension or dimension surrogate, or it returns the values in a valueset. VALUES returns a multiline text value that contains one dimension value on a line.

Return Value

TEXT

Syntax

VALUES(*dimension* [*keyword*] [INTEGER])

Arguments

dimension
A text expression whose value is the name of a dimension, dimension surrogate, or valueset.

keyword
One of the following keywords that specify whether you want the current status list or the default status list for a dimension or a surrogate:

- **NOSTATUS** which indicates that VALUES should return the default status list of a dimension or dimension surrogate rather than its current status list.
- **STATUS** which indicates that VALUES should return the current status list of a dimension or dimension surrogate (Default).

These keywords do not affect valuesets. For a valueset, VALUES returns all the values in that valueset whether you specify `NOSTATUS`, `STATUS`, or nothing.

INTEGER
When you use the INTEGER keyword, the function returns the position numbers of the dimension or dimension surrogate values rather than the values. When you use INTEGER with a valueset, the function returns the position numbers of the values in the existing dimension, not in the valueset.

Notes

Comparing VALUES to CHARLIST

The VALUES function is very similar to the CHARLIST function. `VALUES(MONTH)` returns the same list as `CHARLIST(MONTH)`.

The main differences are:

- For dimensions, the NOSTATUS keyword of VALUES lets you use the default status without first limiting the dimension values to `ALL`.
- The VALUES function lets you use a text expression to specify the dimension or valueset name. See Example 24–31, "VALUES with Text Variables" on page 24-72.

Do Not Use with Composites

Because composites do not have status, you cannot use the VALUES function with a composite. When you attempt to do so, Oracle OLAP displays an error message.

Special Considerations for an Ampersand (&)

Under certain circumstances, an ampersand (`&`) that is intended to be a character in a dimension value name will be interpreted as ampersand substitution. When this happens, Oracle OLAP generates an error message.

This happens because Oracle OLAP recognizes special characters in dimension value names with when they are used in tuples in text expressions. For example, you can include spaces, such as naming a dimension value `New York` instead of `NewYork`. When you have dimension values that include ampersands in their names, refer to Example 24–32, "Workaround for Dimension Value Names Including an Ampersand" on page 24-73.

Examples

Example 24–29 Listing the Values of a Valueset

Suppose an analytic workspace contains a valueset called `monthset` that has the values `Jan95`, `May95`, and `Dec95`. You can use VALUES to list the values in that valueset.

The following statement

```
SHOW VALUES(monthset)
```

produces this output.

```
Jan95
May95
Dec95
```

Example 24–30 Listing Position Numbers of a Dimension

You can use VALUES to list the position numbers instead of the actual values in a dimension or valueset. In this example, because you are using the INTEGER keyword with a valueset instead of a dimension, the function returns the position numbers of the values in the `month` dimension.

The following statement

```
SHOW VALUES(monthset INTEGER)
```

produces this output.

```
61
65
72
```

Therefore, the value `Jan95` is shown as the 61st value in the `month` dimension, `May95` as the 65th value, and `Dec95` as the 72nd value, although they are the first, second, and third values in `monthset`.

Example 24–31 VALUES with Text Variables

This example shows how to assign a dimension name to a text variable and use the text variable in the VALUES function instead of the variable name itself. When the variable `textvar` has the value `district`, `VALUES(textvar)` returns a list of `district` values.

The following statements

```
textvar = 'district'
SHOW VALUES(textvar)
```

produce this output.

```
Boston
Atlanta
Chicago
Dallas
Denver
Seattle
```

To list the values of `district` using the CHARLIST function rather than VALUES, you must use an ampersand.

```
SHOW CHARLIST(&textvar)
```

Because ampersands in a program can degrade performance, you should use VALUES rather than CHARLIST in such cases.

Example 24–32 Workaround for Dimension Value Names Including an Ampersand

When a dimension value name contains an ampersand (`&`) as one of its characters, and when that dimension is a base dimension of a conjoint dimension, then a text expression that contains the names of dimension values in a tuple can generate an error in certain circumstances. This example shows how to avoid this error.

Suppose you use the following statements to define two dimensions.

```
DEFINE prod DIMENSION TEXT
DEFINE geog DIMENSION TEXT
```

Next, you use the following statements to define two conjoint dimensions.

```
DEFINE conj1 DIMENSION <prod geog>
DEFINE conj2 DIMENSION <prod geog>
```

The following statements add dimension values to the `prod` and `geog` dimensions.

```
MAINTAIN prod ADD 'prod1' 'prod&val2'
MAINTAIN geog ADD 'geog1' 'geog&val2'
```

The following statements add tuples (combinations of dimension values) to the CONJ1 conjoint dimension.

```
MAINTAIN conj1 ADD <'prod1' 'geog1'>
MAINTAIN conj1 ADD <'prod&val2' 'geog1'>
```

Now, suppose you want to use the VALUE function with the MAINTAIN command to add those same tuples to the `conj2` conjoint dimension. When you attempt to use the following statement, it will generate an error message.

```
MAINTAIN conj2 ADD VALUES(conj1)
ERROR: (MXMSERR) val2 does not exist in any attached workspace.
```

This error occurs because the ampersand in the dimension value name `prod&val2` is interpreted as an attempt at ampersand substitution.

Instead of using the preceding MAINTAIN command, you can use the following statement to add the tuples to the CONJ2 conjoint dimension.

```
MAINTAIN conj2 MERGE < KEY(conj1 prod) KEY(conj1 geog) >
```

VARCACHE

The VARCACHE option specifies whether Oracle OLAP stores or caches all variable data that is the result of the execution of an AGGREGATE function or $NATRIGGER expression.

Important: The value of the VARCACHE option is only one factor that Oracle OLAP uses to determine whether variable data computed when the AGGREGATE function or $NATRIGGER property executes is stored or cached. For a discussion of the other factors and their interrelationship, see"How Oracle OLAP Determines Whether to Store or Cache Aggregated Data" on page 6-23 and "How Oracle OLAP Determines Whether to Store or Cache Results of $NATRIGGER" on page 6-21.

Syntax

VARCACHE = {VARIABLE | SESSION | <u>NONE</u>}

Arguments

VARIABLE
Specifies that Oracle OLAP stores the data in the variable in the database. When you specify this option, the results of the calculation are permanently stored in the variable when the analytic workspace is updated and committed.

SESSION
Specifies that Oracle OLAP caches the calculated data in the session cache (See "What is an Oracle OLAP Session Cache?" on page 21-54). When you specify this option, the results of the calculation are ignored during updates and commits and are discarded at the end of the session.

Important: When SESSCACHE is set to `NO`, Oracle OLAP does not cache the data even when you specify `SESSION`. In this case, specifying `SESSION` is the same as specifying `NONE`.

NONE
For data that is calculated on the fly using the AGGREGATE function, specifies that Oracle OLAP calculates the data each time the AGGREGATE function executes; Oracle OLAP does not store or cache the data calculated by the AGGREGATE function

Notes

The VARCACHE Option Can Affect All Variables

When you set the VARCACHE option, its setting can affect all variables. When you have not set the `$VARCACHE` property on a variable and there is no CACHE command in the aggmaps that you use with the AGGREGATE function to calculate data on the fly, then it is the VARCACHE option that determines how or if that data will be stored.

VARIABLE

Within an OLAP DML program, the VARIABLE command declares a local variable or valueset for use within that program. A local variable cannot have any dimensions and exists only while the program is running.

Syntax

VARIABLE *name* {*datatype*|*dimension*|VALUESET *dim*}

Arguments

name

The name for the local variable or valueset. When you use the same name as an existing analytic workspace object, the local variable or valueset takes precedence over the analytic workspace object. After you assign a value to the variable or valueset, its value will be available within the program where the VARIABLE command occurs. You name a variable or valueset according to the rules for naming analytic workspace objects (see the DEFINE command).

datatype

The data type of the variable, which indicates the kind of data to be stored. You can specify any of the data types that are listed and described in the DEFINE VARIABLE entry. Also, when you want to the program to be able to receive an argument without converting it to a specific data type, you can also specify `WORKSHEET` for the data type.

dimension

Indicates that *name* is a relation variable, which holds a single value of the specified dimension. The variable can hold a value of the dimension or a position (integer) of the specified dimension. Assigning a value that does not currently exist in the dimension causes an error.

VALUESET *dim*

Indicates that *name* is a valueset. Dim specifies the dimension for which the valueset holds values.

Notes

Persistence of a Local Variable

A local variable or valueset exists only while the program that specified it is running. When the program terminates, the variable or valueset ceases to exist and its value is lost. A program can terminate when a RETURN statement, SIGNAL command, or the last line of the program executes. When the program calls another program, the original program is temporarily suspended and the variable or valueset does exist when the called program ends and control returns to the original program. A program that calls itself recursively has separate local variable or valuesets for each running copy of the program.

Declarations at the Start Of A Program

You must specify all your local variables or valuesets at the beginning of a program, before any executable statements.

Initial Value

The value of a local variable or valueset is initially `NA`.

Duplicate Names

When you give a local variable or valueset the same name as an analytic workspace object, Oracle OLAP assumes you are referring to the local variable or valueset within the program. The analytic workspace object has priority only when the statement requires an analytic workspace object as an argument.

Although the OBJ and EXISTS functions expect an analytic workspace object as an argument, you can use a local text variable or valueset to specify the name of an object.

Formulas and Models

You cannot use local variables or valuesets in a formula or model.

EXPORT and IMPORT Commands

In a program, you can use the EXPORT (to EIF) command to store the value of a local variable or valueset in an EIF file. You must use the AS keyword to give the variable or valueset an analytic workspace object name. The name can be the same as the name of the local variable or valueset. When you use IMPORT (from EIF) to retrieve the value, it is stored as an analytic workspace object. You cannot import the value into a local variable or valueset.

Examples

Example 24–33 Saving a File Unit Number

Suppose you want to write a program to read data from an input file with Data Reader statements. First you need to open the file and save the value of the file unit number assigned to it. At the beginning of the program you can specify a local variable called `unit` to hold the file unit number.

```
DEFINE read.file PROGRAM
LD Read monthly sales data into the analytic workspace
PROGRAM
VARIABLE unit INTEGER
TRAP ON error
unit = FILEOPEN('sales.data' READ)
...
```

Example 24–34 Returning a Dimension Value from a Program

Suppose you want to write a program that analyzes sales for various districts and returns the name of the district in which sales were highest. For the purpose of analysis, the program defines a local variable to hold the district name. When the program ends, it returns the value of the local variable.

```
DEFINE highsales PROGRAM
PROGRAM
VARIABLE districtname district
... "(statements that find the highest district)
RETURN districtname
END
```

VINTSCHED

The VINTSCHED function calculates the interest portion of the payments on a series of variable-rate installment loans that are paid off over a specified number of time periods. For each time period, you specify the initial amount of the loans incurred in that time period and the interest rate that will be charged in that time period for each new or outstanding loan.

Return Value

DECIMAL

Syntax

VINTSCHED(*loans*, *rates*, *n*, [*time-dimension*])

Arguments

loans

A numeric expression that contains the initial amounts of the loans. When *loans* does not have a time dimension, or when *loans* is dimensioned by more than one time dimension, the *time-dimension* argument is required.

rates

A numeric expression that contains the interest rates charged for *loans.* When *rates* is a dimensioned variable, it can be dimensioned by any dimension, including a different time dimension. When *rates* is dimensioned by a time dimension, you specify the interest rate in each time period that will apply to the loans incurred or outstanding in that period. The interest rates are expressed as decimals; for example, a 5 percent rate is expressed as `.05`.

n

A numeric expression that specifies the number of payments required to pay off the loans in the series. The *n* expression can be dimensioned, but it cannot be dimensioned by the time dimension argument. One payment is made in each time period of the time dimension by which *loans* is dimensioned or in each time period of the dimension specified in the *time-dimension* argument. For example, one payment a month is made when *loans* is dimensioned by `month`.

time-dimension
The name of the dimension along which the interest payments are calculated. When *loans* has a dimension of type of DAY, WEEK, MONTH, QUARTER, or YEAR, the *time-dimension* argument is optional, unless *loans* has more than one dimension of these types.

Notes

The Result Dimensions of the Result

The result returned by the VINTSCHED function is dimensioned by the union of all the dimensions of *loans* and *rates* and the dimension that is used as the *time-dimension* argument.

Time Period Results

VINTSCHED calculates the result for a given time period as the sum of the interest due on each loan that is incurred or outstanding in that period.

NA Mismatch Error

When *loans* has a value other than NA and the corresponding value of *rates* is NA, an error occurs.

NASKIP Option

VINTSCHED is affected by the NASKIP option. When NASKIP is set to YES (the default), and a loan value is NA for the affected time period, the result returned by VINTSCHED depends on whether the corresponding interest rate has a value of NA or a value other than NA. Table 24–8, " How NASKIP Affects the Results When a Loan or Rate Value is NA for a Given Time Period" illustrates how NASKIP affects the results when a loan or rate value is NA for a given time period.

Table 24–8 How NASKIP Affects the Results When a Loan or Rate Value is NA for a Given Time Period

Loan Value	Rate Value	Result when NASKIP = YES	Result when NASKIP = NO
Non-NA	NA	Error	Error
NA	Non-NA	Interest values (NA loan value is treated as zero)	NA for affected time periods
NA	NA	NA for affected time periods	NA for affected time periods

As an example, suppose a loan expression and a corresponding interest expression both have NA values for 1997, but both have values other than NA for succeeding years. When the number of payments is 3, VINTSCHED returns NA for 1997, 1996, and 1995. For 1997, VINTSCHED returns the interest portion of the payment due for loans incurred in 1995, 1996, and 1997.

Time Dimensions

The VINTSCHED calculation begins with the first value of the time dimension, regardless of how the status of that dimension may be limited. For example, suppose *loans* is dimensioned by `year`, and the values of `year` range from `Yr95` to `Yr99`. The calculation always begins with `Yr95`, even when you limit the status of `year` so that it does not include `Yr95`.

However, when *loans* is not dimensioned by the time dimension, the VINTSCHED calculation begins with the first value in the current status of the time dimension. For example, suppose *loans* is not dimensioned by `year`, but `year` is specified as *time-dimension*. When the status of `year` is limited to `Yr97` to `Yr99`, the calculation begins with `Yr97` instead of `Yr95`.

Related Functions

The FINTSCHED function, which calculates the interest portion of the payments on a series of fixed-rate loans, and the VPMTSCHED and FPMTSCHED functions, which calculate the payment schedules (principal plus interest) for variable-rate and fixed-rate loans.

Examples

Example 24–35 Using VINTSCHED

The following statements create two variables called `loans` and `rates`.

```
DEFINE loans DECIMAL <year>
DEFINE rates DECIMAL <year>
```

Suppose you assign the following values to the variables `loans` and `rates`.

```
YEAR               LOANS      RATES
-------------- ---------- ----------
Yr95               100.00       0.05
Yr96               200.00       0.06
Yr97               300.00       0.07
Yr98                 0.00       0.07
Yr99                 0.00       0.07
```

For each year, `loans` contains the initial value of the variable-rate loan incurred during that year. For each year, the value of `rates` is the interest rate that will be charged for that year on any loans incurred or outstanding in that year.

The following statement specifies that each loan is to be paid off in three payments, calculates the interest portion of the payments on the loans,

```
REPORT W 20 HEADING 'Payment' VINTSCHED(loans, rates, 3, year)
```

and produces the following report.

```
YEAR                        Payment
-------------- --------------------
Yr95                           5.00
Yr96                          16.10
Yr97                          33.06
Yr98                          19.43
Yr99                           7.48
```

The interest payment for 1995 is interest on the loan of $100 incurred in 1995, at 5 percent. The interest payment for 1996 is the sum of the interest on the remaining principal of the 1995 loan, plus interest on the loan of $200 incurred in 1996; the interest rate for both loans is 6 percent. The 1997 interest payment is the sum of the interest on the remaining principal of the 1995 loan, interest on the remaining principal of the 1996 loan, and interest on the loan of $300 incurred in 1997; the interest rate for all three loans is 7 percent. Since the 1995 loan is paid off in 1997, the payment for 1998 represents 7 percent interest on the remaining principal of the 1996 and 1997 loans. In 1999, the interest payment is on the remaining principal of the 1997 loan.

VNF

The VNF command assigns a value name format (VNF) to the definition of a dimension with a type of DAY, WEEK, MONTH, QUARTER, or YEAR. A VNF is a template that controls the input and display format for values of DAY, WEEK, MONTH, QUARTER, and YEAR dimensions. The template can include format specifications for any of the components that identify a time period (day, month, calendar year, fiscal year, and period within a fiscal year).

Important: You can only use this function with dimensions of type DAY, WEEK, MONTH, QUARTER, or YEAR.You can*not* use this function for time dimensions that are implemented as hierarchical dimensions of type TEXT.

In order to assign a VNF to a definition, the definition must be the one most recently defined or considered during the current session. When it is not, you must first use a CONSIDER command to make it the current definition.

Syntax

VNF [*template*]

Arguments

template

A text expression that specifies the format for entering and displaying the values of the current dimension. When *template* is omitted, any existing VNF for the current definition is deleted and the default VNF is used (see Table 24–15, " Default VNFs for DAY, WEEK, MONTH, QUARTER and YEAR Dimensions").

Note: When you enter a dimension value that does not conform to the VNF, Oracle OLAP attempts to interpret the value as a date. See "Entering Dimension Values as Dates" on page 24-91

A template contains a code for each component that you use to describe a time period in the current dimension. The code for each component must be preceded by a left angle bracket and followed by a right angle bracket. Basic information about coding a template is provided in Table 24–9, " Basic Codes for Components in VNF Templates", Table 24–10, " Component Combinations Allowed in VNF Templates", and Table 24–11, " Format Styles for Day Available in VNF Templates".

Table 24–9, " Basic Codes for Components in VNF Templates" lists the basic codes for the components of time periods. It uses a sample dimension called MYQTR, which is a QUARTER dimension that ends in June. The examples are from the quarter July 1, 1995 through September 30, 1995. The period code (`P`) specifies the numeric position of a time period within a fiscal year. You can use the `P` code with any dimension, but only when you use it along with the `FF` or `FFB` code. The `B` code specifies the beginning period.

Table 24–9 Basic Codes for Components in VNF Templates

Code	Meaning	Sample Values

`<D>`
`<M>`
`<YY>`
	Day of the month on which the period ends
	Month in which the period ends
	Calendar year in which the period ends
		`30`
		`9`
		`95`

`<FF>`	Fiscal year that contains the period; the fiscal year is identified by the calendar year in which the fiscal year ends	`96`

`<DB>`
`<MB>`
`<YYB>`
	Day of the month on which the period begins
	Month in which the period begins
	Calendar year in which the period begins
		`1`
		`7`
		`95`

`<FFB>`	Fiscal year that contains the period; the fiscal year is identified by the calendar year in which the fiscal year begins	`95`
`<P>`	The period's numeric position within the fiscal year	`1`
`<NAME>`	Name of the dimension	`MYQTR`

Table 24–10, " Component Combinations Allowed in VNF Templates" lists the component combinations you can combine in a VNF for each type of dimensions of type DAY, WEEK, MONTH, QUARTER, or YEAR. Notice that you can use the fiscal year codes (`FF` or `FFB`) in a template for any dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR. However, the fiscal year codes have a special meaning for WEEK dimensions and for phased MONTH, QUARTER, and YEAR

dimensions. For other dimensions, the fiscal year is identical to the calendar year. See "Fiscal Years for a Dimension of Type WEEK" on page 24-89, "Fiscal Years for Dimensions of Type MONTH, QUARTER, or YEAR" on page 24-90, and "Fiscal Years for Dimensions of Type DAY" on page 24-90.

Table 24–10 Component Combinations Allowed in VNF Templates

Type of Dimension	Component Combinations	Sample Values
DAY, WEEK, MONTH, QUARTER, YEAR	`<D> <M> <YY>`	`31 3 96`
	`<DB> <MB> <YYB>`	`1 4 95`
	`<P> <FF>`	`1 96`
	`<P> <FFB>`	`1 95`
MONTH, QUARTER, YEAR	`<M> <YY>`	`3 96`
	`<MB> <YYB>`	`4 95`
	`<M> <FF>`	`3 96`
	`<M> <FFB>`	`3 95`
	`<MB> <FF>`	`4 96`
	`<MB <FFB>`	`4 95`
YEAR	`<YY>`	`96`
	`<FF>`	`96`
	`<FFB>`	`95`

Notice that in place of the basic codes listed in Table 24–10, " Component Combinations Allowed in VNF Templates", you can substitute any of the format styles listed in Table 24–11, " Format Styles for Day Available in VNF Templates". You can also include the `<NAME>` component with any of the component combinations listed in Table 24–10.

You cannot specify a template that includes too few or too many components. The VNF must allow you to input dimension values without ambiguity. See "Coding VNFs to Prevent Ambiguity" on page 24-90.

However, if you include only the component combinations that are allowed for a particular type of dimension, and if the VNF permits unambiguous interpretation of input, you have considerable flexibility in specifying a VNF template:

- You can specify the components in any order.
- You can include text before, after, and between the components.

In place of the basic codes for the day, month, calendar year, fiscal year, and period that were listed in Table 24–10, " Component Combinations Allowed in VNF Templates", you can substitute the format styles listed in Table 24–11, " Format Styles for Day Available in VNF Templates", Table 24–12, " Format Styles for Month Available in VNF Templates", Table 24–13, " Format Styles for Year Available in VNF Templates", and Table 24–14, " Format Styles for Period Available in VNF Templates".

Table 24–11 Format Styles for Day Available in VNF Templates

Format	Meaning	Jan 3, 1995	Nov 12, 2051
`<D>`	One digit or two digits	`3`	`12`
`<DD>`	Two digits	`03`	`12`
`<DS>`	Space-padded, two digits	`3`	`12`

Table 24–12 Format Styles for Month Available in VNF Templates

Format	Meaning	Jan 3, 1995	Nov 12, 2051
`<M>`	One digit or two digits	`1`	`11`
`<MM>`	Two digits	`01`	`11`
`<MS>`	Space-padded, two digits	`1`	`11`
`<MTXT>`	First three letters, uppercase	`JAN`	`NOV`
`<MTXTL>`	First three letters, lowercase	`jan`	`nov`
`<MTEXT>`	Full name, uppercase	`JANUARY`	`NOVEMBER`
`<MTEXTL>`	Full name, lowercase	`january`	`november`

Note that for `MTXT`, `MTXTL`, `MTEXT`, and `MTEXTL`, the actual value displayed depends on the value specified for the MONTHNAMES option:

- For `MTXT` and `MTEXT`, when the name in the MONTHNAMES option is all lowercase, the entire name is converted to uppercase. Otherwise, the first letter is converted to uppercase and the second and subsequent letters remain in their original case.
- For `MTXTL` and `MTEXTL`, when the name in the MONTHNAMES option is all uppercase, the entire name is converted to lowercase. Otherwise the first letter is converted to lowercase and the second and subsequent letters remain in their original case.

Table 24–13 Format Styles for Year Available in VNF Templates

Format	Meaning	Jan 3, 1995	Nov 12, 2051
`<YY>`	Two digits or four digits	`95`	`2051`
`<YYYY>`	Four digits	`1995`	`2051`
`<FF>`	Two digits or four digits	`95`	`2051`
`<FFFF>`	Four digits	`1995`	`2051`

Table 24–14 Format Styles for Period Available in VNF Templates

Format	Meaning	Jan 3, 1995	Nov 12, 2051
`<P>`	One, two, or three digits	`3`	`316`
`<PP>`	Two or three digits	`03`	`316`
`<PS>`	Space-padded, two or three digits	`3`	`316`
`<PPP>`	Three digits	`003`	`316`
`<PPS>`	Space-padded, three digits	`3`	`316`

When you do not provide a VNF for DAY, WEEK, MONTH, QUARTER, and YEAR dimensions, Oracle OLAP uses a default VNF that is suited to the type of dimension ash shown in Table 24–15, " Default VNFs for DAY, WEEK, MONTH, QUARTER and YEAR Dimensions".

You can append the B code to any of the format styles except.

Table 24–15 Default VNFs for DAY, WEEK, MONTH, QUARTER and YEAR Dimensions

Type of Dimension	Default VNF	Example
DAY	<DD><MTXT><YY>	01JAN95
WEEK	W<P>.<FF>	W1.95
Multiple WEEK	<NAME><P>.<FF>	MYWEEK1.95
MONTH	<MTXT><YY>	JAN95
Multiple MONTH	<NAME><P>.<FF>	MYMONTH1.95
QUARTER	Q<P>.<FF>	Q1.95
YEAR	YR<YY>	YR95

Notes

Discarding a VNF

When you want to discard a VNF for a dimension and return to using the default VNF, use the CONSIDER command to make the dimension's definition the current one, and then use a VNF command with no argument.

Angle Brackets

To include an angle bracket as additional text in a template, specify two additional angle brackets for each angle bracket to be included as text (for example, to display the entire value in angle brackets, specify `<<<D> <M> <YY>>>`).

Month Names

The names used in the month component for the `MTXT`, `MTXTL`, `MTEXT`, and `MTEXTL` formats are drawn from the current setting of the MONTHNAMES option.

Fiscal Year Codes

You can use a fiscal year code (`FF` or `FFB`) in a template for any dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR.

Fiscal Years for a Dimension of Type WEEK

For a dimension of type WEEK, a fiscal year starts on the beginning date of the first period (single-week or multiple-week) that ends in a new calendar year. The fiscal year ends on the final date of the final period that is wholly contained in the calendar year.

This definition holds true, regardless of any beginning or ending *date* you specify for a WEEK dimension when you define it. However, the fiscal year does take into account the beginning or ending *day of the week* that you specify (either as a day of the week or as a date).

For example, suppose you define a dimension of type WEEK, named `myweek`, with single-week periods ending on June 2, 1995 (a Friday). The fiscal year that contains June 2, 1995 begins on December 31, 1994 (a Saturday) and ends on December 29, 1995 (a Friday). When the VNF for `myweek` has the `FF` code, this fiscal year is identified as 1995. When the VNF has the `FFB` code, the fiscal year is identified as 1994.

Fiscal Years for Dimensions of Type MONTH, QUARTER, or YEAR

For a dimension of type MONTH, QUERTER, or YEAR with no beginning or ending phase, the fiscal year is identical to the calendar year.

For a MONTH, QUARTER, or YEAR dimension with a beginning or ending phase, each fiscal year for that dimension begins with the beginning month of the phase and ends with the ending month of the phase.

For example, assume you define a dimension of type MONTH, `mymonth`, with four-month periods ending in March, each fiscal year begins on April 1 and ends on March 31. When you use the `FF` code in a VNF for MYMONTH, the fiscal year that starts on April 1, 1995 and ends on March 31, 1996 is identified as 1996. When you use the `FFB` code, this fiscal year is identified as 1995.

Fiscal Years for Dimensions of Type DAY

For a dimension of type DAY, the fiscal year is identical to the calendar year.

Out-of-Range Years

When a VNF specifies a `YY`, `YYB`, `FF`, or `FFB` format, and a year outside the range of 1950 to 2049 is to be displayed, the year is displayed in four digits. You must also supply all four digits when you enter the year as input.

Coding VNFs to Prevent Ambiguity

A VNF template must allow you to input dimension values unambiguously. To prevent ambiguity, you must observe the following restrictions when you code a VNF template:

- You cannot place a letter (either in a component code or in literal text) immediately after a text component of unspecified length (for example, `<MTEXT>`, which specifies a full month name of any length).
- You cannot place a digit (either in a component code or in literal text) immediately after a numeric component of unspecified length (for example, <M>, which can be one digit or two digits, or `<YY>`, which can be two digits or four digits).

Coding VNFs for Model Dimensions

When you define a model that contains equations based on a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR, the VNF for the that dimension must specify dimension values with these format characteristics: the value must start with a letter, and it can contain only letters, digits, underscores, and periods.

Entering Dimension Values

Once you have assigned a VNF to a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR, you cannot use the default VNF for entering values for that dimension. You must enter values in the format of your VNF or as dates.

Entering Dimension Values in VNF Format

When you enter dimension values in a VNF format, you have the following flexibility:

- Letters (either in a component or in literal text) can be either uppercase or lowercase, rather than matching the exact capitalization indicated by the VNF.
- When the template specifies `<MTXT>` or `<MTXTL>`, which indicate the first three letters of the month name, you can include as much of the month name as you want, from the first three letters to the full month name. When the template specifies `<MTEXT>` or `<MTEXTL>`, which indicate a month name of indeterminate length, you can include as much of the name as you want, from the first letter to the full month name. In all cases, however, you must provide enough letters to uniquely match a name in the MONTHNAMES option. For example, to distinguish April from August, you must type at least the first two letters of these names.
- You can include as many or as few spaces as you want between components or between text elements in a dimension value.
- When the template contains date components that are not essential for identifying a time period for a particular dimension, you can specify any date that falls within the desired time period. For example, the `<DD>` component of the template `<DD><MTXT><YY>` is not essential for identifying a period in a MONTH dimension. Therefore, for June 1995 you can specify any date from `01JUN95` through `30JUN95`.

Entering Dimension Values as Dates

When you enter a value of a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR as a date, you can use any of the input styles listed in the DATEORDER entry. When you specify a full date, Oracle OLAP uses the DATEORDER option to resolve any ambiguities. However, you need to specify only the date components that are relevant for the type of dimension you are using:

- For a DAY or WEEK dimension, you must enter all the components (day, month, and year).
- For a MONTH or QUARTER dimension, you only need to enter the month and year components. When you enter an ambiguous value, such as `'0106'`, Oracle OLAP uses the first two characters of the DATEORDER option to resolve the ambiguity. Therefore, the DATEORDER option must be `MYD` or `YMD` in this situation.
- For a YEAR dimension, you only need to enter the year.

Overriding a VNF

For additional flexibility in displaying the values of a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR, you can override the dimension's VNF (and the default VNF when the dimension has no VNF of its own) by using the CONVERT function with a VNF argument.

The VNF argument to CONVERT enables you to include all the template codes that are permitted in the template for the VNF command, but it does not prevent you from specifying too few components or more components than are necessary for identifying a value. In addition, the VNF argument enables you to use additional codes that are not allowed in the VNF template.

Examples

Example 24–36 Assigning a VNF for a Dimension of Type MONTH

The following statements provide a VNF for the existing dimension of type MONTH named `month`.

```
CONSIDER month
VNF <mtextl>, <yyyy>
```

Example 24–37 Adding Values to a Dimension of Type MONTHi

The following statements add dimension values in the style of the new VNF, using just enough letters to distinguish the month names rather than the full names that the `<MTEXTL>` code in the VNF specifies.

```
MAINTAIN month ADD 'JA, 1995' 'MAR, 1995'
Limit month TO LAST 3
REPORT month
```

These statements produce the following output.

```
MONTH
--------------
January, 1995
February, 1995
March, 1995
```

Note that Oracle OLAP automatically adds the time periods between the ones you specify in the MAINTAIN command.

Example 24–38 Assigning a VNF for WEEK

The following statements define a dimension of type WEEK named `week`, add a VNF to the `week` definition, and add values to the `week` dimension.

```
DEFINE week DIMENSION WEEK
VNF Week <p>.<ff>
MAINTAIN week ADD '01JAN95' '30JAN95'
REPORT week
```

These statements produce the following output.

```
WEEK
--------------
Week 1.95
Week 2.95
Week 3.95
Week 4.95
Week 5.95
```

When you use the MAINTAIN command to add values to the `week` dimension, you can specify the new values as dates rather than as values that conform to the VNF. However, the VNF is used for displaying output in the desired format.

VPMTSCHED

The VPMTSCHED function calculates a payment schedule (principal plus interest) for paying off a series of variable-rate installment loans over a specified number of time periods. For each time period, you specify the initial amount of the loans incurred in that time period and the interest rate that will be charged in that time period for each new or outstanding loan.

Return Value

DECIMAL

Syntax

VPMTSCHED(*loans*, *rates*, *n*, [*time-dimension*])

Arguments

loans
A numeric expression that contains the initial amounts of the loans. When *loans* does not have a time dimension, or when *loans* is dimensioned by more than one time dimension, the *time-dimension* argument is required.

rates
A numeric expression that contains the interest rates charged for *loans.* When *rates* is a dimensioned variable, it can be dimensioned by any dimension, including a different time dimension. When *rates* is dimensioned by a time dimension, you specify the interest rate in each time period that will apply to the loans incurred or outstanding in that period. The interest rates are expressed as decimals; for example, a 5 percent rate is expressed as .05.

n
A numeric expression that specifies the number of payments required to pay off the loans in the series. The *n* expression can be dimensioned, but it cannot be dimensioned by the time dimension argument. One payment is made in each time period of the time dimension by which *loans* is dimensioned or in each time period of the dimension specified in the *time-dimension* argument. For example, one payment a month is made when *loans* is dimensioned by `month`.

time-dimension

The name of the dimension along which the interest payments are calculated. When *loans* has a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR, the *time-dimension* argument is optional, unless *loans* has more than dimension of this type.

Notes

The Result Dimensions of the Result

The result returned by the VPMTSCHED function is dimensioned by the union of all the dimensions of *loans* and *rates* and the dimension used as the *time-dimension* argument.

Time-Period Payment Calculation

VPMTSCHED calculates the payment for a given time period as the sum of the principal and interest due on each loan that is incurred or outstanding in that period.

NA Mismatch Error

When *loans* has a value other than `NA` and the corresponding value of *rates* is `NA`, an error occurs.

NASKIP Option

VPMTSCHED is affected by the NASKIP option. When NASKIP is set to YES (the default), and a loan value is `NA` for the affected time period, the result returned by VPMTSCHED depends on whether the corresponding interest rate has a value of `NA` or a value other than `NA`. Table 24–8, " How NASKIP Affects the Results When a Loan or Rate Value is NA for a Given Time Period" on page 24-81illustrates how NASKIP affects the results when a loan or rate value is `NA` for a given time period.

As an example, suppose a loan expression and a corresponding interest expression both have `NA` values for 1994, but both have values other than `NA` for succeeding years. When the number of payments is 3, VPMTSCHED returns `NA` for 1994, 1995, and 1996. For 1997, VPMTSCHED returns the payment due for loans incurred in 1995, 1996, and 1997.

Time Dimensions

The VPMTSCHED calculation begins with the first value of the time dimension, regardless of how the status of that dimension may be limited. For example, suppose *loans* is dimensioned by `year`, and the values of `year` range from `Yr95` to

`Yr99`. The calculation always begins with `Yr95`, even when you limit the status of `year` so that it does not include `Yr95`.

However, when *loans* is not dimensioned by the time dimension, the VPMTSCHED calculation begins with the first value in the current status of the time dimension. For example, suppose *loans* is not dimensioned by `year`, but `year` is specified as *time-dimension*. When the status of `year` is limited to `Yr97` to `Yr99`, the calculation begins with `Yr97` instead of `Yr95`.

Related Functions

The FPMTSCHED function, which calculates a payment schedule for a series of fixed-rate loans, and the VINTSCHED and FINTSCHED functions, which calculate the interest portion of the payments on variable-rate and fixed-rate loans.

Examples

Example 24–39 Using VPMTSCHED

The following statements create two variables called `loans` and `rates`.

```
DEFINE loans DECIMAL <year>
DEFINE rates DECIMAL <year>
```

Suppose you assign the following values to the variables `loans` and `rates`.

```
YEAR                LOANS      RATES
-------------- ---------- ----------
Yr95               100.00       0.05
Yr96               200.00       0.06
Yr97               300.00       0.07
Yr98                 0.00       0.07
Yr99                 0.00       0.07
```

For each year, `loans` contains the initial value of the variable-rate loan incurred during that year. For each year, the value of `rates` is the interest rate that will be charged for that year on any loans incurred or outstanding in that year.

The following statement specifies that each loan is to be paid off in three payments, calculates the schedule for paying off the principal and interest on the loans,

```
REPORT W 20 HEADING 'Payment' VPMTSCHED(loans, rates, 3, year)
```

and produces the following report.

```
YEAR                      Payment
------------- -------------------
Yr95                        36.72
Yr96                       112.06
Yr97                       227.78
Yr98                       190.19
Yr99                       114.32
```

The payment for 1995 is the principal due on the loan of $100 incurred in 1995, plus interest on the loan at 5 percent. The payment due in 1996 is the sum of the second payment of principal on the loan incurred in 1995, plus the first payment of principal on the loan of $200 incurred in 1996, plus interest on the remaining principals of both loans at 6 percent. The 1997 payment is the sum of the third and final principal payment on the loan incurred in 1995, the second of the three principal payments on the 1996 loan, the first payment of principal on the loan of $300 incurred in 1997, plus interest on the remaining principals of all three loans at 7 percent. Since the 1995 loan is paid off in 1997, the payment for 1998 covers the principal and interest for the 1996 and 1997 loans. The payment for 1999 is the final payment of principal and interest for the 1997 loan.

WEEKDAYSNEWYEAR

For a dimension of type WEEK, the WEEKDAYSNEWYEAR option determines how many days of the new year there must be for a week to be identified as week 1 of the new year.

By default, week 1 in a given year is the first week that contains at least one day in the new year. For example, January 1, 2000, is a Saturday. Using the default, the first week in that year (`W1.00`) is the period from Sunday, December 26, 1999, through Saturday, January 1, 2000.

Using WEEKDAYSNEWYEAR, you can specify how many days of the year must be present in week 1 in that year. When you use WEEKDAYSNEWYEAR to specify that the first week in a year must contain two or more days, then the week of December 26, 1999, through January 1, 2000, is the last week in 1999 (`W53.99`), and the week of January 2 through January 8 is the first week in the year 2000 (`W1.00`).

Data type

INTEGER

Syntax

WEEKDAYSNEWYEAR = *days*

Arguments

days
An INTEGER expression in the range 1 through 7 that indicates how many days in the year must be present in week 1 of that year. The default value for *days* is `1`.

Examples

The Effect of WEEKDAYSNEWYEAR

The following statements send a list of weeks with the associated ending dates for each of those weeks to the current outfile.

```
DEFINE week DIMENSION WEEK
MAINTAIN week ADD '12 18 99' '1 15 00'
weekdaysnewyear = 2
REPORT W 22 CONVERT(week date)
```

These statements produce the following output.

```
WEEK             CONVERT(WEEK DATE)
-------------- --------------------
W51.99         18DEC99
W52.99         25DEC99
W53.99         01JAN00
W1.00          08JAN00
W2.00          15JAN00
```

January 1, 2000, is a Saturday, so setting WEEKDAYSNEWYEAR to `2` causes the week from January 2 through January 8 to appear as `W1.00`.

WEEKOF

The WEEKOF function returns an INTEGER in the range of 1 to 53, which gives the week of the year in which a specified date falls. The result has the same dimensions as the specified DATE expression.

Return Value

INTEGER

Syntax

WEEKOF(*date-expression*)

Arguments

date-expression
An expression that has the DATE data type, or a text expression that specifies a date. See "TEXT-to-DATE Conversion" on page 24-100.

Notes

TEXT-to-DATE Conversion

In place of a DATE expression, you can specify a text expression that has values conforming to a valid input style for dates. Oracle OLAP automatically converts the values of the text expression to DATE values, using the current setting of the DATEORDER option to resolve any ambiguity.

Determining Week 1

The value of WEEKDAYSNEWYEAR specifies how many days of the new year there must be in the week for WEEKOF to consider it to be week 1 of the new year. For example, when January 1 is on a Wednesday, then the week of December 29 to January 4 has four days in the new year. WEEKDAYSNEWYEAR must therefore have a value of 4 or less for that week to be counted as week 1. This determination of week 1 affects the numbering of all weeks in the year.

Examples

Example 24–40 Finding Today's Week

The following statement sends the week of the year in which today's date falls to the current outfile.

```
SHOW WEEKOF(TODAY)
```

When today's date is August 5, 1996, which is a Monday, this statement produces the following output.

```
32
```

Example 24–41 Finding the Week of a Date

The following statement sends the week of the year in which July 4 falls in 1996 to the current outfile.

```
SHOW WEEKOF('04JUL96')
```

This statement produces the following output.

```
27
```

WHILE

The WHILE command repeatedly executes a statement while the value of a Boolean expression remains `TRUE`. You can only use WHILE within a program.

Syntax

WHILE *boolean-expression*

  statement block

Arguments

boolean-expression

Serves as the criterion for statement execution. While the expression remains `TRUE`, *statement* is repeatedly executed. When the expression becomes `FALSE`, the execution of *statement* ceases, and the program continues with the next line. Ensure that something in the *statement* (or statements) eventually causes the Boolean expression to become `FALSE`; otherwise, the code becomes an endless loop.

statement block

One or more statements to be executed while the Boolean expression is `TRUE`. You can execute two or more statements by enclosing them within DO ... DOEND brackets. The DO command should follow immediately after the WHILE command.

Notes

WHILE Compared to IF

The WHILE command's main use is as an alternative to the IF...THEN...ELSE command.When you want one or more statements in your program to execute repeatedly for as long as a Boolean expression remains `TRUE`, you use WHILE. When you want them to execute only once when a Boolean expression is `TRUE`, you use IF.

Boolean Constant

You can specify a constant for the Boolean expression. When your statement is WHILE TRUE, make sure to include a BREAK, RETURN, or EXIT command between DO ... DOEND so the program can finish the loop.

Branching in a Loop

You can use the BREAK, CONTINUE, and GOTO commands to branch within, or out of, a WHILE loop, thereby altering the sequence of statement execution.

Related Statements

IF...THEN...ELSE, DO ... DOEND, and FOR.

Examples

Example 24–42 Using a WHILE Loop in a Program

In the following program lines, the statements following DO are executed as long as the Boolean expression `count LT 10` is `TRUE`. Within the loop, the code searches for an instance of some condition and, when it finds one, it adds `1` to `count`. When `count` reaches `10`, the loop ends. The code in the loop must ensure that `count` will, at some time, reach `10`. Otherwise, the loop will never end.

```
WHILE count LT 10
   DO
    ..." (statements)
    IF ....
      count = count + 1
   DOEND
```

WIDTH_BUCKET

For a given expression, the WIDTH_BUCKET function returns the bucket number into which the value of this expression would fall after being evaluated.

Return Value

An INTEGER.

Syntax

WIDTH_BUCKET (*expr* , *min_value* , *max_value* , *num_buckets*)

Arguments

expr
The expression for which the histogram is being created. This expression must evaluate to a number or a datetime value. When *expr* evaluates to `NA`, then the expression returns `NA`.

min_value
An expression that resolves to the minimum end point of the acceptable range for *expr*. This expression must evaluate to number or datetime values, and can*not* evaluate to `NA`.

max_value
An expression that resolves to the maximum end point of the acceptable range for *expr*. This expression must evaluate to number or datetime values, and can*not* evaluate to `NA`.

num_buckets
An expression that resolves to a constant indicating the number of buckets. This expression must evaluate to a positive INTEGER.

Notes

Underflow Bucket

The function also creates (when needed) an underflow bucket numbered `0` and an overflow bucket numbered *num_buckets*`+1`. These buckets handle values less than *min_value* and more than *max_value* and are helpful in checking the reasonableness of endpoints.

Notes

Constructing Equiwidth Histograms

WIDTH_BUCKET lets you construct equiwidth histograms, in which the histogram range is divided into intervals that have identical size. (Compare this function with NTILE, which creates equiheight histograms.) Ideally each bucket is a "closed-open" interval of the real number line. For example, a bucket can be assigned to cores between `10.00` and `19.999...` to indicate that `10` is included in the interval and `20` is excluded. This is sometimes denoted `(10, 20)`.

WKSDATA

The WKSDATA function returns the data type of each individual cell in a worksheet or the data type of a program argument with the WORKSHEET data type. You can use WKSDATA to help in the process of transferring labels and data between text files and Oracle OLAP.

Return Value

The data type of individual worksheet cells.

Syntax

WKSDATA(*worksheetname*)

Arguments

worksheetname
Specifies the name of an Oracle OLAP worksheet object, such as `workunits`.

Notes

Checking One or More Cells

You can use WKSDATA to return the data type of a single worksheet cell by using a qualified data reference for the cell, as in the following format.

SHOW WKSDATA(*worksheetname*(WKSROW *n*, WKSCOL *n*))

Or you can use the REPORT command in this format with WKSDATA to provide the contents of all the cells in a worksheet side-by-side with their data types.

REPORT *worksheetname* WKSDATA(*worksheetname*)

Multiple Data Types

You should always use care when using worksheet objects in expressions. Because a worksheet object can contain multiple data types, the actual data type of individual worksheet cells is not considered when an OLAP DML statement is compiled. Instead, code is generated to convert each worksheet cell to the data type it expects at that position in the expression. This may lead to unexpected results in some cases.

Text Data

All textual data (as opposed to numeric, Boolean, date, and so on) in a worksheet has the TEXT data type. The ID and NTEXT data types are not supported in worksheets.

Examples

Example 24–43 Checking Data Imported from a Worksheet

Suppose you have imported a flat data file into a worksheet called `workunits`. You can use WKSDATA to provide a quick way to determine which areas to treat as dimension values and which as data values in bringing the worksheet into standard OLAP workspace format.

This statement produces this output following the statement that shows the data in `workunits`

```
REPORT workunits

               -----------------WORKUNITS-----------------
               ------------------WKSCOL-------------------
WKSROW             1          2          3          4
------------- ---------- ---------- ---------- ----------
            1         NA Jan96      Feb96      Mar96
            2 Tents             307        209        277
            3 Canoes            352        411        488
            4 Racquets        1,024      1,098      1,144
            5 Sportswear      1,141      1,262      1,340
            6 Footwear        2,525      2,660      2,728
```

This statement uses the WKSDATA function to produce the report following the statement, which shows the data type of each cell in the worksheet.

```
REPORT WKSDATA(workunits)

               ------------WKSDATA(WORKUNITS)-------------
               ------------------WKSCOL-------------------
WKSROW             1          2          3          4
------------- ---------- ---------- ---------- ----------
            1         NA TEXT       TEXT       TEXT
            2 TEXT       INTEGER    INTEGER    INTEGER
            3 TEXT       INTEGER    INTEGER    INTEGER
            4 TEXT       INTEGER    INTEGER    INTEGER
            5 TEXT       INTEGER    INTEGER    INTEGER
            6 TEXT       INTEGER    INTEGER    INTEGER
```

YESSPELL

(Read-only) The YESSPELL option holds the text that is used for `TRUE` Boolean values in the output of OLAP DML statements.

The value of the YESSPELL option is the word for "yes" in the current language, as specified by the NLS_LANGUAGE option. (See NLS Options on page 18-54.) For example, when NLS_LANGUAGE is set to `American`, then the value of YESSPELL is `YES`. When NLS_LANGUAGE is set to `Spanish`, then the value of YESSPELL is `SI`.

Data type

TEXT

Syntax

YESSPELL

Examples

Example 24–44 Seeing the Effect of the YESSPELL Value

Suppose you have a variable called BOOLVAR that currently has a value of `YES`. When "si" is the word for "yes" in the language specified by the NLS_LANGUAGE option,

```
SHOW boolvar
```

produces the following output.

```
si
```

YRABSTART

The YRABSTART option sets the specific 100-year period associated with years that are read or displayed using a two-digit abbreviation.

Data type

INTEGER

Syntax

YRABSTART = *year*

Arguments

year

A four-digit INTEGER expression that indicates the year at which the 100-year period begins. You can specify any value in the range 1000 to 9999. However, when you specify a value greater than 9900 for *year,* requests to read or display two-digit year values that correspond to a year later than 9999 will result in a return value of `NA`. The default is 1950; two-digit year abbreviations are interpreted as being in the range 1950 to 2049 unless a different range is set through YRABSTART.

Examples

Example 24–45 Using the Default Value

The following statements specify a date format and send output to the current outfile.

```
DATEFORMAT = '<Mtextl> <d>, <yyyy>'
SHOW MAKEDATE(96 9 13)
```

These statements produce the following output.

```
September 13, 1996
```

Example 24–46 Setting the 100-Year Period for a Date

The following statements set a 100-year period of 2000 to 2099 and send the output to the current outfile.

```
YRABSTART = 2000
SHOW MAKEDATE(96 9 13)
```

These statements produce the following output.

```
September 13, 2096
```

YYOF

The YYOF function returns an INTEGER in the range of 1000 to 9999, giving the year in which a specified date falls. The result returned by YYOF has the same dimensions as the specified date expression.

Return Value

INTEGER

Syntax

YYOF(*date-expression*)

Arguments

date-expression
An expression that has the DATE data type, or a text expression that specifies a date. See"TEXT-to-DATE Conversion" on page 24-111.

Notes

TEXT-to-DATE Conversion

In place of a date expression, you can specify a text expression that has values that conform to a valid input style for dates. YYOF automatically converts the values of the text expression to DATE values, using the current setting of the DATEORDER option to resolve any ambiguity.

Commas in Year Values

When the COMMAS option is set to YES when you display the value returned by YYOF, the year is displayed with a comma separating the thousands (for example, `1,996`). To avoid this, you can set the COMMAS option to `NO` before displaying the year.

Examples

Example 24–47 Obtaining the Current Year

The following statements send the year in which today's date falls to the current outfile.

```
COMMAS = NO
SHOW YYOF(TODAY)
```

When today's date is January 15, 1996, these statements produce the following output.

```
1996
```

ZEROROW

The ZEROROW option suppresses report rows with numeric values that are all NAs or all zeros or would be represented as zeros. ZEROROW affects output produced by the REPORT and ROW commands.

Data type

BOOLEAN

Syntax

ZEROROW = {YES|NO}

Arguments

YES
Suppresses report rows that contain any numeric values when all the numeric values would be shown either as zeros or NAs.

NO
Produces all rows of the report, regardless of the values they contain. (Default)

Notes

Non-Numeric Data

Even when a row contains non-numeric data, such as TEXT, ID, or BOOLEAN values, along with numeric values, the row is suppressed when ZEROROW is YES and all the numeric values would be shown either as zeros or NAs.

Values Close to Zero

When your report includes a small number, such as `0.004`, the number of decimal places being shown affects whether ZEROROW treats that number as zero. When you are producing a report with totals, the actual number will be used to calculate the total, even when the number is suppressed.

The Effect of NASPELL

The value of NASPELL does not affect the way ZEROROW handles `NA` values.

The Effect of ZSPELL

The value of ZSPELL does not affect the functioning of ZEROROW; numeric zero values are treated as zeros regardless of their spelling in output.

Examples

Example 24–48 Suppressing Report Rows of All-Zero Data

Suppose you have a variable called `worstcase`, that is dimensioned by `division`, `month`, and `line`, in which you store the results of calculations to project sales. When you produce a report of the results, you want to suppress any rows for which the value of the worst-case projections is zero for all months in the status. Set ZEROROW to YES, as shown in the following statements.

```
ZEROROW = YES
LIMIT line TO 'Revenue'
LIMIT month TO 'Nov95' TO 'Feb96'
REPORT WIDTH 8 DOWN division ACROSS month: worstcase
```

These statements produce the following report.

```
LINE: REVENUE
         -----------------WORSTCASE-----------------
         -------------------MONTH-------------------
DIVISION   Nov95      Dec95      Jan96      Feb96
-------- ---------- ---------- ---------- ----------
Camping        0.00       0.00  45,500.00  47,400.00
Sporting       0.00       0.00  29,200.00  28,400.00
Clothing       0.00       0.00  15,200.00  14,900.00
```

In the preceding report, no rows are suppressed, since some months for each division have projected sales. However, when you lay out this report with `month` down and `division` across, the rows for `Nov95` and `Dec95` are suppressed, because these months have no projected sales.

```
REPORT DOWN month ACROSS division: worstcase
```

This statement produces the following report.

```
LINE: REVENUE
               -----------WORSTCASE------------
               ------------DIVISION------------
MONTH           Camping    Sporting   Clothing
-------------- ---------- ---------- ----------
Jan96           45,500.00  29,200.00  15,200.00
Feb96           47,400.00  28,400.00  14,900.00
```

ZEROTOTAL

The ZEROTOTAL command resets one or all subtotals of specified report columns to zero. You use the ZEROTOTAL command when you produce reports with the ROW command.

Syntax

ZEROTOTAL [{*n*|ALL} [*column1 columnN*]]

Arguments

ZEROTOTAL with no arguments resets *all* subtotals in *all* columns to zero.

n
An INTEGER expression that specifies one of the 32 subtotals (1 to 32) Oracle OLAP accumulates for each numeric column in a report. For the specified columns, this subtotal is set to zero.

ALL
Sets all 32 subtotals to zero for the specified columns. ALL is the default when there are no arguments. To reset all the subtotals to zero for specific columns, you must include ALL in the statement.

```
ZEROTOTAL ALL 1 4 7
```

column
The column number of a report column. Column number 1 refers to the left-most column in a report, regardless of the type of data it contains. When you do not supply any column number arguments, Oracle OLAP sets the specified subtotal (or all subtotals) to zero for all the columns in the report.

Notes

Initializing Column Subtotals

When you use the ROW command to produce a report, use the ZEROTOTAL command at the beginning of the report program to initialize all 32 subtotals for all columns to zero. The REPORT command automatically resets all subtotals to zero before producing output.

Resetting Column Subtotals

You can also use ZEROTOTAL in a report program when you only want to reset some subtotals or when you want to start accumulating new subtotals without inserting the subtotals accumulated so far. A subtotal is automatically reset to zero after it is accessed with the SUBSTR function in its own column. However, a subtotal is not reset to zero after it is accessed with the RUNTOTAL function.

Related Functions

ZEROTOTAL affects the results returned by the RUNTOTAL and SUBSTR functions. See the entries for RUNTOTAL and SUBSTR.

Examples

Example 24–49 Resetting All Report Column Subtotals

In a report, you want to show a dollar sales total, followed by a detailed summary of unit sales for each district. You also want to show a total for unit sales at the end of the report, but you do not want the dollar sales figures included in that total. After generating the total dollar sales, use ZEROTOTAL to reset all your subtotals to zero. Then when you use `SUBTOTAL(1)` later in the report, it only totals the unit sales for each district.

Suppose you have these statement lines in your program.

```
LIMIT product TO 'Footwear'
LIMIT month TO 'Jul96' TO 'Sep96'
ROW 'Total Dollar Sales' ACROSS month: -
   DECIMAL 0 TOTAL(sales month)
BLANK
ROW 'Unit Sales'
ZEROTOTAL ALL
FOR district
   ROW INDENT 5 district ACROSS month: units
ROW 'Total Unit Sales' ACROSS month: -
   OVER '-' SUBTOTAL(1)
```

These statements produce the following output.

```
Total Dollar Sales      607,552    581,229    658,850

Unit Sales
     Boston               3,538      3,369      3,875
     Atlanta              4,058      3,866      4,251
     Chicago              3,943      3,509      4,058
     Dallas                 814        824        867
     Denver               1,581      1,532      1,667
     Seattle              2,053      2,193      2,617
                     ---------- ---------- ----------
Total Unit Sales      15,987.00  15,293.00  17,335.00
```

ZSPELL

The ZSPELL option holds the default text that is used for representing numeric zero values in output produced by the HEADING, REPORT, and ROW commands.

Data type

TEXT

Syntax

ZSPELL = {'*text*'|<u>'OFF'</u>}

Arguments

text

The spelling to use as the default spelling for numeric zero values. When you specify an expression rather than a text literal, you can omit the single quotes.

OFF

Shows a zero (0) with the appropriate number of decimal places (determined by a DECIMAL attribute) for each numeric zero value. (Default)

Notes

Assigning Zero Values

ZSPELL affects output only; it does not affect the way you assign a zero value. For example, even when you have set ZSPELL to `NONE`, you still assign a zero value as follows.

```
var1 = 0
```

Showing Decimal Places

The default of OFF means that a zero value is shown as 0 (zero), with the number of decimal places indicated by a DECIMAL attribute (for example, `0.00`). When you set ZSPELL to the text character `0`, zero values are shown as a `0` with no decimal places, regardless of any DECIMAL specification.

Values Close to Zero

When your output includes a small number, such as `0.004`, the number of decimal places shown affects whether ZSPELL treats the number as zero. See Example 24–51, "Showing Very Small Numbers" on page 24-120.

Examples

Example 24–50 Showing Zero Values as NONE

This example changes the value of ZSPELL, so that a zero value in the DECIMAL variable `testvar` is shown as NONE in report output. When ZSPELL is set to its default value of `OFF`, the Oracle OLAP statements

```
testvar = 0.00
ROW testvar
```

produce the following output.

```
          0.00
```

In contrast, these OLAP DML statements

```
ZSPELL = 'NONE'
ROW testvar
```

produce the following output.

```
          NONE
```

Example 24–51 Showing Very Small Numbers

This example illustrates how the number of decimal places shown in output affects whether ZSPELL treats very small numbers as zeros. When ZSPELL is set to its default value of `OFF`, these OLAP DML statements

```
ZSPELL = 'OFF'
testvar = 0.004
ROW DECIMAL 3 testvar
```

produce the following output.

```
          0.004
```

The following statements set ZSPELL to `NONE` and specify two decimal places for the output.

```
ZSPELL = 'NONE'
ROW DECIMAL 2 testvar
```

These statements produce the following output.

```
          NONE
```

With ZSPELL still set to `NONE`, the following statement specifies three decimal places for the output.

```
ROW DECIMAL 3 testvar
```

This statement produces the following output.

```
          0.004
```

Part III
Appendixes

Part III provides summary information about OLAP DML statements.

This part contains the following appendixes:

A

Functions and Commands by Functional Category

This appendix provides categorized lists of the OLAP DML commands, functions, and programs that you use in OLAP DML programs to analyze data. For listings of OLAP data definition statements, see "OLAP DML as a Data Definition Language" on page 1-5. For options and system properties by category, see "Categories of Options" on page 1-2 and "OLAP DML Properties" on page 1-3.

This chapter includes the following topics:

- File Reading and Writing Statements
- Reporting Statements
- Statements Related to Using OLAP_TABLE in SQL

Session Statements

Table A–1, "General System Statements" lists the OLAP DML functions and commands that you use to find out information about your session.

Table A–1 General System Statements

Statement	Description
CDA	Identifies or changes the current directory object for your session.
EVERSION	Returns a text value that specifies the internal Oracle OLAP build number.
LOG command	Starts or stops the recording of a session to a disk file .
RECAP	Sends statements that were previously entered during the current session to the current outfile or to a file that you specify.
REDO	Re-executes a statement that you entered earlier in your session.
REEDIT	Enables you to edit a statement that you entered earlier in your session.
RESERVED	Returns a list of reserved words in the OLAP DML, or indicates whether or not a word that you specify is reserved in the OLAP DML.
SYSDATE	Returns the current date and time in the format specified by the NLS_DATE_FORMAT option.
SYSINFO	Provides information about the Oracle user for the current session.
SYSTEM	Identifies the platform on which Oracle OLAP is running.

Data Type Conversion

Table A–2, " Data Type Conversion Functions" lists the OLAP DML functions that you to populate variables and relations and to convert data from one data type to another.

Table A–2 Data Type Conversion Functions

Statement	Description
CONVERT	Converts values from one type of data to another.
TCONVERT	Converts data from one dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR to another dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR. You can specify an aggregation method or an allocation method to use in the conversion.
TO_CHAR	Converts a date, number, or NTEXT expression to a TEXT expression in a specified format.
TO_DATE	Converts a formatted TEXT or NTEXT expression to a DATETIME value.
TO_NCHAR	Converts a TEXT expression, date, or number to NTEXT in a specified format.
TO_NUMBER	Converts a formatted TEXT or NTEXT expression to a number.

Assignment Statements

Table A–3, "Assignment Statements" lists the OLAP DML statements that you use to assign values to objects.

Table A–3 Assignment Statements

Statement	Description
SET	Assigns one or more values to a variable, relation, dimension surrogate, worksheet, valueset, or option. When an object has one or more dimensions, the SET command loops over the values in status for each dimension of the target object and assigns a data value to the corresponding cell of the target object
SET1	Assigns a single value to a variable, option, relation, or dimension surrogate. When an object has one or more dimensions, the SET1 command assigns the value to the object cell that is in current status.

Table A–3 Assignment Statements

Statement	Description
MAINTAIN ADD	Adds new TEXT, ID, and INTEGER values to a non-concat dimension or a composite; or adds a new temporary calculated member to a dimension .
UNRAVEL	When used in conjunction with SET, copies the values of an expression into the cells of a variable when the dimensions of the expression are not the same as the dimensions of the variable.

Statements for Working with NA Values

Table A–4, " Statements for Working with NA Values" lists the OLAP DML statements that you use to work with `NA` values.

Table A–4 Statements for Working with NA Values

Statement	Description
CACHE	Within an aggregation specification, tells Oracle OLAP whether to cache or store `NA` values when a summary value calculates to `NA`
COALESCE	Returns the first non-NA expression in a list of expressions, or NA when all of the expressions evaluate to NA.
NAFILL	Returns the values of the source expression with any NA values appearing as the specified fill expression.
NVL	Replaces a NA value with a string.
NVL2	Returns one value when the value of a specified expression is not NA, or another value when the value of the specified expression is NA.

Text Functions

Within the general category of text functions, the OLAP DML statements can be grouped into the following subcategories:

- General character functions
- Byte functions
- Multiline functions

General Character Functions

Table A–5, " General Character Functions" lists the OLAP DML statements that you use to manipulate text based on characters.

Table A–5 General Character Functions

Statement	Description
ASCII	Returns the decimal representation of the first character of an expression.
BLANKSTRIP	Removes leading or trailing blank spaces from text values.
CHANGECHARS	Changes one or more occurrences of a specified string in a text expression to another string.
EXTCHARS	Extracts a portion of a text expression using characters.
FINDCHARS	Returns the character position of the beginning of a specified group of characters within a text expression.
GREATEST	Returns the largest expression in a list of expressions.
INITCAP	Returns a specified text expression, with the first letter of each word in uppercase and all other letters in lowercase.
INSCHARS	Inserts one or more characters into a text expression.
INSTR	Searches a string for a substring using characters and returns the position in the string that is the first character of a specified occurrence of the substring.
JOINCHARS	Joins two or more text values, as characters, as a single line.
LEAST	Returns the smallest expression in a list of expressions.
LIKECASE	Controls whether the LIKE operator is case sensitive.
LIKEESCAPE	An escape character for the LIKE operator.
LOWCASE	Converts all alphabetic characters in a text expression into lowercase.
LPAD	Returns an expression, left-padded to a specified length with the specified characters; or, when the expression to be padded is longer than the length specified after padding, only that portion of the expression that fits into the specified length.
LTRIM	Removes characters from the left of a text expression, with all the leftmost characters that appear in another text expression removed.
MAXCHARS	The number of characters in the longest line of a multiline text expression. The result returned by MAXCHARS has the same dimensions as the specified expression.

Table A–5 (Cont.) General Character Functions

Statement	Description
NULLIF	Compares one expression with another and returns NA when the expressions are equal, or the base expression when they are not.
NUMCHARS	The number of characters in a text expression.
OBSCURE	Provides two mechanisms for encrypting a single-line text expression. Depending on the mechanism you use, OBSCURE can also restore the encrypted value to its original form.
REMCHARS	Removes one or more characters from a text expression and returns the value that remains.
REPLCHARS	Replaces one or more characters in a text expression.
RPAD	Returns an expression, right-padded to a specified length with the specified characters; or, when the expression to be padded is longer than the length specified after padding, only that portion of the expression that fits into the specified length.
RTRIM	Removes characters from the right of a text expression, with all the rightmost characters that appear in another text expression removed.
SUBSTR	Returns a portion of string, beginning at a specified character position, and a specified number of characters long.
TEXTFILL	Reformats a text value to fit compactly into lines of a specified width, regardless of its current format.
TRIM	Removes leading or trailing characters (or both) from a character string.
UPCASE	Converts all alphabetic characters in a text expression into uppercase.

Byte Functions

Table A–6, " Byte Functions" lists the OLAP DML statements that you use to manipulate text based on bytes.

Table A–6 Byte Functions

Statement	Description
CHANGEBYTES	Changes one or more occurrences of a specified string in a text expression to another string.
EXTBYTES	Extracts a portion of a text expression using bytes.

Table A–6 (Cont.) Byte Functions

Statement	Description
FINDBYTES	Returns the byte position of the beginning of a specified group of bytes within a text expression.
INSBYTES	Inserts one or more bytes into a text expression.
INSTRB	Searches a string for a substring using bytes and returns the position in the string that is the first byte of a specified occurrence of the substring.
JOINBYTES	Joins two or more text values, as bytes, as a single line.
MAXBYTES	The number of bytes in the longest line of a multiline text expression.
NULLIF	The number of bytes in a text expression.
REMBYTES	Removes one or more bytes from a text expression and returns the value that remains.
REPLBYTES	Replaces one or more bytes in a text expression.
SUBSTRB	Returns a portion of string, beginning at a specified byte position, and a specified number of bytes long.

Multiline Text Functions

Table A–7, " MultiLine Text Functions" lists the OLAP DML statements that you use to manipulate multiline text.

Table A–7 MultiLine Text Functions

Statement	Description
CHARLIST	Transforms an expression into a multiline text value with a separate line for each value of the original expression.
EXTCOLS	Extracts specified columns from each line of a multiline text value.
EXTLINES	Extracts lines from a multiline text expression.
FILTERLINES	Applies a filter expression that you create to each line of a multiline text expression.
FINDLINES	Determines the position of one or more lines in a multiline text expression.
INLIST	Determines whether every line of a text value is a line in a second text value.
INSCOLS	Inserts into the columns of a multiline TEXT value all the columns of another TEXT value.

Table A–7 (Cont.) MultiLine Text Functions

Statement	Description
INSLINES	Inserts one or more lines into a multiline text expression.
JOINCOLS	Joins the corresponding lines of two or more multiline text values.
JJOINLINES	Joins the values of two or more text expressions into a single multiline value.
MAXBYTES	The number of bytes in the longest line of a multiline text expression.
NUMLINES	The number of lines in each value of a text expression. The result returned by NUMLINES has the same dimensions as the specified expression.
REMCOLS	Removes specified columns from every line of a multiline TEXT value.
REMLINES	Removes one or more lines from a multiline TEXT expression and returns the value that remains.
REPLCOLS	Replaces some or all of the character columns in one multiline TEXT value with the columns of another.
REPLLINES	Replaces one or more lines in a multiline text expression.
SORTLINES	Sorts the lines in a multiline TEXT value.
UNIQUELINES	Removes duplicate lines in a multiline TEXT value and sorts the lines in ascending order.

Date and Time Functions

Table A–8, " Date and Time Functions" describes the OLAP DML date and time functions.

Table A–8 Date and Time Functions

Statement	Description
ADD_MONTHS	Returns the date that is the specified number of months after the specified date.
BEGINDATE	Returns the beginning date of the first time period for which an expression has a non-NA value.
DAYOF	Returns an integer in the range of 1 through 7, giving the day of the week on which a specified date falls.
DDOF	Returns an integer in the range of 1 through 31, giving the day of the month on which a specified date falls.

Table A–8 (Cont.) Date and Time Functions

Statement	Description
ENDDATE	Returns the ending date of the last time period for which an expression has a non-NA value.
ENDOF	Returns the last date of a time period in dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR.
LAST_DAY	Returns the last day of the month in which a particular date falls.
MAKEDATE	Returns the DATE value that corresponds to specified integer values for a year, month, and day.
MMOF	Returns an integer in the range of 1 to 12, giving the month in which a specified date falls. The result returned by MMOF has the same dimensions as the specified DATE expression.
MONTHS_BETWEEN	Calculates the number of months between two dates.
NEW_TIME	Converts a date and time from one time zone to another.
NEXT_DAY	Returns the date of the first instance of a particular day of the week that follows the specified date.
ROUND (for dates and time)	Returns a date and time value rounded to a specified date format; or, when you do not specify a format, the date and time value rounded to the nearest day.
STARTOF	Returns the starting date of a time period in a dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR.
SYSDATE	Returns the current date and time in the format specified by the NLS_DATE_FORMAT option.
TCONVERT	Converts data from one dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR to another dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR.
TOD	Returns the current time of day in the form hh:mm:ss using a 24-hour format.
TODAY	Returns the current date as a DATE value.
TRIM	Returns the date and time value truncated to a specified date format; or, when you do not specify a format, returns the date and time value truncated to the nearest day.
VNF	Assigns a value name format (VNF) to the definition of a dimension with a type of DAY, WEEK, MONTH, QUARTER, or YEAR.

Table A–8 (Cont.) Date and Time Functions

Statement	Description
WEEKOF	Returns an INTEGER in the range of 1 to 53, which gives the week of the year in which a specified date falls.
YYOF	Returns an INTEGER in the range of 1000 to 9999, giving the year in which a specified date falls.

Numeric Functions

Oracle OLAP offers the following types of numeric functions:

- General numeric functions for typical mathematical processing (for example, ranking and finding logs and tangets). For listing. see Table A–9, " General Numeric Functions".
- Financial functions. For listing. see Table A–10, " Financial Functions".
- Statistical functions. For listing. see Table A–11, " Statistical Functions".
- Time-series functions such as LAG and MOVINGMIN. For listing. see Table A–12, "Time-Series Functions".
- Aggregation functions, such as COUNT and TOTAL. For listing. see Table A–13, " Aggregation Functions"

General Numeric Functions

Table A–9, " General Numeric Functions" lists the OLAP DML functions for calculation.

Table A–9 General Numeric Functions

Function	Description
ABS	Calculates the absolute value of an expression.
ANTILOG	Calculates the value of e (the base of natural logarithms) raised to a specific power.
ANTILOG10	Calculates the value of 10 raised to a specified power.
ARCCOS	Calculates the angle value (in radians) of a specified cosine.
ARCSIN	Calculates the angle value (in radians) of a specified sine.
ARCTAN	Calculates the angle value (in radians) of a specified tangent.

Table A–9 (Cont.) General Numeric Functions

Function	Description
ARCTAN2	Returns a full-range (0 - 2 pi) numeric value indicating the arc tangent of a given ratio.
BITAND	Computes an AND operation on the bits of two integers.
CEIL	Returns the smallest whole number greater than or equal to a specified number.
COS	Calculates the cosine of an angle expression.
COSH	Calculates the hyperbolic cosine of an angle expression.
DECODE	Compares one expression to one or more other expressions and, when the base expression is equal to a search expression, returns the corresponding result expression; or, when no match is found, returns the default expression when it is specified, or NA when it is not.
EXP	Returns e raised to the nth power, where e equals 2.71828183....
FLOOR	Returns the largest whole number equal to or less than a specified number.
GREATEST	Returns the largest expression in a list of expressions. All expressions after the first are implicitly converted to the data type of the first expression before the comparison.
INSTRB	Calculates the integer part of a decimal number by truncating its decimal fraction.
LEAST	Returns the smallest expression in a list of expressions. All expressions after the first are implicitly converted to the data type of the first expression before the comparison.
LOG function	Computes the natural logarithm of an expression.
LOG10	Computes the logarithm base 10 of an expression.
MAX	Calculates the larger value of two expressions.
MIN	Calculates the smaller value of two expressions.
NULLIF	Compares one expression with another and returns NA when the expressions are equal, or the base expression when they are not.
REM	Returns the remainder after one numeric expression is divided by another.
ROUND (for numbers)	Returns the number rounded to the nearest multiple of a second number you specify or to the number of decimal places indicated by the second number.

Table A–9 (Cont.) General Numeric Functions

Function	Description
SIGN	Returns a value that indicates if a specified number is less than, equal to, or greater than 0 (zero).
SIN	Calculates the sine of an angle expression. The result returned by SIN is a decimal value with the same dimensions as the specified expression.
SINH	Calculates the hyperbolic sine of an angle expression.
SQRT	Computes the square root of an expression.
TAN	Calculates the tangent of an angle expression.
TANH	Calculates the hyperbolic tangent of an angle expression.
TRUNC (for numbers)	Truncates a number to a specified number of decimal places.
WIDTH_ BUCKET	Returns the bucket number into which the value of an expression would fall after being evaluated.

Financial Functions

Table A–10, " Financial Functions" lists the OLAP DML functions for financial calculation.

Table A–10 Financial Functions

Function	Description
DEPRDECL	Calculates the depreciation expenses for a series of assets. DEPRDECL uses the declining balance method to depreciate the assets over the specified lifetime of the assets.
DEPRDECLSW	Calculates the depreciation expenses for a series of assets. DEPRDECLSW uses a variation on the declining balance method to depreciate assets over the specified lifetime of the assets.
DEPRSL	Calculates the depreciation expenses for a series of assets. DEPRSL uses the straight-line method to depreciate the assets over the specified lifetime of the assets.
DEPRSOYD	Calculates the depreciation expenses for a series of assets. DEPRSOYD uses the sum-of-years'-digits method to depreciate the assets over the specified lifetime of the assets.
FINTSCHED	Calculates the interest portion of the payments on a series of fixed-rate installment loans that are paid off over a specified number of time periods.

Table A–10 (Cont.) Financial Functions

Function	Description
FPMTSCHED	Calculates a payment schedule (principal plus interest) for paying off a series of fixed-rate installment loans over a specified number of time periods.
GROWRATE	Calculates the growth rate of a time-series expression, based on the first and last values of the series.
IRR	Computes the internal rate of return associated with a series of cash flow values. Each value of the result is calculated to be the discount rate for a period that makes the net present value of the corresponding cash flows equal to zero.
NPV	Computes the net present value of a series of cash flow values.
VINTSCHED	Calculates the interest portion of the payments on a series of variable-rate installment loans that are paid off over a specified number of time periods.
VPMTSCHED	Calculates a payment schedule (principal plus interest) for paying off a series of variable-rate installment loans over a specified number of time periods.

Statistical Functions

Table A–11, " Statistical Functions" lists the OLAP DML functions for statistical calculation.

Table A–11 Statistical Functions

Statement	Description
CATEGORIZE	Groups the values of a numeric expression into categories.
CORRELATION	Returns the correlation coefficients for the pairs of data values in two expressions.
NORMAL	Returns a random value from a normal distribution with a specified mean and standard deviation. The result returned by NORMAL is dimensioned by all the dimensions of the mean and standard deviation expressions.
RANDOM	Produces a number that is randomly distributed between specified low and high boundaries.
STDDEV	Calculates the standard deviation of the values of an expression.

Time-Series Functions

Table A–12, "Time-Series Functions" lists the OLAP DML time-series functions.

Table A–12 Time-Series Functions

Function	Description
CUMSUM	Computes cumulative totals over a dimension.
LAG	Returns the values of a dimensioned variable or expression at a specified offset of a dimension prior to the current value of that dimension.
LAGABSPCT	Returns the percentage difference between the value of a dimensioned variable or expression at a specified offset of a dimension prior to the current value of that dimension and the current value of the dimensioned variable or expression.
LAGDIF	Returns the difference between the value of a dimensioned variable or expression at a specified offset of a dimension prior to the current value of that dimension and the current value of the dimensioned variable or expression.
LAGPCT	Returns the percentage difference between the value of a dimensioned variable or expression at a specified offset of a dimension prior to the current value of that dimension and the current value of the dimensioned variable or expression.
LEAD	Returns the values of a dimensioned variable or expression at a specified offset of a dimension subsequent to the current value of that dimension.
MOVINGAVERAGE	Computes a series of averages for the values of a dimensioned variable or expression over a specified dimension. For each dimension value in status, MOVINGAVERAGE computes the average of the data in the range specified, relative to the current dimension value.
MOVINGMAX	Returns a series of maximum values of a dimensioned variable or expression over a specified dimension. For each dimension value in status, MOVINGMAX searches the data for the maximum value in the range specified, relative to the current dimension value.
MOVINGMIN	Returns a series of minimum values for the values of a dimensioned variable or expression over a specified dimension. For each dimension value in status, MOVINGMIN searches the data for the minimum value in the range specified, relative to the current dimension value.

Table A–12 Time-Series Functions

Function	Description
MOVINGTOTAL	Computes a series of totals for the values of a dimensioned variable or expression over a specified dimension. For each dimension value in status, MOVINGTOTAL computes the total of the data in the range specified, relative to the current dimension value.

Aggregation Functions

Table A–13, " Aggregation Functions" lists the OLAP DML aggregation functions. The OLAP DML also provides an aggmap object that you can use to aggregate data, see "Aggregation Statements" on page A-17 for a list of related OLAP DML statements.

Table A–13 Aggregation Functions

Statements	Description
ANY	Returns YES when any values of a Boolean expression are TRUE, or NO when none of the values are TRUE.
AVERAGE	Calculates the average of the values of an expression.
COUNT	Retrieves the number of TRUE values of a Boolean expression, or 0 (zero) if no values of the expression are TRUE.
EVERY	Returns YES when every value of a Boolean expression is TRUE, or NO if any value of the expression is FALSE.
LARGEST	Returns the largest value of an expression. You can use this function to compare numeric values or date values.
MEDIAN	Calculates the median of the values of an expression.
MODE	Returns the mode (the most frequently occurring value) of a numeric expression; or NA when there are no duplicate values in the data.
NONE	Returns YES when no values of a Boolean expression are TRUE; or NO when any value of the expression is true.
PERCENTAGE	Computes the percent of total for each value in a numeric expression.
SMALLEST	Returns the smallest value of an expression. You can use this function to compare numeric values or date values.
TCONVERT	Converts data from one dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR to another dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR. You can specify an aggregation method or an allocation method to use in the conversion.

Table A–13 (Cont.) Aggregation Functions

Statements	Description
TOTAL	Calculates the total of the values of an expression.

Forecast and Regression Statements

Within the general category of forecast and regression statements, the OLAP DML statements can be grouped in the following subcategories:

- Simple forecasts and regressions
- Forecasts and regressions using a forecasting context

Simple Forecasts and Regressions

Table A–14, " Statements for Simple Forecasts and Regressions" lists the OLAP DML that you use to calculate simple forecasts and regressions.

Table A–14 Statements for Simple Forecasts and Regressions

Statement	Description
FORECAST	Forecasts data by one of three methods: straight-line trend, exponential growth, or Holt-Winters extrapolation.
FORECAST.REPORT	A program that produces a standard report of a forecast generated using the FORECAST command.
INFO	Obtains information that has been produced by the FORECAST command or the REGRESS command.
REGRESS	Calculates a simple multiple linear regression or a weighted regression.
REGRESS.REPORT	A program that produces a standard report of a regression created using the REGRESS command.
SMOOTH	Computes a single or a double exponential smoothing of a numeric expression.

Statements for Forecasting Using a Forecasting Context

Table A–15, "Statements for Forecasting Using a Forecasting Context" lists the OLAP DML that you use to calculate a sophisticated forecast using a forecasting context. Typically, you use these statements in an OLAP DML program in the order in which they are listed.

Table A–15 Statements for Forecasting Using a Forecasting Context

Statement	Description
FCOPEN	Creates a forecasting context and returns a handle to this context.
FCSET	Sets the values of various parameters that determine the characteristics of the forecast.
FCEXEC	Executes a forecast based on the parameters options specified by the FCSET command for the forecast.
FCQUERY	Returns the results of a forecast created when the FCEXEC command executed.
FCCLOSE	Closes a forecasting context.

Aggregation Statements

Table A–16, " General Aggregation Statements" lists the OLAP DML statements that support data aggregation. The OLAP DML also provides the aggregation functions listed in Table A–13, " Aggregation Functions".

Table A–16 General Aggregation Statements

Statement	Description
AGGMAP	Marks the aggmap as anaggregation specification and enters or changes the aggregation specification.
AGGMAP ADD or REMOVE model	Adds or removes a model from a previously defined aggmap object of type AGGMAP.
AGGMAP SET	Specifies the default aggmap for a variable.
AGGMAPINFO	Returns information about the specification for an aggmap object in your analytic workspace.
AGGREGATE command	Calculates data for one or more variables as specified by the specified aggmap object.
AGGREGATE function	Calculates the data of a variable at runtime, in response to a user's request. Often used as the expression of a $NATRIGGER property.
AGGREGATION	Within a model, creates a custom aggregation.
ALLCOMPILE	A program that compiles every compilable object in your current analytic workspace, one at a time.
COMPILE	Generates compiled code for a compilable object, such as an OLAP DML program, formula, model, or aggmap without running it and saves the compiled code in the analytic workspace.

Table A–16 (Cont.) General Aggregation Statements

Statement	Description
DEFINE AGGMAP	Creates a new aggmap object.
MAINTAIN ADD	Adds a new temporary calculated member as a custom aggregation to a dimension or adds new values to a non-concat dimension or a composite.
PARTITIONCHECK	Identifies whether an aggmap object is compatible with the partitioning specified by a partition template object.
ROLLUP	Without the use of an aggmap object, calculates totals for a hierarchy of values where each level of the hierarchy is an aggregation of the values in the level below it and the members of the hierarchy are contained in a single "embedded-total" dimension, so called because it contains both a detail (lowest) level and levels that are aggregations of lower levels.
TCONVERT	Converts data from one dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR to another dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR. You can specify an aggregation method or an allocation method to use in the conversion.

Allocation Statements

Table A–17, " General Allocation Statements" lists the OLAP DML statements that you use to allocate data.

Table A–17 General Allocation Statements

Statement	Description
DEFINE AGGMAP	Creates a new aggmap object.
ALLOCMAP	Marks an aggmap as an allocation specification and enters or changes an allocation specification.
AGGMAPINFO	Returns information about the specification for an aggmap object in your analytic workspace.
ALLOCATE	Allocates values into a variable based on the specification provided by an aggmap object.
TCONVERT	Converts data from one dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR to another dimension of type DAY, WEEK, MONTH, QUARTER, or YEAR. You can specify an aggregation method or an allocation method to use in the conversion.

Workspace Object Operation Statements

Table A–18, " Workspace Object Operation Statements" lists the OLAP DML statements that you use for common workspace operations such as defining or deleting the object and defining the contents of the object

Table A–18 Workspace Object Operation Statements

Statement	Description
LOAD	Loads the definition of an OLAP DML program, formula, or model into memory.
CLEAR	Deletes the data that you specify for one or more variables.
GROUPINGID	Populates a previously-defined variable with the grouping ids for the values of a hierarchical dimension.
HIERHEIGHT command	Populates a previously-defined relation with the values of a specified hierarchical dimension by level.
HIERHEIGHT function	Returns the value of a node at a specified level for the first value in the current status list of a hierarchical dimension.
LOAD	Loads the definition of an OLAP DML program, formula, or model into memory.
PERMIT	Controls access to analytic workspace objects by granting or denying read-only and read/write access permission for workspace objects and for specific values of dimensions and dimensioned objects; and by granting or denying permission to maintain dimensions and to change permission for workspace objects.
PERMITRESET	Causes the values of permission conditions to be reevaluated. Permission conditions consist of one or more Boolean expressions that designate the criteria used by PERMIT commands associated with an object.
VALSPERPAGE	Calculates the maximum number of values for a variable of a given width that will fit on one page. Pages are units of storage in the workspace.

Dimension and Composite Operation Statements

Table A–19, " Dimension and Composite Operation Statements" lists the OLAP DML statements that you use to define the contents of dimensions and composites and to manipulate dimension status.

Table A–19 Dimension and Composite Operation Statements

Statement	Description
ALLSTAT	Sets the status of all dimensions in the current analytic workspace to all their values.
BASEDIM	Returns the name of the dimension from which the current value of a concat dimension comes.
BASEVAL	Returns the values of the base dimensions of a concat dimension. If a base dimension is a concat dimension, then the values of its base dimensions are returned, also.
HIERCHECK	Checks the parent relation of a hierarchical dimension to make sure it has no loops (that is, that no value is specified as its own ancestor or descendant in the parent relation).
INSTAT	Checks whether a dimension or dimension surrogate value is in the current status list or whether a dimension value is in a valueset.
ISVALUE	Tests whether a dimension or a composite has a specified value.
KEY	Returns the value of the specified base dimension for a value of a conjoint dimension or a composite.
LIMIT command	Sets the current status list of a dimension and its dimension surrogates, or assigns values to a valueset.
LIMIT function	Returns the dimension or dimension surrogate values that are currently in status.
MAINTAIN	Adds non-concat dimension values (including temporary calculated members) and composite values; deletes non-concat dimension values and composite values; moves non-concat and concat dimension values; and rename and merges non-concat dimension values.
QUAL	Specifies a qualified data reference (QDR).
SORT	Arranges the order of values in the current status list of a dimension or a dimension surrogate, or in a valueset.
STATALL	Returns YES when default status is currently in effect for a given dimension (that is, when STATLIST would return ALL); or NO when default status is not currently in effect for a given dimension
STATFIRST	Returns the first value in the current status list of a dimension or a dimension surrogate, or in a valueset.
STATLAST	Returns the last value in the current status list of a dimension or a dimension surrogate, or in a valueset.
STATLEN	Returns the number of values in the current status list of a dimension or a dimension surrogate, or in a valueset.

Table A–19 (Cont.) Dimension and Composite Operation Statements

Statement	Description
STATLIST	Returns a list of all values in the current status list of a dimension or dimension surrogate, or in a valueset.
STATMAX	Returns the latest value in the current status list of a dimension or a dimension surrogate, or in a valueset.
STATMIN	Returns the earliest value in the current status list of a dimension or a dimension surrogate, or in a valueset.
STATRANK	Returns the position of a dimension or dimension surrogate value in the current status list or in a valueset.
STATUS	Sends to the current outfile the status of one or more dimensions, dimension surrogates, or valuesets, or the status of all dimensions in an analytic workspace.
STATVAL	Returns the dimension value that corresponds to a specified position in the current status list of a dimension or a dimension surrogate, or in a valueset.
TALLY	The number of values of a dimension that correspond to each value of one or more related dimensions.
VALUES	Returns the default status list or the current status list of a dimension or dimension surrogate, or it returns the values in a valueset.

Formula Statements

Table A–20, " Statements for Formulas" lists the OLAP DML statements that you use when working with formula objects.

Table A–20 Statements for Formulas

Statement	Description
DEFINE FORMULA	Creates a new formula object.
EQ	Specifies the expression to be calculated for a formula that has already been defined. Be sure to distinguish between the EQ statement and the EQ operator used to compare values of the same type.

Modeling Statements

Table A–21, " General Modeling Statements" lists the OLAP DML statements that you use to create and manipulate model objects.

Table A–21 General Modeling Statements

Statement	Description
DEFINE MODEL	Creates a new model object.
INFO	Obtains information that has been produced for a model in your analytic workspace.
MODEL	At the command level, adds contents to a model object. Within an aggmap, executes a predefined model.
MODEL.COMPRPT	Produces a report that shows how model equations are grouped into blocks.
MODEL.DEPRT	Produces a report that lists the variables and dimension values on which each model equation depends.
MODEL.XEQRPT	Produces a report about the execution of the model.

Programming Statements

Within the general category of programming, the OLAP DML statements can be grouped into the following subcategories:

- Handling programs
- Statements that are only used in programs
- Statements that are primarily used in programs
- Debugging programs
- Creating and managing trigger programs

Additionally, you often use statements for forecasts, regression, reporting, importing and exporting data, embedding SQL within an OLAP DML program, and triggering the execution of programs when a particular OLAP DML program executes. For tables outlining these statements see "Forecast and Regression Statements" on page A-16 and "File Reading and Writing Statements" on page A-28, "Statements for Importing and Exporting Data" on page A-29, "Reporting Statements" on page A-29, and "Statements for Working with Startup and Trigger Programs" on page A-26.

Statements for Handling Programs

Table A–22, " Statements for Handling Programs" lists the OLAP DML statements that you use to hide, compile, and call programs.

Table A–22 Statements for Handling Programs

Statement	Description
ALLCOMPILE	Compiles every compilable object in your current analytic workspace, one at a time.
CALL	Invokes an OLAP DML program, and, when the program has arguments, passes these arguments to the called program.
COMPILE	Generates compiled code for a compilable object, such as an OLAP DML program, formula, model, or aggmap without running it and saves the compiled code in the analytic workspace.
DEFINE PROGRAM	Creates a new program object.
PROGRAM	Assigns contents to the most recently defined or considered OLAP DML program.
HIDE	Hides the text of a program, so that you cannot display it using the DESCRIBE command, the EDIT command, or the OBJ function. You can perform all other actions on the program, including executing, compiling, renaming, or exporting.
UNHIDE	Unhides the text of a program that has been made invisible by using the HIDE command.

Statement Used Only in Programs

Table A–23, " Statements Used Only in OLAP DML Programs" lists the OLAP DML statements that you can use only within the contents of an OLAP DML program.

Table A–23 Statements Used Only in OLAP DML Programs

Statement	Description
ARG	Lets you reference arguments passed to a program by returning one argument as a text value.
ARGCOUNT	Returns the number of arguments that were specified when the current program was invoked.
ARGFR	Lets you reference the arguments that are passed to a program by returning a group of one or more arguments, beginning with the specified argument number, as a single text value.
ARGS	Lets you reference the arguments that are passed to a program by returning all the arguments as a single text value.
ARGUMENT	Declares an argument that is expected by a program.

Table A–23 (Cont.) Statements Used Only in OLAP DML Programs

Statement	Description
BREAK	Transfers program control from within a SWITCH, FOR, or WHILE statement to the statement immediately following the DOEND associated with SWITCH, FOR, or WHILE.
CALLTYPE	Returns a value that Indicates whether a program was invoked as a function, as a command, or by using the CALL command.
CONTINUE	Transfers program control to the end of a FOR or WHILE loop (just before the DO/DOEND statement), allowing the loop to repeat. You can use CONTINUE only within programs and only with FOR or WHILE.
DO ... DOENDs	Brackets a group of one or more statements. DO and DOEND are normally used to bracket a group of statements that are to be executed under a condition specified by an IF statement, a group of statements in a repeating loop introduced by FOR or WHILE, or the CASE labels for a SWITCH statement.
FOR	Specifies one or more dimensions whose status will control the repetition of one or more statements.
GOTO	Alters the sequence of statement execution within the program by indicating the next program statement to execute.
IF...THEN...ELSE	Executes one or more statements in a program if a specified condition is met. Optionally, it also executes an alternative statement or group of statements when the condition is not met.
RETURN	Terminates execution of a program prior to its last line. You can optionally specify a value that the program will return.
SIGNAL	Produces an error message and halts normal execution of the program. When the program contains an active trap label, execution branches to the label. Without a trap label, execution of the program terminates and, if the program was called by another program, execution control returns to the calling program.
SWITCH	Provides a multipath branch in a program. The specific path taken during program execution depends on the value of the control expression that is specified with SWITCH.
TEMPSTAT	Limits the dimension you are looping over, inside a FOR loop or inside a loop that is generated by the REPORT command. Status is restored after the statement following TEMPSTAT. If a DO ... DOEND phrase follows TEMPSTAT, status is restored when the matched DOEND or a BREAK or GOTO statement is encountered.

Table A–23 (Cont.) Statements Used Only in OLAP DML Programs

Statement	Description
TRAP	Causes program execution to branch to a label when an error occurs in a program or when the user interrupts the program. When execution branches to the trap label, that label is deactivated.
VARIABLE	Declares a local variable or valueset for use within a program. A local variable cannot have any dimensions and exists only while the program is running.
WHILE	Repeatedly executes a statement while the value of a Boolean expression remains TRUE.
END	Marks the end of the program contents.

Statements Used Primarily in Programs

Table A–24, "Statements Used Primarily in OLAP DML Programs" lists the OLAP DML statements that are used primarily in OLAP DML programs.

Table A–24 Statements Used Primarily in OLAP DML Programs

Statement	Description
ACROSS	Specifies a text expression that contains one or more statements to be executed in a loop.
CONTEXT command	Lets you create and use a context during your Oracle OLAP session. A context is a means of preserving object values. After you create a context, you can save the current status of dimensions and the values of options, single-cell variables, valuesets, and single-cell relations in the context. You can then restore some or all of the object values from the context.
CONTEXT function	Obtains information about object values that are saved in a context. You must first create the context with the CONTEXT command.
INFO (PARSE)	Obtains information that has been produced by the PARSE command.
PARSE	Parses a specified group of expressions.
POP	Restores the status of a dimension, the status of a valueset, or the value of an option or single-cell variable that was saved with a previous PUSH command.
POPLEVEL	Restores all values saved with PUSH commands that were executed since the last POPLEVEL command specifying the same marker.
PUSH	Saves the current status of a dimension, the status of a valueset, or the value of an option or single-cell variable.

Table A–24 Statements Used Primarily in OLAP DML Programs

Statement	Description
PUSHLEVEL	Marks the start of a series of PUSH commands.
SLEEP	Suspends the operation of Oracle OLAP for at least the specified number of seconds.

Statements for Program Debugging

Table A–25, " OLAP DML Program Debugging Statements" lists the OLAP DML statements that you use to debug OLAP DML programs.

Table A–25 OLAP DML Program Debugging Statements

Statement	Description
BACK	Returns the names of all currently executing programs, listed one a line in a multiline text value.
DBGOUTFILE	Sends debugging information to a file.
MONITOR	Records data on the performance cost of each line in a specified OLAP DML program.
TRACKPRG	Tracks the performance cost of every OLAP DML program that runs while you have tracking turned on.

Statements for Working with Startup and Trigger Programs

Trigger programs and startup programs are programs that Oracle OLAP automatically executes when a particular OLAP DML statement executes. Table A–26, "Statements for Working with Startup and Trigger Programs" on page A-26 lists the OLAP DML statements that you can use to create and manage trigger programs.

Table A–26 Statements for Working with Startup and Trigger Programs

Statement	Description
CALLTYPE	Within an OLAP DML program, the CALLTYPE function indicates whether a program was invoked as a function, as a command, by using the CALL command, or triggered by the execution of an OLAP DML statement.

Table A–26 Statements for Working with Startup and Trigger Programs

Statement	Description
ONATTACH	A program that you create and that Oracle OLAP checks for by name when an AW ATTACH statement executes. Depending on the value returned by the program, Oracle OLAP executes the code within the program immediately after attaching the analytic workspace.
PERMIT_READ	A program that you create and that Oracle OLAP checks for by name when an AW ATTACH read-only statement executes. Depending on the value returned by the program, Oracle OLAP executes the code within the program after attaching the analytic workspace.
PERMIT_WRITE	A program that you create and that Oracle OLAP checks for by name when an AW ATTACH read/write statement executes. Depending on the value returned by the program, Oracle OLAP executes the code within the program after attaching the analytic workspace.
TRIGGER command	Associates a previously-created program to a previously-defined object and identifies the object event that automatically executes the program; or a disassociates a trigger program from the object.
TRIGGER function	Retrieves the event, subevent, or name of the object or analytic workspace that caused the execution of a TRIGGER_DEFINE program, a TRIGGER_DEFINE program, or any programs identified as triggers using the TRIGGER command.
TRIGGER_AFTER_UPDATE	A program that you create and that Oracle OLAP checks for by name when an UPDATE statement executes. When the program exists, Oracle OLAP executes the program after the UDPATE occurs.
TRIGGER_BEFORE_UPDATE	A program that you create and that Oracle OLAP checks for by name when an UPDATE statement executes. When the program exists, Oracle OLAP executes the program and then, depending on the value returned by the program (if any), either does nor does not update the workspace.
TRIGGER_DEFINE	A program that you create and that Oracle OLAP checks for by name when a DEFINE statement executes. When the program exists, Oracle OLAP executes the program and then, depending on the value returned by the program (if any), either does nor does not define the object.
TRIGGERASSIGN	Typically used in trigger program for an Assign event, the TRIGGERASSIGN statement replaces one assigned value.

File Reading and Writing Statements

Table A–27, " File Reading and Writing Statements" lists the OLAP DML statements that you use when reading data from files or to files.

Table A–27 File Reading and Writing Statements

Statement	Description
CDA	Identifies or changes the current directory object for your session.
FETCH	Closes an open file. If the file has not been opened, an error occurs.
FILECOPY	Copies the contents of one file (the source file) to another file (the target file).
FILEDELETE	Deletes a file from the operating system disk space.
FILEERROR	Returns information about the first error that occurred when you are processing a record from an input file with the data reading statements FILEREAD and FILEVIEW.
FILEGET	Returns text from a file that has been opened for reading; or NA when FILEGET reaches the end of the file.
FILEMOVE	Changes the name or location of a file that you specify. The new file name may be the same or different from the original name.
FILENEXT	Makes a record available for processing by the FILEVIEW command.
FILEOPEN	Opens a file, assigns it a fileunit number (an arbitrary integer), and returns that number.
FILEPAGE	Forces a page break in your output when PAGING is on.
FILEPUT	Writes data that is specified in a text expression to a file that is opened in WRITE or APPEND mode.
FILEQUERY	Returns information about a file.
FILEREAD	Reads records from an input file and processes data according to action statements that you specify.
FILESET	Sets the paging attributes of a specified fileunit.
FILEVIEW	In conjunction with the FILENEXT function, reads one record at a time of an input file, processes the data, and stores the data in Oracle OLAP dimensions and variables according to the descriptions of the fields.
GET	Requests input from the current input stream.
INFILE	Reads statement input from a specified file.

Table A–27 (Cont.) File Reading and Writing Statements

Statement	Description
LISTFILES	Lists all the open files that can be referenced by the FILEQUERY function.
LOG command	Starts or stops the recording of a session to a disk file. All lines of input and output are recorded.
OUTFILE	Redirects the text output of statements to a file.
RECNO	Reports the current record number of a file opened for reading; or NA when Oracle OLAP has reached the end of the file.

Statements for Importing and Exporting Data

Table A–28, " Statements for Importing and Exporting Data" lists the OLAP DML statements that you use to import and export data.

Table A–28 Statements for Importing and Exporting Data

Statements	Description
EXPORT	Copies both data and object definitions from your workspace to an EIF file, or copies an OLAP DML worksheet object to a spreadsheet file.
IMPORT	Copies data from an EIF file, a text file, or a spreadsheet into an analytic workspace.
WKSDATA	Returns the data type of each individual cell in a worksheet.
SQL	Typically, used in a program to copy data to and from relational tables, passes instructions written in Structured Query Language (SQL) to the relational manager from Oracle OLAP.

Reporting Statements

Table A–29, " Reporting Statements" lists the OLAP DML statements that you use to create simple reports.

Table A–29 Reporting Statements

Statement	Description
BLANK	Sends one or more blank lines to the current outfile.

Table A–29 (Cont.) Reporting Statements

Statement	Description
COLVAL	Within a ROW command, ROW function, or REPORT command, returns a numeric value from a column to the left of the current column in the same row of a report.
HEADING	Produces titles and column headings for a report.
PAGE	Forces a page break in output when PAGING is set to YES.
REPORT	Produces output for one or more data expressions.
ROW command	Produces a line of data in cells, one after another in a single row.
ROW function	Returns a line of data in cells, one after another in a single row.
RUNTOTAL	Within a ROW command, ROW function, or REPORT command, returns the running total of an expression.
SHOW	Displays a single value of an expression.
STDHDR	Generates the standard Oracle OLAP heading at the top of every page of report output.
SUBTOTAL	Within a ROW command, ROW function, or REPORT command, returns the value of one of the subtotals accumulated in a report.
ZEROTOTAL	Within a ROW command, ROW function, or REPORT command, resets one or all subtotals of specified report columns to zero.

Statements Related to Using OLAP_TABLE in SQL

Table A–30, " Statements Related to OLAP_TABLE" lists the OLAP DML statements that support the use of the OLAP_TABLE function.

Table A–30 Statements Related to OLAP_TABLE

Statement	Description
FETCH	Specifies how analytic workspace data is retrieved for use in the relational table created by the OLAP_TABLE function which you use to access analytic workspace data using SQL.
GROUPINGID	Populates a previously-defined variable with the grouping ids for the values of a hierarchical dimension.
HIERHEIGHT command	Populates a previously-defined relation with the values of a specified hierarchical dimension by level.

Table A–30 (Cont.) Statements Related to OLAP_TABLE

Statement	Description
LIMITMAPINFO	Returns the analytic workspace expression that a specified limit map uses to map data into a specified column of a relational table .

B

OLAP DML Statement Changes

This appendix contains listings of the changes made to the OLAP DML.

- Statements Added
- Statements Deleted
- Statements Significantly Changed
- Statements Renamed

Statements Added

The following statements have been added to the OLAP DML. The number in parentheses indicates the specific release in which the statement was added.

$AGGMAP (10.1.0.0)
$AGGREGATE_FROM (10.1.0.0)
$AGGREGATE_FROMVAR (10.1.0.0)
$ALLOCMAP (10.1.0.0)
$COUNTVAR (10.1.0.0)
ACQUIRE (10.1.0.0)
ADD_MONTHS (9.0.0.0)
ALLOCATE (9.2.0.0)
ALLOCERRLOGFORMAT (9.2.0.0)
ALLOCERRLOGHEADER (9.2.0.0)
ALLOCMAP (9.2.0.0)
ARCTAN2 (10.1.0.0)
ASCII (10.1.0.0)
BASEDIM (9.2.0.0)
BASEVAL (9.2.0.0)
BITAND (10.1.0.0)

CDA (9.2.0.0)
CEIL (9.0.0.0)
CHANGEBYTES (9.0.0.0)
CHGDIMS (9.2.0.0)
CHILDLOCK (9.2.0.0)
COALESCE (10.1.0.0)
COMMIT (9.2.0.0)
DEADLOCK (9.2.0.0)
DECODE (10.1.0.0)
DEFINE PARTITION TEMPLATE (10.1.0.0)
DROP DIMENSION (10.1.0.0)
ERRORLOG (9.2.0.0)
ERRORMASK (9.2.0.0)
EXP (10.1.0.0)
EXTBYTES (9.0.0.0)
FETCH (9.2.0.0)
FINDBYTES (9.0.0.0)
FLOOR (9.0.0.0)
GREATEST (10.1.0.0)
GROUPINGID (9.2.0.0)
HIERHEIGHT command (9.2.0.0)
HIERHEIGHT function (9.2.0.0)
INF_STOP_ON_ERROR (10.1.0.0)
INITCAP (10.1.0.0)
INSBYTES (9.0.0.0)
INSTR (10.1.0.0)
INSTRB (10.1.0.0)
JOINBYTES (9.0.0.0)
LAST_DAY (9.0.0.0)
LEAST (10.1.0.0)
LIMITMAPINFO (9.2.0.2)
LPAD (10.1.0.0)
LTRIM (10.1.0.0)
MAXBYTES (9.0.0.0)
MAXFETCH (10.1.0.0)
MAXFETCH (9.0.0.0)
MONTHS_BETWEEN (9.0.0.0)
MULTIPATHHIER (9.0.0.0)
NEW_TIME (9.0.0.0)
NEXT_DAY (9.0.0.0)
NLS Options, specifically:

- NLS_CALENDAR (9.0.0.0)
- NLS_CURRENCY (9.0.0.0)
- NLS_DATE_FORMAT (9.0.0.0)
- NLS_DATE_LANGUAGE (9.0.0.0)
- NLS_DUAL_CURRENCY (9.0.0.0)
- NLS_ISO_CURRENCY (9.0.0.0)
- NLS_LANG (9.0.0.0)
- NLS_LANGUAGE (9.0.0.0)
- NLS_NUMERIC_CHARACTERS (9.0.0.0)
- NLS_SORT (9.0.0.0)
- NLS_TERRITORY (9.0.0.0)

NULLIF (10.1.0.0)
NULLIF (9.0.0.0)
NVL (10.1.0.0)
NVL2 (10.1.0.0)
ONATTACH (10.1.0.0)
PARTITIONCHECK (10.1.0.0)
POUTFILEUNIT (9.2.0.0)
RELEASE (10.1.0.0)
REMBYTES (9.0.0.0)
REPLBYTES (9.0.0.0)
RESYNC (10.1.0.0)
REVERT (10.1.0.0)
ROLE (9.0.0.0)
RPAD (10.1.0.0)
RTRIM (10.1.0.0)
SET1 (10.1.0.0)
SIGN (10.1.0.0)
SOURCEVAL (9.2.0.0)
SUBSTR (10.1.0.0)
SUBSTRB (10.1.0.0)
SYSDATE (9.0.0.0)
TO_CHAR (9.0.0.0)
TO_DATE (9.0.0.0)
TO_NCHAR (9.2.0.0)
TO_NUMBER (9.0.0.0)
TRACEFILEUNIT (9.2.0.0)
TRIGGER command (10.1.0.0)
TRIGGER function (10.1.0.0)
TRIGGER_DEFINE (10.1.0.0)
TRIGGER_AFTER_UPDATE (10.1.0.0)

TRIGGER_BEFORE_UPDATE (10.1.0.0)
TRIGGERASSIGN (10.1.0.0)
TRIM (10.1.0.0)
TRIM (9.0.0.0)
USERID (9.0.0.0)
USETRIGGERS (10.1.0.0)
VALUESET (10.1.0.0)
WIDTH_BUCKET (10.1.0.0)

Statements Deleted

The followingstatements have been deleted from the OLAP DML. The number in parentheses indicates the specific release in which the statement was deleted.

_UPDATEOLDVERS (9.2.0.0)
_XCALONGTIME (9.0.0.0)
_XCARETRIES (9.0.0.0)
_XCASHORTIME (9.0.0.0)
ALLOWQONS (9.2.0.0)
CACHEHITS (9.2.0.0)
CACHEMISSES (9.2.0.0)
CACHETRIES (9.2.0.0)
CHARSET (9.0.0.0)
CHDIR (9.2.0.0)
CHDRIVE (9.2.0.0)
COMQUERY (9.0.0.0)
COMSET (9.0.0.0)
COMUNIT (9.0.0.0)
CONNECT (9.0.0.0)
DBEXTENDPATH (9.2.0.0)
DBGSESSION (9.2.0.0)
DBREPORT (9.2.0.0)
DBSEARCHPATH (9.2.0.0)
DBTEMPPATH (9.2.0.0)
DEFINE EXTCALL (9.0.0.0)
DGCART (9.2.0.0)
DIR (9.2.0.0)
DISCONNECT (9.0.0.0)
EPRODUCT (9.2.0.0)
ERELEASE (9.2.0.0)
EXECBREAK (9.0.0.0)

EXECSTART (9.0.0.0)
EXECSTATUS (9.0.0.0)
EXECUTE (9.0.0.0)
EXECWAIT (9.0.0.0)
EXTARGS (9.0.0.0)
FETCH (9.0.0.0) -- SNAPI
FILEMODEMASK (9.2.0.0)
IFCOPY (9.2.0.0)
LONGOBJNAMES (9.0.0.0)
MAXFETCH (9.0.0.0)
MKDIR (9.0.0.0)
NAPAGEFREE (9.2.0.0)
ODBC.CONNECTION (9.0.0.0)
ODBC.CONNLIST (9.0.0.0)
ODBC.DISCONN (9.0.0.0)
ODBC.SOURCE (9.0.0.0)
ODBC.SOURCELIST (9.0.0.0)
PGCACHEHITS (9.2.0.0)
PGCACHEMISSES (9.2.0.0)
PAGEPAUSE (9.2.0.0)
PAGEPROMPT (9.2.0.0)
PAUSE (9.2.0.0)
RETRIEVE (9.0.0.0)
RMDIR (9.0.0.0)
SESSIONQUERY (9.0.0.0)
SHARESESSION (9.0.0.0)
SHELL (9.0.0.0)
SQL CONNECT (9.0.0.0)
SQL DISCONNECT (9.0.0.0)
SQL.DMBS (9.0.0.0)
SQL.DMBSLIST (9.0.0.0)
STRIP (9.2.0.0)
THREADEXTCALL (9.0.0.0)
TRACE (9.2.0.0)
TRANSLATE (9.0.0.0)
TRANSPORT (9.0.0.0)
WATCH (9.2.0.0)
XABORT (9.0.0.0)
XCAPORTNUMBER (9.0.0.0)
XCLOSE (9.0.0.0)
XOPEN (9.0.0.0)

Statements Significantly Changed

The following OLAP DML statements were significantly changed. Examples of significant changes are the addition of a new keyword or a change in a default value. The number in parentheses indicates the last release in which the statement was significantly changed. See also "Statements Renamed" on page B-7 for a list of renamed statements.

AGGMAPINFO (10.1.0.0)
ARGUMENT (10.1.0.0)
AW ATTACH (10.1.0.0)
CACHE (10.1.0.0)
CHGDFN (10.1.0.0)
CONVERT (9.2.0.0)
DECIMALCHAR (9.2.0.0)
DEFINE VARIABLE (10.1.0.0)
EXPORT (9.2.0.0)
FCQUERY (9.2.0.0)
FCSET (9.2.0.0)
FILEOPEN (9.0.0.0)
FILEQUERY (9.0.0.0)
FILEREAD (9.2.0.0)
HIERHEIGHT command (9.2.0.0)
IMPORT (9.0.0.0)
INFILE (9.0.0.0)
LAG (9.2.0.2)
LAGABSPCT (9.2.0.2)
LAGDIF (9.2.0.2)
LAGPCT (9.2.0.2)
LEAD (9.2.0.2)
LIMIT command (9.2.0.2)
LIMIT function (9.2.0.0)
MAINTAIN ADD SESSION (10.1.0.0)
MAINTAIN ADD TO PARTITION (10.1.0.0)
MODEL (9.2.0.2)
MOVINGAVERAGE (9.2.0.2)
MOVINGMAX (9.2.0.2)
MOVINGMIN (9.2.0.2)
MOVINGTOTAL (9.2.0.2)
NOSPELL (9.2.0.0)
OBJ (10.1.0.0)
OUTFILE (9.0.0.0)

PROGRAM (9.2.0.0)
PROPERTY (9.0.0.0)
RECURSIVE (9.0.0.0)
RELATION (for aggregation) (9.2.0.2)
RELATION (for allocation) (9.2.0.2)
ROUND (9.0.0.0)
SQL (10.1.0.0)
SYSDATE (9.2.0.0)
SYSINFO (9.2.0.2)
SYSTEM (9.2.0.0)
THOUSANDSCHAR (9.2.0.0)
UPDATE (10.1.0.0)
VARIABLE (10.1.0.0)
VALSPERPAGE (10.1.0.0)
YESSPELL (9.2.0.0)

Statements Renamed

The following OLAP DML statements have been renamed. The number in parentheses indicates the specific release in which the statement was renamed.

DATABASE command to AW command (9.2.0.0)
DATABASE function to AW function (9.2.0.0)
DBDESCRIBE to AWDESCRIBE (9.2.0.0)
DBWAITTIME to AWWAITTIME (9.2.0.0)
DEFAULTDBFSIZE to DECODE (9.2.0.0)
OESEIFVERSION to EIFVERSION (9.2.0.0)

Index