Oracle® Database

Data Warehousing Guide

10*g* Release 1 (10.1)

Part No. B10736-01

December 2003

ORACLE®

Oracle Database Data Warehousing Guide, 10*g* Release 1 (10.1)

Part No. B10736-01

Primary Author: Paul Lane

Contributing Authors: Viv Schupmann, Ingrid Stuart (Change Data Capture)

Contributors: Patrick Amor, Hermann Baer, Mark Bauer, Subhransu Basu, Srikanth Bellamkonda, Randy Bello, Tolga Bozkaya, Lucy Burgess, Rushan Chen, Benoit Dageville, John Haydu, Lilian Hobbs, Hakan Jakobsson, George Lumpkin, Alex Melidis, Valarie Moore, Cetin Ozbutun, Ananth Raghavan, Jack Raitto, Ray Roccaforte, Sankar Subramanian, Gregory Smith, Murali Thiyagarajan, Ashish Thusoo, Thomas Tong, Jean-Francois Verrier, Gary Vincent, Andreas Walter, Andy Witkowski, Min Xiao, Tsae-Feng Yu

Contents

8 Basic Materialized Views

9 Advanced Materialized Views

12 Extraction in Data Warehouses

13 Transportation in Data Warehouses

14 Loading and Transformation

15 Maintaining the Data Warehouse

16 Change Data Capture

17 SQLAccess Advisor

Volume 2

Part V Data Warehouse Performance

18 Query Rewrite

19 Schema Modeling Techniques

20 SQL for Aggregation in Data Warehouses

21 SQL for Analysis and Reporting

22 SQL for Modeling

23 OLAP and Data Mining

24 Using Parallel Execution

Send Us Your Comments

Oracle Database Data Warehousing Guide, 10*g* Release 1 (10.1)

Part No. B10736-01

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: (650)506-7227. Attn: Server Technologies Documentation Manager
- Postal service:
  Oracle Corporation
  Server Technologies Documentation
  500 Oracle Parkway, Mailstop 4op11
  Redwood Shores, CA 94065
  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

This manual provides information about Oracle's data warehousing capabilities.

This preface contains these topics:

Note: The *Oracle Data Warehousing Guide* contains information that describes the features and functionality of the Oracle Database Standard Edition, Oracle Database Enterprise Edition, and Oracle Database Personal Edition products. These products have the same basic features. However, several advanced features are available only with the Oracle Database Enterprise Edition or Oracle Database Personal Edition, and some of these are optional. For example, to create partitioned tables and indexes, you must have the Oracle Database Enterprise Edition or Oracle Database Personal Edition.

Audience

This guide is intended for database administrators, system administrators, and database application developers who design, maintain, and use data warehouses.

To use this document, you need to be familiar with relational database concepts, basic Oracle server concepts, and the operating system environment under which you are running Oracle.

Organization

This document contains:

Part 1: Concepts

Chapter 1, "Data Warehousing Concepts"

This chapter contains an overview of data warehousing concepts.

Part 2: Logical Design

Chapter 2, "Logical Design in Data Warehouses"

This chapter discusses the logical design of a data warehouse.

Part 3: Physical Design

Chapter 3, "Physical Design in Data Warehouses"

This chapter discusses the physical design of a data warehouse.

Chapter 4, "Hardware and I/O Considerations in Data Warehouses"

This chapter describes hardware, input-output, and storage considerations.

Chapter 5, "Parallelism and Partitioning in Data Warehouses"

This chapter describes the basics of parallelism and partitioning in data warehouses.

Chapter 6, "Indexes"

This chapter describes how to use indexes in data warehouses.

Chapter 7, "Integrity Constraints"
This chapter describes how to use integrity constraints in data warehouses.

Chapter 8, "Basic Materialized Views"
This chapter introduces basic materialized views concepts.

Chapter 9, "Advanced Materialized Views"
This chapter describes how to use materialized views in data warehouses.

Chapter 10, "Dimensions"
This chapter describes how to use dimensions in data warehouses.

Part 4: Managing the Data Warehouse Environment

Chapter 11, "Overview of Extraction, Transformation, and Loading"
This chapter is an overview of the ETL process.

Chapter 12, "Extraction in Data Warehouses"
This chapter describes extraction issues.

Chapter 13, "Transportation in Data Warehouses"
This chapter describes transporting data in data warehouses.

Chapter 14, "Loading and Transformation"
This chapter describes transforming and loading data in data warehouses.

Chapter 15, "Maintaining the Data Warehouse"
This chapter describes how to refresh a data warehouse.

Chapter 16, "Change Data Capture"
This chapter describes how to use Change Data Capture capabilities.

Chapter 17, "SQLAccess Advisor"
This chapter describes how to use the SQLAccess Advisor.

Part 5: Data Warehouse Performance

Chapter 18, "Query Rewrite"
This chapter describes how to use query rewrite.

Chapter 19, "Schema Modeling Techniques"
This chapter describes the schemas useful in data warehousing environments.

Chapter 20, "SQL for Aggregation in Data Warehouses"
This chapter explains how to use SQL aggregation in data warehouses.

Chapter 21, "SQL for Analysis and Reporting"
This chapter explains how to use analytic functions in data warehouses.

Chapter 22, "SQL for Modeling"
This chapter explains how to use the spreadsheet clause for SQL modeling.

Chapter 23, "OLAP and Data Mining"
This chapter describes using analytic services and data mining in combination with Oracle Database10*g*.

Chapter 24, "Using Parallel Execution"
This chapter describes how to tune data warehouses using parallel execution.

Glossary
The glossary defines important terms used in this guide.

Related Documentation

For more information, see these Oracle resources:

- *Oracle Database Performance Tuning Guide*

Many of the examples in this book use the sample schemas of the seed database, which is installed by default when you install Oracle. Refer to *Oracle Database Sample Schemas* for information on how these schemas were created and how you can use them yourself.

Printed documentation is available for sale in the Oracle Store at

http://oraclestore.oracle.com/

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at

http://otn.oracle.com/membership/

If you already have a username and password for OTN, then you can go directly to the documentation section of the OTN Web site at

http://otn.oracle.com/documentation

For additional information, see:

- *The Data Warehouse Toolkit* by Ralph Kimball (John Wiley and Sons, 1996)
- *Building the Data Warehouse* by William Inmon (John Wiley and Sons, 1996)

Conventions

This section describes the conventions used in the text and code examples of this documentation set. It describes:

- Conventions in Text
- Conventions in Code Examples

Conventions in Text

We use various conventions in text to help you more quickly identify special terms. The following table describes those conventions and provides examples of their use.

Convention	Meaning	Example
Bold	Bold typeface indicates terms that are defined in the text or terms that appear in a glossary, or both.	When you specify this clause, you create an **index-organized table**.
Italics	Italic typeface indicates book titles or emphasis.	*Oracle Database Concepts* Ensure that the recovery catalog and target database do *not* reside on the same disk.

Convention	Meaning	Example
UPPERCASE monospace (fixed-width) font	Uppercase monospace typeface indicates elements supplied by the system. Such elements include parameters, privileges, datatypes, RMAN keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, usernames, and roles.	You can specify this clause only for a NUMBER column. You can back up the database by using the BACKUP command. Query the TABLE_NAME column in the USER_TABLES data dictionary view. Use the DBMS_STATS.GENERATE_STATS procedure.
lowercase monospace (fixed-width) font	Lowercase monospace typeface indicates executables, filenames, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, usernames and roles, program units, and parameter values. **Note:** Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	Enter sqlplus to open SQL*Plus. The password is specified in the orapwd file. Back up the datafiles and control files in the /disk1/oracle/dbs directory. The department_id, department_name, and location_id columns are in the hr.departments table. Set the QUERY_REWRITE_ENABLED initialization parameter to TRUE. Connect as oe user. The JRepUtil class implements these methods.
lowercase italic monospace (fixed-width) font	Lowercase italic monospace font represents placeholders or variables.	You can specify the *parallel_clause*. Run U*old_release*.SQL where *old_release* refers to the release you installed prior to upgrading.

Conventions in Code Examples

Code examples illustrate SQL, PL/SQL, SQL*Plus, or other command-line statements. They are displayed in a monospace (fixed-width) font and separated from normal text as shown in this example:

```
SELECT username FROM dba_users WHERE username = 'MIGRATE';
```

The following table describes typographic conventions used in code examples and provides examples of their use.

Convention	Meaning	Example
[ ]	Brackets enclose one or more optional items. Do not enter the brackets.	DECIMAL (*digits* [, *precision*])
{ }	Braces enclose two or more items, one of which is required. Do not enter the braces.	{ENABLE | DISABLE}
|	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	{ENABLE | DISABLE} [COMPRESS | NOCOMPRESS]
...	Horizontal ellipsis points indicate either: ■ That we have omitted parts of the code that are not directly related to the example ■ That you can repeat a portion of the code	CREATE TABLE ... AS *subquery*; SELECT *col1*, *col2*, ... , *coln* FROM employees;
. . .	Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example.	SQL> SELECT NAME FROM V$DATAFILE; NAME ---------- /fsl/dbs/tbs_01.dbf /fs1/dbs/tbs_02.dbf . . . /fsl/dbs/tbs_09.dbf 9 rows selected.
Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as shown.	acctbal NUMBER(11,2); acct CONSTANT NUMBER(4) := 3;
Italics	Italicized text indicates placeholders or variables for which you must supply particular values.	CONNECT SYSTEM/*system_password* DB_NAME = *database_name*
UPPERCASE	Uppercase typeface indicates elements supplied by the system. We show these terms in uppercase in order to distinguish them from terms you define. Unless terms appear in brackets, enter them in the order and with the spelling shown. However, because these terms are not case sensitive, you can enter them in lowercase.	SELECT last_name, employee_id FROM employees; SELECT * FROM USER_TABLES; DROP TABLE hr.employees;

Convention	Meaning	Example

lowercase	Lowercase typeface indicates programmatic elements that you supply. For example, lowercase indicates names of tables, columns, or files.
	
	Note: Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.
		`SELECT last_name, employee_id FROM employees;`
		`sqlplus hr/hr`
		`CREATE USER mjones IDENTIFIED BY ty3MU9;`

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.

What's New in Oracle Database?

This section describes the new features of Oracle Database 10*g* Release 1 (10.1) and provides pointers to additional information. New features information from previous releases is also retained to help those users migrating to the current release.

The following section describse new features in Oracle Database:

- Oracle Database 10g Release 1 (10.1) New Features in Data Warehousing

Oracle Database 10*g* Release 1 (10.1) New Features in Data Warehousing

- **SQL Model Calculations**

  The `MODEL` clause enables you to specify complex formulas while avoiding multiple joins and `UNION` clauses. This clause supports OLAP queries such as share of ancestor and prior period comparisons, as well as calculations typically done in large spreadsheets. The `MODEL` clause provides building blocks for budgeting, forecasting, and statistical applications.

  See Also: Chapter 22, "SQL for Modeling"

- **SQLAccess Advisor**

  The SQLAccess Advisor tool and its related `DBMS_ADVISOR` package offer improved capabilities for recommending indexing and materialized view strategies.

  See Also: Chapter 17, "SQLAccess Advisor"

- **Materialized Views**

  The `TUNE_MVIEW` procedure shows how to specify a materialized view so that it is fast refreshable and can use advanced types of query rewrite.

- **Materialized View Refresh Enhancements**

  Materialized view refresh has new optimizations for data warehousing and OLAP environments. The enhancements include more efficient calculation and update techniques, support for nested refresh, along with improved cost analysis.

  See Also: Chapter 15, "Maintaining the Data Warehouse"

- **Query Rewrite Enhancements**

  Query rewrite performance and capabilities have been improved.

  See Also: Chapter 18, "Query Rewrite"

- **Partitioning Enhancements**

You can now use partitioning with index-organized tables. Also, materialized views in OLAP are able to use partitioning. You can now use hash-partitioned global indexes.

> **See Also:** Chapter 5, "Parallelism and Partitioning in Data Warehouses"

- **Change Data Capture**

  Oracle now supports asynchronous change data capture as well as synchronous change data capture.

  > **See Also:** Chapter 16, "Change Data Capture"

- **ETL Enhancements**

  Oracle's extraction, transformation, and loading capabilities have been improved with several `MERGE` improvements and better external table capabilities.

  > **See Also:** Chapter 11, "Overview of Extraction, Transformation, and Loading"

- **Storage Management**

  Oracle Managed Files has simplified the administration of a database by providing functionality to automatically create and manage files, so the database administrator no longer needs to manage each database file.

  Automatic Storage Management provides additional functionality for managing not only files, but also the disks. In addition, you can now use ultralarge data files.

  > **See Also:** Chapter 4, "Hardware and I/O Considerations in Data Warehouses"

Part I

Concepts

This section introduces basic data warehousing concepts.

It contains the following chapter:

- Chapter 1, "Data Warehousing Concepts"

1

Data Warehousing Concepts

This chapter provides an overview of the Oracle data warehousing implementation. It includes:

- What is a Data Warehouse?
- Data Warehouse Architectures

Note that this book is meant as a supplement to standard texts about data warehousing. This book focuses on Oracle-specific material and does not reproduce in detail material of a general nature. Two standard texts are:

- *The Data Warehouse Toolkit* by Ralph Kimball (John Wiley and Sons, 1996)
- *Building the Data Warehouse* by William Inmon (John Wiley and Sons, 1996)

What is a Data Warehouse?

A data warehouse is a relational database that is designed for query and analysis rather than for transaction processing. It usually contains historical data derived from transaction data, but can include data from other sources. Data warehouses separate analysis workload from transaction workload and enable an organization to consolidate data from several sources.

In addition to a relational database, a data warehouse environment includes an extraction, transportation, transformation, and loading (ETL) solution, online analytical processing (OLAP) and data mining capabilities, client analysis tools, and other applications that manage the process of gathering data and delivering it to business users.

> **See Also:** Chapter 11, "Overview of Extraction, Transformation, and Loading"

A common way of introducing data warehousing is to refer to the characteristics of a data warehouse as set forth by William Inmon:

- Subject Oriented
- Integrated
- Nonvolatile
- Time Variant

Subject Oriented

Data warehouses are designed to help you analyze data. For example, to learn more about your company's sales data, you can build a data warehouse that concentrates on sales. Using this data warehouse, you can answer questions such as "Who was our best customer for this item last year?" This ability to define a data warehouse by subject matter, sales in this case, makes the data warehouse subject oriented.

Integrated

Integration is closely related to subject orientation. Data warehouses must put data from disparate sources into a consistent format. They must resolve such problems as naming conflicts and inconsistencies among units of measure. When they achieve this, they are said to be integrated.

Nonvolatile

Nonvolatile means that, once entered into the data warehouse, data should not change. This is logical because the purpose of a data warehouse is to enable you to analyze what has occurred.

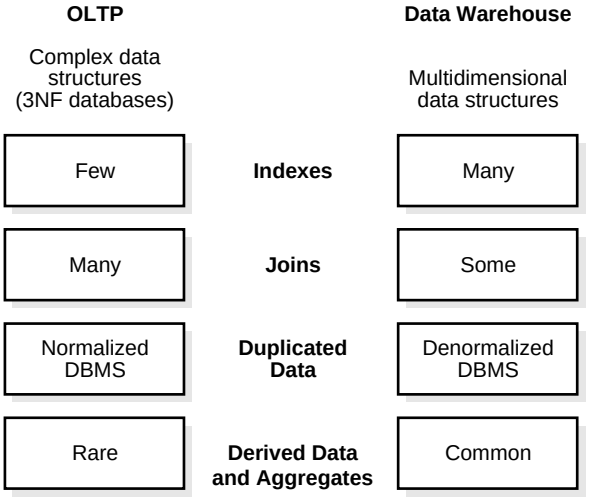
Time Variant

In order to discover trends in business, analysts need large amounts of data. This is very much in contrast to **online transaction processing (OLTP)** systems, where performance requirements demand that historical data be moved to an archive. A data warehouse's focus on change over time is what is meant by the term time variant.

Contrasting OLTP and Data Warehousing Environments

Figure 1–1 illustrates key differences between an OLTP system and a data warehouse.

Figure 1–1 Contrasting OLTP and Data Warehousing Environments



One major difference between the types of system is that data warehouses are not usually in **third normal form (3NF)**, a type of data normalization common in OLTP environments.

Data warehouses and OLTP systems have very different requirements. Here are some examples of differences between typical data warehouses and OLTP systems:

- Workload

  Data warehouses are designed to accommodate *ad hoc* queries. You might not know the workload of your data warehouse in advance, so a data warehouse should be optimized to perform well for a wide variety of possible query operations.

  OLTP systems support only predefined operations. Your applications might be specifically tuned or designed to support only these operations.

- Data modifications

  A data warehouse is updated on a regular basis by the ETL process (run nightly or weekly) using bulk data modification techniques. The end users of a data warehouse do not directly update the data warehouse.

  In OLTP systems, end users routinely issue individual data modification statements to the database. The OLTP database is always up to date, and reflects the current state of each business transaction.

- Schema design

  Data warehouses often use denormalized or partially denormalized schemas (such as a star schema) to optimize query performance.

  OLTP systems often use fully normalized schemas to optimize update/insert/delete performance, and to guarantee data consistency.

- Typical operations

  A typical data warehouse query scans thousands or millions of rows. For example, "Find the total sales for all customers last month."

  A typical OLTP operation accesses only a handful of records. For example, "Retrieve the current order for this customer."

- Historical data

  Data warehouses usually store many months or years of data. This is to support historical analysis.

  OLTP systems usually store data from only a few weeks or months. The OLTP system stores only historical data as needed to successfully meet the requirements of the current transaction.

Data Warehouse Architectures

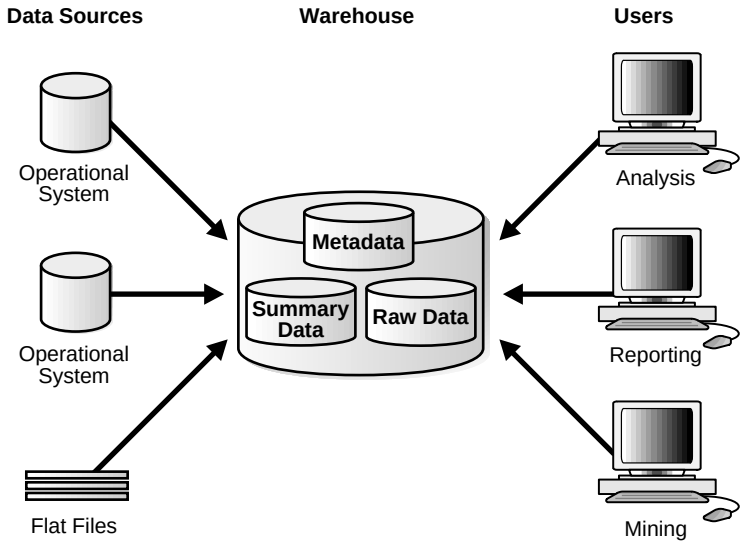
Data warehouses and their architectures vary depending upon the specifics of an organization's situation. Three common architectures are:

- Data Warehouse Architecture (Basic)
- Data Warehouse Architecture (with a Staging Area)
- Data Warehouse Architecture (with a Staging Area and Data Marts)

Data Warehouse Architecture (Basic)

Figure 1–2 shows a simple architecture for a data warehouse. End users directly access data derived from several source systems through the data warehouse.

Figure 1–2 Architecture of a Data Warehouse

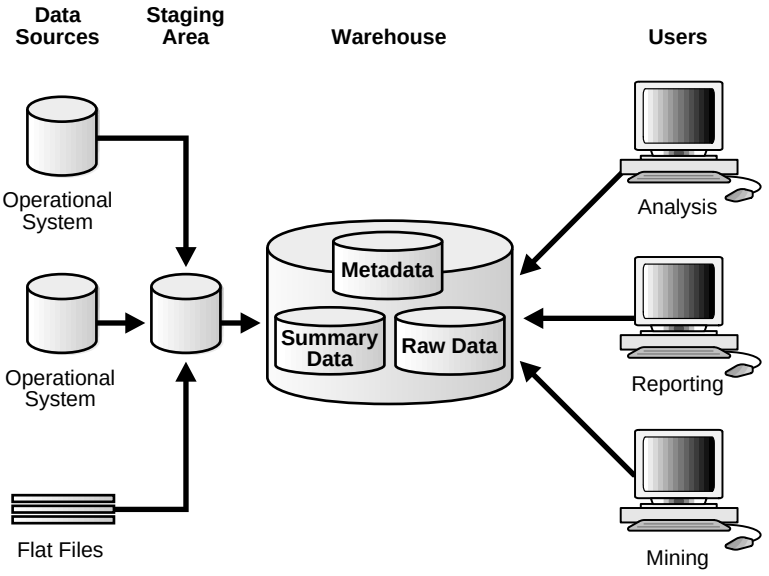


In Figure 1–2, the metadata and raw data of a traditional OLTP system is present, as is an additional type of data, summary data. Summaries are very valuable in data warehouses because they pre-compute long operations in advance. For example, a typical data warehouse query is to retrieve something such as August sales. A summary in an Oracle database is called a **materialized view**.

Data Warehouse Architecture (with a Staging Area)

In Figure 1–2, you need to clean and process your operational data before putting it into the warehouse. You can do this programmatically, although most data warehouses use a **staging area** instead. A staging area simplifies building summaries and general warehouse management. Figure 1–3 illustrates this typical architecture.

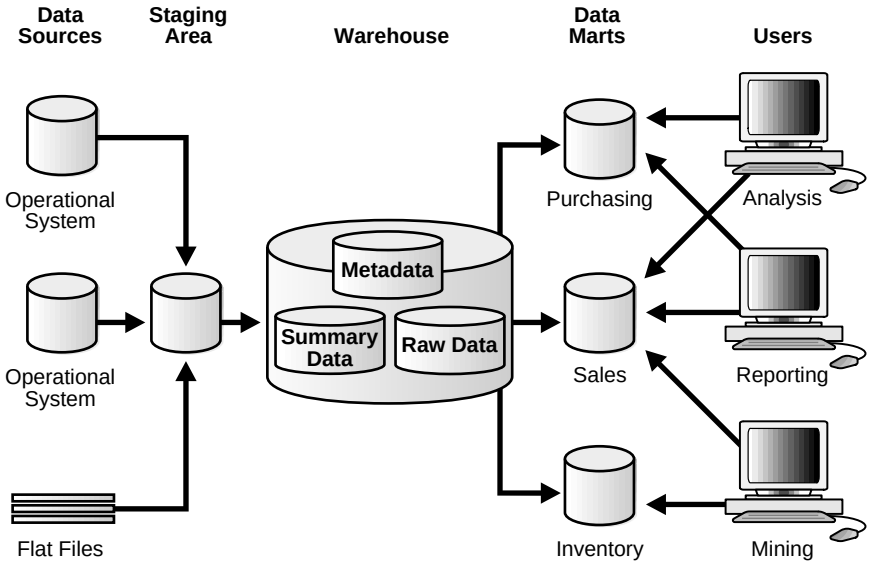
Figure 1–3 Architecture of a Data Warehouse with a Staging Area



Data Warehouse Architecture (with a Staging Area and Data Marts)

Although the architecture in Figure 1–3 is quite common, you may want to customize your warehouse's architecture for different groups within your organization. You can do this by adding **data marts**, which are systems designed for a particular line of business. Figure 1–4 illustrates an example where purchasing, sales, and inventories are separated. In this example, a financial analyst might want to analyze historical data for purchases and sales.

Figure 1–4 Architecture of a Data Warehouse with a Staging Area and Data Marts



Note: Data marts are an important part of many data warehouses, but they are not the focus of this book.

See Also: *Data Mart Suites* documentation for further information regarding data marts

Part II

Logical Design

This section deals with the issues in logical design in a data warehouse.

It contains the following chapter:

2

Logical Design in Data Warehouses

This chapter explains how to create a logical design for a data warehousing environment and includes the following topics:

Logical Versus Physical Design in Data Warehouses

Your organization has decided to build a data warehouse. You have defined the business requirements and agreed upon the scope of your application, and created a conceptual design. Now you need to translate your requirements into a system deliverable. To do so, you create the logical and physical design for the data warehouse. You then define:

- The specific data content
- Relationships within and between groups of data
- The system environment supporting your data warehouse
- The data transformations required
- The frequency with which data is refreshed

The logical design is more conceptual and abstract than the physical design. In the logical design, you look at the logical relationships among the objects. In the physical design, you look at the most effective way of storing and retrieving the objects as well as handling them from a transportation and backup / recovery perspective.

Orient your design toward the needs of the end users. End users typically want to perform analysis and look at aggregated data, rather than at individual transactions. However, end users might not know what they need until they see it. In addition, a well-planned design allows for growth and changes as the needs of users change and evolve.

By beginning with the logical design, you focus on the information requirements and save the implementation details for later.

Creating a Logical Design

A logical design is conceptual and abstract. You do not deal with the physical implementation details yet. You deal only with defining the types of information that you need.

One technique you can use to model your organization's logical information requirements is entity-relationship modeling. Entity-relationship modeling involves identifying the things of importance (entities), the properties of these things (attributes), and how they are related to one another (relationships).

The process of logical design involves arranging data into a series of logical relationships called entities and attributes. An **entity** represents a chunk of

information. In relational databases, an entity often maps to a table. An **attribute** is a component of an entity that helps define the uniqueness of the entity. In relational databases, an attribute maps to a column.

To be sure that your data is consistent, you need to use unique identifiers. A **unique identifier** is something you add to tables so that you can differentiate between the same item when it appears in different places. In a physical design, this is usually a primary key.

While entity-relationship diagramming has traditionally been associated with highly normalized models such as OLTP applications, the technique is still useful for data warehouse design in the form of dimensional modeling. In dimensional modeling, instead of seeking to discover atomic units of information (such as entities and attributes) and all of the relationships between them, you identify which information belongs to a central fact table and which information belongs to its associated dimension tables. You identify business subjects or fields of data, define relationships between business subjects, and name the attributes for each subject.

> **See Also:** Chapter 10, "Dimensions" for further information regarding dimensions

Your logical design should result in (1) a set of entities and attributes corresponding to fact tables and dimension tables and (2) a model of operational data from your source into subject-oriented information in your target data warehouse schema.

You can create the logical design using a pen and paper, or you can use a design tool such as Oracle Warehouse Builder (specifically designed to support modeling the ETL process) or Oracle Designer (a general purpose modeling tool).

> **See Also:** Oracle Designer and Oracle Warehouse Builder documentation sets

Data Warehousing Schemas

A schema is a collection of database objects, including tables, views, indexes, and synonyms. You can arrange schema objects in the schema models designed for data warehousing in a variety of ways. Most data warehouses use a dimensional model.

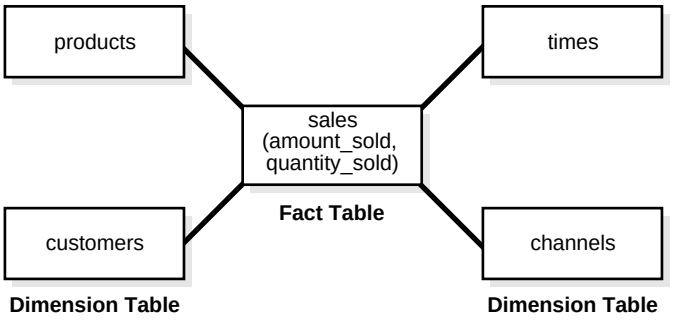
The model of your source data and the requirements of your users help you design the data warehouse schema. You can sometimes get the source model from your company's enterprise data model and reverse-engineer the logical data model for the data warehouse from this. The physical implementation of the logical data

warehouse model may require some changes to adapt it to your system parameters—size of machine, number of users, storage capacity, type of network, and software.

Star Schemas

The star schema is the simplest data warehouse schema. It is called a star schema because the diagram resembles a star, with points radiating from a center. The center of the star consists of one or more fact tables and the points of the star are the dimension tables, as shown in Figure 2–1.

Figure 2–1 Star Schema



The most natural way to model a data warehouse is as a star schema, where only one join establishes the relationship between the fact table and any one of the dimension tables.

A star schema optimizes performance by keeping queries simple and providing fast response time. All the information about each level is stored in one row.

Note: Oracle Corporation recommends that you choose a star schema unless you have a clear reason not to.

Other Schemas

Some schemas in data warehousing environments use third normal form rather than star schemas. Another schema that is sometimes useful is the snowflake schema, which is a star schema with normalized dimensions in a tree structure.

See Also: Chapter 19, "Schema Modeling Techniques" for further information regarding star and snowflake schemas in data warehouses and *Oracle Database Concepts* for further conceptual material

Data Warehousing Objects

Fact tables and dimension tables are the two types of objects commonly used in dimensional data warehouse schemas.

Fact tables are the large tables in your data warehouse schema that store business measurements. Fact tables typically contain facts and foreign keys to the dimension tables. Fact tables represent data, usually numeric and additive, that can be analyzed and examined. Examples include `sales`, `cost`, and `profit`.

Dimension tables, also known as lookup or reference tables, contain the relatively static data in the data warehouse. Dimension tables store the information you normally use to contain queries. Dimension tables are usually textual and descriptive and you can use them as the row headers of the result set. Examples are `customers` or `products`.

Fact Tables

A fact table typically has two types of columns: those that contain numeric facts (often called measurements), and those that are foreign keys to dimension tables. A fact table contains either detail-level facts or facts that have been aggregated. Fact tables that contain aggregated facts are often called summary tables. A fact table usually contains facts with the same level of aggregation. Though most facts are additive, they can also be semi-additive or non-additive. Additive facts can be aggregated by simple arithmetical addition. A common example of this is sales. Non-additive facts cannot be added at all. An example of this is averages. Semi-additive facts can be aggregated along some of the dimensions and not along others. An example of this is inventory levels, where you cannot tell what a level means simply by looking at it.

Creating a New Fact Table

You must define a fact table for each star schema. From a modeling standpoint, the primary key of the fact table is usually a composite key that is made up of all of its foreign keys.

Dimension Tables

A dimension is a structure, often composed of one or more hierarchies, that categorizes data. Dimensional attributes help to describe the dimensional value. They are normally descriptive, textual values. Several distinct dimensions, combined with facts, enable you to answer business questions. Commonly used dimensions are customers, products, and time.

Dimension data is typically collected at the lowest level of detail and then aggregated into higher level totals that are more useful for analysis. These natural rollups or aggregations within a dimension table are called hierarchies.

Hierarchies

Hierarchies are logical structures that use ordered levels as a means of organizing data. A hierarchy can be used to define data aggregation. For example, in a time dimension, a hierarchy might aggregate data from the month level to the quarter level to the year level. A hierarchy can also be used to define a navigational drill path and to establish a family structure.

Within a hierarchy, each level is logically connected to the levels above and below it. Data values at lower levels aggregate into the data values at higher levels. A dimension can be composed of more than one hierarchy. For example, in the product dimension, there might be two hierarchies—one for product categories and one for product suppliers.

Dimension hierarchies also group levels from general to granular. Query tools use hierarchies to enable you to drill down into your data to view different levels of granularity. This is one of the key benefits of a data warehouse.

When designing hierarchies, you must consider the relationships in business structures. For example, a divisional multilevel sales organization.

Hierarchies impose a family structure on dimension values. For a particular level value, a value at the next higher level is its parent, and values at the next lower level are its children. These familial relationships enable analysts to access data quickly.

Levels A level represents a position in a hierarchy. For example, a time dimension might have a hierarchy that represents data at the month, quarter, and year levels. Levels range from general to specific, with the root level as the highest or most general level. The levels in a dimension are organized into one or more hierarchies.

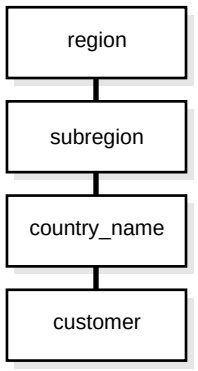
Level Relationships Level relationships specify top-to-bottom ordering of levels from most general (the root) to most specific information. They define the parent-child relationship between the levels in a hierarchy.

Hierarchies are also essential components in enabling more complex rewrites. For example, the database can aggregate an existing sales revenue on a quarterly base to a yearly aggregation when the dimensional dependencies between quarter and year are known.

Typical Dimension Hierarchy

Figure 2–2 illustrates a dimension hierarchy based on `customers`.

Figure 2–2 Typical Levels in a Dimension Hierarchy



See Also: Chapter 10, "Dimensions" and Chapter 18, "Query Rewrite" for further information regarding hierarchies

Unique Identifiers

Unique identifiers are specified for one distinct record in a dimension table. Artificial unique identifiers are often used to avoid the potential problem of unique identifiers changing. Unique identifiers are represented with the # character. For example, `#customer_id`.

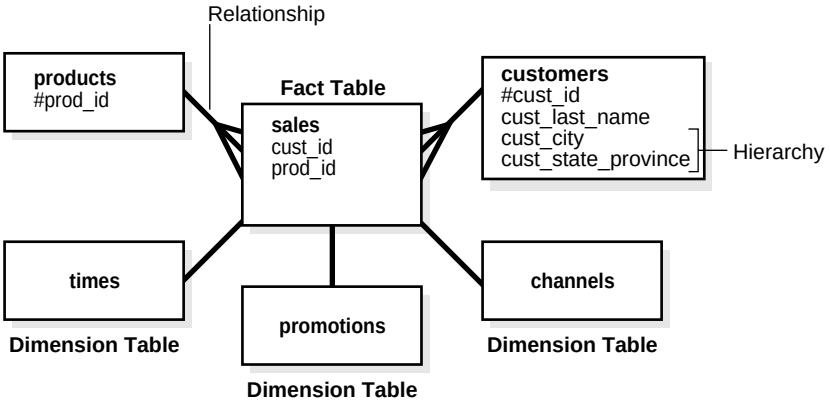
Relationships

Relationships guarantee business integrity. An example is that if a business sells something, there is obviously a customer and a product. Designing a relationship between the sales information in the fact table and the dimension tables products and customers enforces the business rules in databases.

Example of Data Warehousing Objects and Their Relationships

Figure 2–3 illustrates a common example of a `sales` fact table and dimension tables `customers`, `products`, `promotions`, `times`, and `channels`.

Figure 2–3 Typical Data Warehousing Objects



Part III

Physical Design

This section deals with the physical design of a data warehouse.

It contains the following chapters:

3

Physical Design in Data Warehouses

This chapter describes the physical design of a data warehousing environment, and includes the following topics:

Moving from Logical to Physical Design

Logical design is what you draw with a pen and paper or design with Oracle Warehouse Builder or Oracle Designer before building your data warehouse. Physical design is the creation of the database with SQL statements.

During the physical design process, you convert the data gathered during the logical design phase into a description of the physical database structure. Physical design decisions are mainly driven by query performance and database maintenance aspects. For example, choosing a partitioning strategy that meets common query requirements enables Oracle Database to take advantage of partition pruning, a way of narrowing a search before performing it.

See Also:

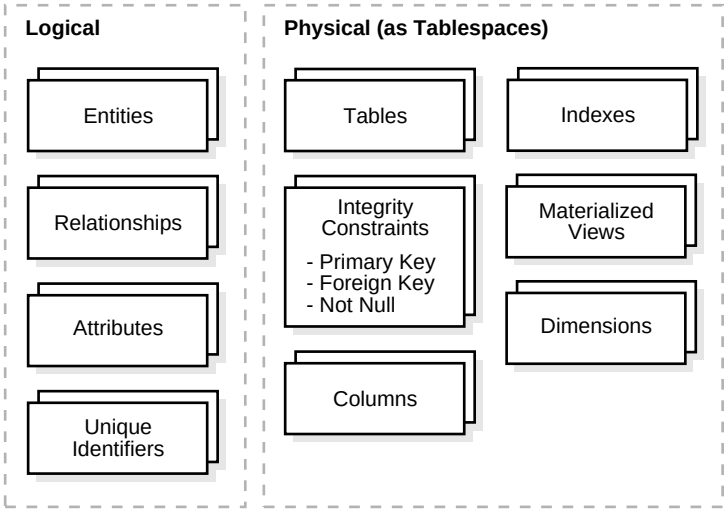
- Chapter 5, "Parallelism and Partitioning in Data Warehouses" for further information regarding partitioning
- *Oracle Database Concepts* for further conceptual material regarding all design matters

Physical Design

During the logical design phase, you defined a model for your data warehouse consisting of entities, attributes, and relationships. The entities are linked together using relationships. Attributes are used to describe the entities. The **unique identifier** (UID) distinguishes between one instance of an entity and another.

Figure 3–1 illustrates a graphical way of distinguishing between logical and physical designs.

Figure 3–1 Logical Design Compared with Physical Design



During the physical design process, you translate the expected schemas into actual database structures. At this time, you have to map:

- Entities to tables
- Relationships to foreign key constraints
- Attributes to columns
- Primary unique identifiers to primary key constraints
- Unique identifiers to unique key constraints

Physical Design Structures

Once you have converted your logical design to a physical one, you will need to create some or all of the following structures:

- Tablespaces
- Tables and Partitioned Tables
- Views
- Integrity Constraints
- Dimensions

Some of these structures require disk space. Others exist only in the data dictionary. Additionally, the following structures may be created for performance improvement:

- Indexes and Partitioned Indexes
- Materialized Views

Tablespaces

A tablespace consists of one or more datafiles, which are physical structures within the operating system you are using. A datafile is associated with only one tablespace. From a design perspective, tablespaces are containers for physical design structures.

Tablespaces need to be separated by differences. For example, tables should be separated from their indexes and small tables should be separated from large tables. Tablespaces should also represent logical business units if possible. Because a tablespace is the coarsest granularity for backup and recovery or the transportable tablespaces mechanism, the logical business design affects availability and maintenance operations.

You can now use ultralarge data files, a significant improvement in very large databases.

> **See Also:** Chapter 4, "Hardware and I/O Considerations in Data Warehouses" for information regarding tablespaces

Tables and Partitioned Tables

Tables are the basic unit of data storage. They are the container for the expected amount of raw data in your data warehouse.

Using partitioned tables instead of nonpartitioned ones addresses the key problem of supporting very large data volumes by allowing you to decompose them into smaller and more manageable pieces. The main design criterion for partitioning is manageability, though you will also see performance benefits in most cases because of partition pruning or intelligent parallel processing. For example, you might choose a partitioning strategy based on a sales transaction date and a monthly granularity. If you have four years' worth of data, you can delete a month's data as it becomes older than four years with a single, fast DDL statement and load new data while only affecting 1/48th of the complete table. Business questions regarding the last quarter will only affect three months, which is equivalent to three partitions, or 3/48ths of the total volume.

Partitioning large tables improves performance because each partitioned piece is more manageable. Typically, you partition based on transaction dates in a data warehouse. For example, each month, one month's worth of data can be assigned its own partition.

Table Compression

You can save disk space by compressing heap-organized tables. A typical type of heap-organized table you should consider for table compression is partitioned tables.

To reduce disk use and memory use (specifically, the buffer cache), you can store tables and partitioned tables in a compressed format inside the database. This often leads to a better scaleup for read-only operations. Table compression can also speed up query execution. There is, however, a cost in CPU overhead.

Table compression should be used with highly redundant data, such as tables with many foreign keys. You should avoid compressing tables with much update or other DML activity. Although compressed tables or partitions are updatable, there is some overhead in updating these tables, and high update activity may work against compression by causing some space to be wasted.

> **See Also:** Chapter 5, "Parallelism and Partitioning in Data Warehouses" and Chapter 15, "Maintaining the Data Warehouse"

Views

A view is a tailored presentation of the data contained in one or more tables or other views. A view takes the output of a query and treats it as a table. Views do not require any space in the database.

> **See Also:** *Oracle Database Concepts*

Integrity Constraints

Integrity constraints are used to enforce business rules associated with your database and to prevent having invalid information in the tables. Integrity constraints in data warehousing differ from constraints in OLTP environments. In OLTP environments, they primarily prevent the insertion of invalid data into a record, which is not a big problem in data warehousing environments because accuracy has already been guaranteed. In data warehousing environments, constraints are only used for query rewrite. `NOT NULL` constraints are particularly common in data warehouses. Under some specific circumstances, constraints need

space in the database. These constraints are in the form of the underlying unique index.

See Also: Chapter 7, "Integrity Constraints"

Indexes and Partitioned Indexes

Indexes are optional structures associated with tables or clusters. In addition to the classical B-tree indexes, bitmap indexes are very common in data warehousing environments. Bitmap indexes are optimized index structures for set-oriented operations. Additionally, they are necessary for some optimized data access methods such as star transformations.

Indexes are just like tables in that you can partition them, although the partitioning strategy is not dependent upon the table structure. Partitioning indexes makes it easier to manage the data warehouse during refresh and improves query performance.

See Also: Chapter 6, "Indexes" and Chapter 15, "Maintaining the Data Warehouse"

Materialized Views

Materialized views are query results that have been stored in advance so long-running calculations are not necessary when you actually execute your SQL statements. From a physical design point of view, materialized views resemble tables or partitioned tables and behave like indexes in that they are used transparently and improve performance.

See Also: Chapter 10, "Dimensions" Chapter 8, "Basic Materialized Views"

Dimensions

A dimension is a schema object that defines hierarchical relationships between columns or column sets. A hierarchical relationship is a functional dependency from one level of a hierarchy to the next one. A dimension is a container of logical relationships and does not require any space in the database. A typical dimension is city, state (or province), region, and country.

See Also: Chapter 10, "Dimensions"

4

Hardware and I/O Considerations in Data Warehouses

This chapter explains some of the hardware and I/O issues in a data warehousing environment and includes the following topics:

- Overview of Hardware and I/O Considerations in Data Warehouses
- Storage Management

Overview of Hardware and I/O Considerations in Data Warehouses

I/O performance should always be a key consideration for data warehouse designers and administrators. The typical workload in a data warehouse is especially I/O intensive, with operations such as large data loads and index builds, creation of materialized views, and queries over large volumes of data. The underlying I/O system for a data warehouse should be designed to meet these heavy requirements.

In fact, one of the leading causes of performance issues in a data warehouse is poor I/O configuration. Database administrators who have previously managed other systems will likely need to pay more careful attention to the I/O configuration for a data warehouse than they may have previously done for other environments.

This chapter provides the following five high-level guidelines for data-warehouse I/O configurations:

- Configure I/O for Bandwidth not Capacity
- Stripe Far and Wide
- Use Redundancy
- Test the I/O System Before Building the Database
- Plan for Growth

The I/O configuration used by a data warehouse will depend on the characteristics of the specific storage and server capabilities, so the material in this chapter is only intended to provide guidelines for designing and tuning an I/O system.

> **See Also:** *Oracle Database Performance Tuning Guide* for additional information on I/O configurations and tuning

Configure I/O for Bandwidth not Capacity

Storage configurations for a data warehouse should be chosen based on the I/O bandwidth that they can provide, and not necessarily on their overall storage capacity. Buying storage based solely on capacity has the potential for making a mistake, especially for systems less than 500GB is total size. The capacity of individual disk drives is growing faster than the I/O throughput rates provided by those disks, leading to a situation in which a small number of disks can store a large volume of data, but cannot provide the same I/O throughput as a larger number of small disks.

As an example, consider a 200GB data mart. Using 72GB drives, this data mart could be built with as few as six drives in a fully-mirrored environment. However, six drives might not provide enough I/O bandwidth to handle a medium number of concurrent users on a 4-CPU server. Thus, even though six drives provide sufficient storage, a larger number of drives may be required to provide acceptable performance for this system.

While it may not be practical to estimate the I/O bandwidth that will be required by a data warehouse before a system is built, it is generally practical with the guidance of the hardware manufacturer to estimate how much I/O bandwidth a given server can potentially utilize, and ensure that the selected I/O configuration will be able to successfully feed the server. There are many variables in sizing the I/O systems, but one basic rule of thumb is that your data warehouse system should have multiple disks for each CPU (at least two disks for each CPU at a bare minimum) in order to achieve optimal performance.

Stripe Far and Wide

The guiding principle in configuring an I/O system for a data warehouse is to maximize I/O bandwidth by having multiple disks and channels access each database object. You can do this by striping the datafiles of the Oracle Database. A striped file is a file distributed across multiple disks. This striping can be managed by software (such as a logical volume manager), or within the storage hardware. The goal is to ensure that each tablespace is striped across a large number of disks (ideally, all of the disks) so that any database object can be accessed with the highest possible I/O bandwidth.

Use Redundancy

Because data warehouses are often the largest database systems in a company, they have the most disks and thus are also the most susceptible to the failure of a single disk. Therefore, disk redundancy is a requirement for data warehouses to protect against a hardware failure. Like disk-striping, redundancy can be achieved in many ways using software or hardware.

A key consideration is that occasionally a balance must be made between redundancy and performance. For example, a storage system in a RAID-5 configuration may be less expensive than a RAID-0+1 configuration, but it may not perform as well, either. Redundancy is necessary for any data warehouse, but the approach to redundancy may vary depending upon the performance and cost constraints of each data warehouse.

Test the I/O System Before Building the Database

The most important time to examine and tune the I/O system is before the database is even created. Once the database files are created, it is more difficult to reconfigure the files. Some logical volume managers may support dynamic reconfiguration of files, while other storage configurations may require that files be entirely rebuilt in order to reconfigure their I/O layout. In both cases, considerable system resources must be devoted to this reconfiguration.

When creating a data warehouse on a new system, the I/O bandwidth should be tested before creating all of the database datafiles to validate that the expected I/O levels are being achieved. On most operating systems, this can be done with simple scripts to measure the performance of reading and writing large test files.

Plan for Growth

A data warehouse designer should plan for future growth of a data warehouse. There are many approaches to handling the growth in a system, and the key consideration is to be able to grow the I/O system without compromising on the I/O bandwidth. You cannot, for example, add four disks to an existing system of 20 disks, and grow the database by adding a new tablespace striped across only the four new disks. A better solution would be to add new tablespaces striped across all 24 disks, and over time also convert the existing tablespaces striped across 20 disks to be striped across all 24 disks.

Storage Management

Two features to consider for managing disks are Oracle Managed Files and Automatic Storage Management. Without these features, a database administrator must manage the database files, which, in a data warehouse, can be hundreds or even thousands of files. Oracle Managed Files simplifies the administration of a database by providing functionality to automatically create and manage files, so the database administrator no longer needs to manage each database file. Automatic Storage Management provides additional functionality for managing not only files but also the disks. With Automatic Storage Management, the database administrator would administer a small number of disk groups. Automatic Storage Management handles the tasks of striping and providing disk redundancy, including rebalancing the database files when new disks are added to the system.

See Also: *Oracle Database Administrator's Guide* for more details

5

Parallelism and Partitioning in Data Warehouses

Data warehouses often contain large tables and require techniques both for managing these large tables and for providing good query performance across these large tables. This chapter discusses two key methodologies for addressing these needs: parallelism and partitioning.

These topics are discussed:

- Overview of Parallel Execution
- Granules of Parallelism
- Partitioning Design Considerations

> **Note:** Parallel execution is available only with the Oracle Database Enterprise Edition.

Overview of Parallel Execution

Parallel execution dramatically reduces response time for data-intensive operations on large databases typically associated with decision support systems (DSS) and data warehouses. You can also implement **parallel execution** on certain types of online transaction processing (OLTP) and hybrid systems. Parallel execution is sometimes called **parallelism**. Simply expressed, parallelism is the idea of breaking down a task so that, instead of one process doing all of the work in a query, many processes do part of the work at the same time. An example of this is when four processes handle four different quarters in a year instead of one process handling all four quarters by itself. The improvement in performance can be quite high. In this case, each quarter will be a **partition**, a smaller and more manageable unit of an index or table.

> **See Also:** *Oracle Database Concepts* for further conceptual information regarding parallel execution

When to Implement Parallel Execution

The most common use of parallel execution is in DSS and data warehousing environments. Complex queries, such as those involving joins of several tables or searches of very large tables, are often best executed in parallel.

Parallel execution is useful for many types of operations that access significant amounts of data. Parallel execution improves processing for:

- Large table scans and joins
- Creation of large indexes
- Partitioned index scans
- Bulk inserts, updates, and deletes
- Aggregations and copying

You can also use parallel execution to access object types within an Oracle database. For example, use parallel execution to access LOBs (large objects).

Parallel execution benefits systems that have *all* of the following characteristics:

- Symmetric multi-processors (SMP), clusters, or massively parallel systems
- Sufficient I/O bandwidth
- Underutilized or intermittently used CPUs (for example, systems where CPU usage is typically less than 30%)

- Sufficient memory to support additional memory-intensive processes such as sorts, hashing, and I/O buffers

If your system lacks any of these characteristics, parallel execution might not significantly improve performance. In fact, parallel execution can reduce system performance on overutilized systems or systems with small I/O bandwidth.

> **See Also:** Chapter 24, "Using Parallel Execution" for further information regarding parallel execution requirements

Granules of Parallelism

Different parallel operations use different types of parallelism. The optimal physical database layout depends on the parallel operations that are most prevalent in your application or even of the necessity of using partitions.

The basic unit of work in parallelism is a called a granule. Oracle Database divides the operation being parallelized (for example, a table scan, table update, or index creation) into granules. Parallel execution processes execute the operation one granule at a time. The number of granules and their size correlates with the degree of parallelism (DOP). It also affects how well the work is balanced across query server processes. There is no way you can enforce a specific granule strategy as Oracle Database makes this decision internally.

Block Range Granules

Block range granules are the basic unit of most parallel operations, even on partitioned tables. Therefore, from an Oracle Database perspective, the degree of parallelism is not related to the number of partitions.

Block range granules are ranges of physical blocks from a table. The number and the size of the granules are computed during runtime by Oracle Database to optimize and balance the work distribution for all affected parallel execution servers. The number and size of granules are dependent upon the size of the object and the DOP. Block range granules do not depend on static preallocation of tables or indexes. During the computation of the granules, Oracle Database takes the DOP into account and tries to assign granules from different datafiles to each of the parallel execution servers to avoid contention whenever possible. Additionally, Oracle Database considers the disk affinity of the granules on MPP systems to take advantage of the physical proximity between parallel execution servers and disks.

When block range granules are used predominantly for parallel access to a table or index, administrative considerations (such as recovery or using partitions for

deleting portions of data) might influence partition layout more than performance considerations.

Partition Granules

When partition granules are used, a query server process works on an entire partition or subpartition of a table or index. Because partition granules are statically determined by the structure of the table or index when a table or index is created, partition granules do not give you the flexibility in parallelizing an operation that block granules do. The maximum allowable DOP is the number of partitions. This might limit the utilization of the system and the load balancing across parallel execution servers.

When partition granules are used for parallel access to a table or index, you should use a relatively large number of partitions (ideally, three times the DOP), so that Oracle can effectively balance work across the query server processes.

Partition granules are the basic unit of parallel index range scans and of parallel operations that modify multiple partitions of a partitioned table or index. These operations include parallel creation of partitioned indexes, and parallel creation of partitioned tables.

> **See Also:** *Oracle Database Concepts* for information on disk striping and partitioning

Partitioning Design Considerations

In conjunction with parallel execution, partitioning can improve performance in data warehouses. The following are the main design considerations for partitioning:

- Types of Partitioning
- Partition Pruning
- Partition-Wise Joins

Types of Partitioning

This section describes the partitioning features that significantly enhance data access and improve overall application performance. This is especially true for applications that access tables and indexes with millions of rows and many gigabytes of data.

Partitioned tables and indexes facilitate administrative operations by enabling these operations to work on subsets of data. For example, you can add a new partition, organize an existing partition, or drop a partition and cause less than a second of interruption to a read-only application.

Using the partitioning methods described in this section can help you tune SQL statements to avoid unnecessary index and table scans (using partition pruning). You can also improve the performance of massive join operations when large amounts of data (for example, several million rows) are joined together by using partition-wise joins. Finally, partitioning data greatly improves manageability of very large databases and dramatically reduces the time required for administrative tasks such as backup and restore.

Granularity can be easily added or removed to the partitioning scheme by splitting partitions. Thus, if a table's data is skewed to fill some partitions more than others, the ones that contain more data can be split to achieve a more even distribution. Partitioning also allows one to swap partitions with a table. By being able to easily add, remove, or swap a large amount of data quickly, swapping can be used to keep a large amount of data that is being loaded inaccessible until loading is completed, or can be used as a way to stage data between different phases of use. Some examples are current day's transactions or online archives.

> **See Also:** *Oracle Database Concepts* for an introduction to the ideas behind partitioning

Partitioning Methods

Oracle offers four partitioning methods:

- Range Partitioning
- Hash Partitioning
- List Partitioning
- Composite Partitioning

Each partitioning method has different advantages and design considerations. Thus, each method is more appropriate for a particular situation.

Range Partitioning Range partitioning maps data to partitions based on ranges of partition key values that you establish for each partition. It is the most common type of partitioning and is often used with dates. For example, you might want to partition sales data into monthly partitions.

Range partitioning maps rows to partitions based on ranges of column values. Range partitioning is defined by the partitioning specification for a table or index in PARTITION BY RANGE(column_list) and by the partitioning specifications for each individual partition in VALUES LESS THAN(value_list), where column_list is an ordered list of columns that determines the partition to which a row or an index entry belongs. These columns are called the partitioning columns. The values in the partitioning columns of a particular row constitute that row's partitioning key.

value_list is an ordered list of values for the columns in the column list. Each value must be either a literal or a TO_DATE or RPAD function with constant arguments. Only the VALUES LESS THAN clause is allowed. This clause specifies a non-inclusive upper bound for the partitions. All partitions, except the first, have an implicit low value specified by the VALUES LESS THAN literal on the previous partition. Any binary values of the partition key equal to or higher than this literal are added to the next higher partition. Highest partition being where MAXVALUE literal is defined. Keyword, MAXVALUE, represents a virtual infinite value that sorts higher than any other value for the data type, including the null value.

The following statement creates a table sales_range that is range partitioned on the sales_date field:

```
CREATE TABLE sales_range
(salesman_id  NUMBER(5),
salesman_name VARCHAR2(30),
sales_amount  NUMBER(10),
sales_date    DATE)
COMPRESS
PARTITION BY RANGE(sales_date)
(PARTITION sales_jan2000 VALUES LESS THAN(TO_DATE('02/01/2000','DD/MM/YYYY')),
 PARTITION sales_feb2000 VALUES LESS THAN(TO_DATE('03/01/2000','DD/MM/YYYY')),
 PARTITION sales_mar2000 VALUES LESS THAN(TO_DATE('04/01/2000','DD/MM/YYYY')),
 PARTITION sales_apr2000 VALUES LESS THAN(TO_DATE('05/01/2000','DD/MM/YYYY')));
```

Note: This table was created with the COMPRESS keyword, thus all partitions inherit this attribute.

See Also: *Oracle Database SQL Reference* for partitioning syntax and the *Oracle Database Administrator's Guide* for more examples

Hash Partitioning Hash partitioning maps data to partitions based on a hashing algorithm that Oracle applies to a partitioning key that you identify. The hashing algorithm evenly distributes rows among partitions, giving partitions approximately the same size. Hash partitioning is the ideal method for distributing data evenly across devices. Hash partitioning is a good and easy-to-use alternative to range partitioning when data is not historical and there is no obvious column or column list where logical range partition pruning can be advantageous.

Oracle Database uses a linear hashing algorithm and to prevent data from clustering within specific partitions, you should define the number of partitions by a power of two (for example, 2, 4, 8).

The following statement creates a table `sales_hash`, which is hash partitioned on the `salesman_id` field:

```
CREATE TABLE sales_hash
(salesman_id  NUMBER(5),
salesman_name VARCHAR2(30),
sales_amount  NUMBER(10),
week_no       NUMBER(2))
PARTITION BY HASH(salesman_id)
PARTITIONS 4;
```

> **See Also:** *Oracle Database SQL Reference* for partitioning syntax and the *Oracle Database Administrator's Guide* for more examples

> **Note:** You cannot define alternate hashing algorithms for partitions.

List Partitioning List partitioning enables you to explicitly control how rows map to partitions. You do this by specifying a list of discrete values for the partitioning column in the description for each partition. This is different from range partitioning, where a range of values is associated with a partition and with hash partitioning, where you have no control of the row-to-partition mapping. The advantage of list partitioning is that you can group and organize unordered and unrelated sets of data in a natural way. The following example creates a list partitioned table grouping states according to their sales regions:

```
CREATE TABLE sales_list
(salesman_id  NUMBER(5),
salesman_name VARCHAR2(30),
sales_state   VARCHAR2(20),
```

```
sales_amount  NUMBER(10),
sales_date    DATE)
PARTITION BY LIST(sales_state)
(PARTITION sales_west VALUES('California', 'Hawaii') COMPRESS,
 PARTITION sales_east VALUES('New York', 'Virginia', 'Florida'),
 PARTITION sales_central VALUES('Texas', 'Illinois'));
```

Partition `sales_west` is furthermore created as a single compressed partition within `sales_list`. For details about partitioning and compression, see "Partitioning and Table Compression" on page 5-16.

An additional capability with list partitioning is that you can use a default partition, so that all rows that do not map to any other partition do not generate an error. For example, modifying the previous example, you can create a default partition as follows:

```
CREATE TABLE sales_list
(salesman_id  NUMBER(5),
salesman_name VARCHAR2(30),
sales_state   VARCHAR2(20),
sales_amount  NUMBER(10),
sales_date    DATE)
PARTITION BY LIST(sales_state)
(PARTITION sales_west VALUES('California', 'Hawaii'),
 PARTITION sales_east VALUES ('New York', 'Virginia', 'Florida'),
 PARTITION sales_central VALUES('Texas', 'Illinois'),
 PARTITION sales_other VALUES(DEFAULT));
```

> **See Also:** *Oracle Database SQL Reference* for partitioning syntax, "Partitioning and Table Compression" on page 5-16 for information regarding data segment compression, and the *Oracle Database Administrator's Guide* for more examples

Composite Partitioning Composite partitioning combines range and hash or list partitioning. Oracle Database first distributes data into partitions according to boundaries established by the partition ranges. Then, for range-hash partitioning, Oracle uses a hashing algorithm to further divide the data into subpartitions within each range partition. For range-list partitioning, Oracle divides the data into subpartitions within each range partition based on the explicit list you chose.

Index Partitioning

You can choose whether or not to inherit the partitioning strategy of the underlying tables. You can create both local and global indexes on a table partitioned by range, hash, or composite methods. Local indexes inherit the partitioning attributes of their related tables. For example, if you create a local index on a composite table, Oracle automatically partitions the local index using the composite method. See Chapter 6, "Indexes" for more information.

Performance Issues for Range, List, Hash, and Composite Partitioning

This section describes performance issues for:

- When to Use Range Partitioning
- When to Use Hash Partitioning
- When to Use List Partitioning
- When to Use Composite Range-Hash Partitioning
- When to Use Composite Range-List Partitioning

When to Use Range Partitioning Range partitioning is a convenient method for partitioning historical data. The boundaries of range partitions define the ordering of the partitions in the tables or indexes.

Range partitioning organizes data by time intervals on a column of type DATE. Thus, most SQL statements accessing range partitions focus on timeframes. An example of this is a SQL statement similar to "select data from a particular period in time." In such a scenario, if each partition represents data for one month, the query "find data of month 98-DEC" needs to access only the December partition of year 98. This reduces the amount of data scanned to a fraction of the total data available, an optimization method called partition pruning.

Range partitioning is also ideal when you periodically load new data and purge old data. It is easy to add or drop partitions.

It is common to keep a rolling window of data, for example keeping the past 36 months' worth of data online. Range partitioning simplifies this process. To add data from a new month, you load it into a separate table, clean it, index it, and then add it to the range-partitioned table using the EXCHANGE PARTITION statement, all while the original table remains online. Once you add the new partition, you can drop the trailing month with the DROP PARTITION statement. The alternative to using the DROP PARTITION statement can be to archive the partition and make it read only, but this works only when your partitions are in separate tablespaces.

In conclusion, consider using range partitioning when:

- Very large tables are frequently scanned by a range predicate on a good partitioning column, such as ORDER_DATE or PURCHASE_DATE. Partitioning the table on that column enables partition pruning.
- You want to maintain a rolling window of data.
- You cannot complete administrative operations, such as backup and restore, on large tables in an allotted time frame, but you can divide them into smaller logical pieces based on the partition range column.

The following example creates the table `salestable` for a period of two years, 1999 and 2000, and partitions it by range according to the column `s_salesdate` to separate the data into eight quarters, each corresponding to a partition.

```
CREATE TABLE salestable
  (s_productid  NUMBER,
   s_saledate   DATE,
   s_custid     NUMBER,
   s_totalprice NUMBER)
PARTITION BY RANGE(s_saledate)
 (PARTITION sal99q1 VALUES LESS THAN (TO_DATE('01-APR-1999', 'DD-MON-YYYY')),
  PARTITION sal99q2 VALUES LESS THAN (TO_DATE('01-JUL-1999', 'DD-MON-YYYY')),
  PARTITION sal99q3 VALUES LESS THAN (TO_DATE('01-OCT-1999', 'DD-MON-YYYY')),
  PARTITION sal99q4 VALUES LESS THAN (TO_DATE('01-JAN-2000', 'DD-MON-YYYY')),
  PARTITION sal00q1 VALUES LESS THAN (TO_DATE('01-APR-2000', 'DD-MON-YYYY')),
  PARTITION sal00q2 VALUES LESS THAN (TO_DATE('01-JUL-2000', 'DD-MON-YYYY')),
  PARTITION sal00q3 VALUES LESS THAN (TO_DATE('01-OCT-2000', 'DD-MON-YYYY')),
  PARTITION sal00q4 VALUES LESS THAN (TO_DATE('01-JAN-2001', 'DD-MON-YYYY')));
```

When to Use Hash Partitioning The way Oracle Database distributes data in hash partitions does not correspond to a business or a logical view of the data, as it does in range partitioning. Consequently, hash partitioning is not an effective way to manage historical data. However, hash partitions share some performance characteristics with range partitions. For example, partition pruning is limited to equality predicates. You can also use partition-wise joins, parallel index access, and parallel DML. See "Partition-Wise Joins" on page 5-20 for more information.

As a general rule, use hash partitioning for these purposes:

- To improve the availability and manageability of large tables or to enable parallel DML in tables that do not store historical data.
- To avoid data skew among partitions. Hash partitioning is an effective means of distributing data because Oracle hashes the data into a number of partitions,

each of which can reside on a separate device. Thus, data is evenly spread over a sufficient number of devices to maximize I/O throughput. Similarly, you can use hash partitioning to distribute evenly data among the nodes of an MPP platform that uses Oracle Real Application Clusters.

- If it is important to use partition pruning and partition-wise joins according to a partitioning key that is mostly constrained by a distinct value or value list.

Note: In hash partitioning, partition pruning uses only equality or `IN`-list predicates.

If you add or merge a hashed partition, Oracle automatically rearranges the rows to reflect the change in the number of partitions and subpartitions. The hash function that Oracle uses is especially designed to limit the cost of this reorganization. Instead of reshuffling all the rows in the table, Oracles uses an "add partition" logic that splits one and only one of the existing hashed partitions. Conversely, Oracle coalesces a partition by merging two existing hashed partitions.

Although the hash function's use of "add partition" logic dramatically improves the manageability of hash partitioned tables, it means that the hash function can cause a skew if the number of partitions of a hash partitioned table, or the number of subpartitions in each partition of a composite table, is not a power of two. In the worst case, the largest partition can be twice the size of the smallest. So for optimal performance, create a number of partitions and subpartitions for each partition that is a power of two. For example, 2, 4, 8, 16, 32, 64, 128, and so on.

The following example creates four hashed partitions for the table `sales_hash` using the column `s_productid` as the partition key:

```
CREATE TABLE sales_hash
  (s_productid  NUMBER,
   s_saledate   DATE,
   s_custid     NUMBER,
   s_totalprice NUMBER)
PARTITION BY HASH(s_productid)
PARTITIONS 4;
```

Specify partition names if you want to choose the names of the partitions. Otherwise, Oracle automatically generates internal names for the partitions. Also, you can use the `STORE IN` clause to assign hash partitions to tablespaces in a round-robin manner.

> **See Also:** *Oracle Database SQL Reference* for partitioning syntax and the *Oracle Database Administrator's Guide* for more examples

When to Use List Partitioning You should use list partitioning when you want to specifically map rows to partitions based on discrete values.

Unlike range and hash partitioning, multi-column partition keys are not supported for list partitioning. If a table is partitioned by list, the partitioning key can only consist of a single column of the table.

When to Use Composite Range-Hash Partitioning Composite range-hash partitioning offers the benefits of both range and hash partitioning. With composite range-hash partitioning, Oracle first partitions by range. Then, within each range, Oracle creates subpartitions and distributes data within them using the same hashing algorithm it uses for hash partitioned tables.

Data placed in composite partitions is logically ordered only by the boundaries that define the range level partitions. The partitioning of data within each partition has no logical organization beyond the identity of the partition to which the subpartitions belong.

Consequently, tables and local indexes partitioned using the composite range-hash method:

- Support historical data at the partition level.
- Support the use of subpartitions as units of parallelism for parallel operations such as PDML or space management and backup and recovery.
- Are eligible for partition pruning and partition-wise joins on the range and hash partitions.

Using Composite Range-Hash Partitioning Use the composite range-hash partitioning method for tables and local indexes if:

- Partitions must have a logical meaning to efficiently support historical data
- The contents of a partition can be spread across multiple tablespaces, devices, or nodes (of an MPP system)
- You require both partition pruning and partition-wise joins even when the pruning and join predicates use different columns of the partitioned table
- You require a degree of parallelism that is greater than the number of partitions for backup, recovery, and parallel operations

Most large tables in a data warehouse should use range partitioning. Composite partitioning should be used for very large tables or for data warehouses with a well-defined need for these conditions. When using the composite method, Oracle stores each subpartition on a different segment. Thus, the subpartitions may have properties that differ from the properties of the table or from the partition to which the subpartitions belong.

The following example partitions the table `sales_range_hash` by range on the column `s_saledate` to create four partitions that order data by time. Then, within each range partition, the data is further subdivided into 16 subpartitions by hash on the column `s_productid`:

```
CREATE TABLE sales_range_hash(
  s_productid  NUMBER,
  s_saledate   DATE,
  s_custid     NUMBER,
  s_totalprice NUMBER)
PARTITION BY RANGE (s_saledate)
SUBPARTITION BY HASH (s_productid) SUBPARTITIONS 8
 (PARTITION sal99q1 VALUES LESS THAN (TO_DATE('01-APR-1999', 'DD-MON-YYYY')),
  PARTITION sal99q2 VALUES LESS THAN (TO_DATE('01-JUL-1999', 'DD-MON-YYYY')),
  PARTITION sal99q3 VALUES LESS THAN (TO_DATE('01-OCT-1999', 'DD-MON-YYYY')),
  PARTITION sal99q4 VALUES LESS THAN (TO_DATE('01-JAN-2000', 'DD-MON-YYYY')));
```

Each hashed subpartition contains sales data for a single quarter ordered by product code. The total number of subpartitions is 4x8 or 32.

In addition to this syntax, you can create subpartitions by using a subpartition template. This offers better ease in naming and control of location for tablespaces and subpartitions. The following statement illustrates this:

```
CREATE TABLE sales_range_hash(
  s_productid  NUMBER,
  s_saledate   DATE,
  s_custid     NUMBER,
  s_totalprice NUMBER)
PARTITION BY RANGE (s_saledate)
SUBPARTITION BY HASH (s_productid)
SUBPARTITION TEMPLATE(
SUBPARTITION sp1 TABLESPACE tbs1,
SUBPARTITION sp2 TABLESPACE tbs2,
SUBPARTITION sp3 TABLESPACE tbs3,
SUBPARTITION sp4 TABLESPACE tbs4,
SUBPARTITION sp5 TABLESPACE tbs5,
SUBPARTITION sp6 TABLESPACE tbs6,
```

```
SUBPARTITION sp7 TABLESPACE tbs7,
SUBPARTITION sp8 TABLESPACE tbs8)
  (PARTITION sal99q1 VALUES LESS THAN (TO_DATE('01-APR-1999', 'DD-MON-YYYY')),
   PARTITION sal99q2 VALUES LESS THAN (TO_DATE('01-JUL-1999', 'DD-MON-YYYY')),
   PARTITION sal99q3 VALUES LESS THAN (TO_DATE('01-OCT-1999', 'DD-MON-YYYY')),
   PARTITION sal99q4 VALUES LESS THAN (TO_DATE('01-JAN-2000', 'DD-MON-YYYY')));
```

In this example, every partition has the same number of subpartitions. A sample mapping for `sal99q1` is illustrated in Table 5–1. Similar mappings exist for `sal99q2` through `sal99q4`.

Table 5–1 Subpartition Mapping

Subpartition	Tablespace
sal99q1_sp1	tbs1
sal99q1_sp2	tbs2
sal99q1_sp3	tbs3
sal99q1_sp4	tbs4
sal99q1_sp5	tbs5
sal99q1_sp6	tbs6
sal99q1_sp7	tbs7
sal99q1_sp8	tbs8

> **See Also:** *Oracle Database SQL Reference* for details regarding syntax and restrictions

When to Use Composite Range-List Partitioning Composite range-list partitioning offers the benefits of both range and list partitioning. With composite range-list partitioning, Oracle first partitions by range. Then, within each range, Oracle creates subpartitions and distributes data within them to organize sets of data in a natural way as assigned by the list.

Data placed in composite partitions is logically ordered only by the boundaries that define the range level partitions.

Using Composite Range-List Partitioning Use the composite range-list partitioning method for tables and local indexes if:

- Subpartitions have a logical grouping defined by the user.

- The contents of a partition can be spread across multiple tablespaces, devices, or nodes (of an MPP system).
- You require both partition pruning and partition-wise joins even when the pruning and join predicates use different columns of the partitioned table.
- You require a degree of parallelism that is greater than the number of partitions for backup, recovery, and parallel operations.

Most large tables in a data warehouse should use range partitioning. Composite partitioning should be used for very large tables or for data warehouses with a well-defined need for these conditions. When using the composite method, Oracle stores each subpartition on a different segment. Thus, the subpartitions may have properties that differ from the properties of the table or from the partition to which the subpartitions belong.

This statement creates a table `quarterly_regional_sales` that is range partitioned on the `txn_date` field and list subpartitioned on `state`.

```
CREATE TABLE quarterly_regional_sales
(deptno NUMBER, item_no VARCHAR2(20),
 txn_date DATE, txn_amount NUMBER, state VARCHAR2(2))
PARTITION BY RANGE (txn_date)
SUBPARTITION BY LIST (state)
(
PARTITION q1_1999 VALUES LESS THAN(TO_DATE('1-APR-1999','DD-MON-YYYY'))
(SUBPARTITION q1_1999_northwest VALUES ('OR', 'WA'),
SUBPARTITION q1_1999_southwest VALUES ('AZ', 'UT', 'NM'),
SUBPARTITION q1_1999_northeast VALUES ('NY', 'VM', 'NJ'),
SUBPARTITION q1_1999_southeast VALUES  ('FL', 'GA'),
SUBPARTITION q1_1999_northcentral VALUES ('SD', 'WI'),
SUBPARTITION q1_1999_southcentral VALUES ('NM', 'TX')),
PARTITION q2_1999 VALUES LESS THAN(TO_DATE('1-JUL-1999','DD-MON-YYYY'))
(SUBPARTITION q2_1999_northwest VALUES ('OR', 'WA'),
SUBPARTITION q2_1999_southwest VALUES ('AZ', 'UT', 'NM'),
SUBPARTITION q2_1999_northeast VALUES ('NY', 'VM', 'NJ'),
SUBPARTITION q2_1999_southeast VALUES ('FL', 'GA'),
SUBPARTITION q2_1999_northcentral VALUES ('SD', 'WI'),
SUBPARTITION q2_1999_southcentral VALUES ('NM', 'TX')),
PARTITION q3_1999 VALUES LESS THAN (TO_DATE('1-OCT-1999','DD-MON-YYYY'))
(SUBPARTITION q3_1999_northwest VALUES ('OR', 'WA'),
SUBPARTITION q3_1999_southwest VALUES  ('AZ', 'UT', 'NM'),
SUBPARTITION q3_1999_northeast VALUES ('NY', 'VM', 'NJ'),
SUBPARTITION q3_1999_southeast VALUES ('FL', 'GA'),
SUBPARTITION q3_1999_northcentral VALUES ('SD', 'WI'),
SUBPARTITION q3_1999_southcentral VALUES ('NM', 'TX')),
```

```
PARTITION q4_1999 VALUES LESS THAN (TO_DATE('1-JAN-2000','DD-MON-YYYY'))
(SUBPARTITION q4_1999_northwest VALUES('OR', 'WA'),
SUBPARTITION q4_1999_southwest VALUES('AZ', 'UT', 'NM'),
SUBPARTITION q4_1999_northeast VALUES('NY', 'VM', 'NJ'),
SUBPARTITION q4_1999_southeast VALUES('FL', 'GA'),
SUBPARTITION q4_1999_northcentral VALUES ('SD', 'WI'),
SUBPARTITION q4_1999_southcentral VALUES ('NM', 'TX')));
```

You can create subpartitions in a composite partitioned table using a subpartition template. A subpartition template simplifies the specification of subpartitions by not requiring that a subpartition descriptor be specified for every partition in the table. Instead, you describe subpartitions only once in a template, then apply that subpartition template to every partition in the table. The following statement illustrates an example where you can choose the subpartition name and tablespace locations:

```
CREATE TABLE quarterly_regional_sales
(deptno NUMBER,  item_no VARCHAR2(20),
 txn_date DATE,  txn_amount NUMBER,  state VARCHAR2(2))
PARTITION BY RANGE (txn_date)
SUBPARTITION BY LIST (state)
SUBPARTITION TEMPLATE(
SUBPARTITION northwest VALUES ('OR', 'WA') TABLESPACE ts1,
SUBPARTITION southwest VALUES ('AZ', 'UT', 'NM') TABLESPACE ts2,
SUBPARTITION northeast VALUES ('NY', 'VM', 'NJ') TABLESPACE ts3,
SUBPARTITION southeast VALUES ('FL', 'GA') TABLESPACE ts4,
SUBPARTITION northcentral VALUES ('SD', 'WI') TABLESPACE ts5,
SUBPARTITION southcentral VALUES ('NM', 'TX') TABLESPACE ts6)
(
PARTITION q1_1999 VALUES LESS THAN(TO_DATE('1-APR-1999','DD-MON-YYYY')),
PARTITION q2_1999 VALUES LESS THAN(TO_DATE('1-JUL-1999','DD-MON-YYYY')),
PARTITION q3_1999 VALUES LESS THAN(TO_DATE('1-OCT-1999','DD-MON-YYYY')),
PARTITION q4_1999 VALUES LESS THAN(TO_DATE('1-JAN-2000','DD-MON-YYYY')));
```

> **See Also:** *Oracle Database SQL Reference* for details regarding syntax and restrictions

Partitioning and Table Compression

You can compress several partitions or a complete partitioned heap-organized table. You do this by either defining a complete partitioned table as being compressed, or by defining it on a per-partition level. Partitions without a specific declaration

inherit the attribute from the table definition or, if nothing is specified on table level, from the tablespace definition.

To decide whether or not a partition should be compressed or stay uncompressed adheres to the same rules as a nonpartitioned table. However, due to the capability of range and composite partitioning to separate data logically into distinct partitions, such a partitioned table is an ideal candidate for compressing parts of the data (partitions) that are mainly read-only. It is, for example, beneficial in all rolling window operations as a kind of intermediate stage before aging out old data. With data segment compression, you can keep more old data online, minimizing the burden of additional storage consumption.

You can also change any existing uncompressed table partition later on, add new compressed and uncompressed partitions, or change the compression attribute as part of any partition maintenance operation that requires data movement, such as `MERGE PARTITION`, `SPLIT PARTITION`, or `MOVE PARTITION`. The partitions can contain data or can be empty.

The access and maintenance of a partially or fully compressed partitioned table are the same as for a fully uncompressed partitioned table. Everything that applies to fully uncompressed partitioned tables is also valid for partially or fully compressed partitioned tables.

> **See Also:** Chapter 3, "Physical Design in Data Warehouses" for a generic discussion of table compression, Chapter 15, "Maintaining the Data Warehouse" for a sample rolling window operation with a range-partitioned table, and *Oracle Database Performance Tuning Guide* for an example of calculating the compression ratio

Table Compression and Bitmap Indexes

If you want to use table compression on partitioned tables with bitmap indexes, you need to do the following before you introduce the compression attribute for the first time:

1. Mark bitmap indexes unusable.
2. Set the compression attribute.
3. Rebuild the indexes.

The first time you make a compressed partition part of an already existing, fully uncompressed partitioned table, you must either drop all existing bitmap indexes or mark them `UNUSABLE` prior to adding a compressed partition. This must be done irrespective of whether any partition contains any data. It is also independent of the

operation that causes one or more compressed partitions to become part of the table. This does not apply to a partitioned table having B-tree indexes only.

This rebuilding of the bitmap index structures is necessary to accommodate the potentially higher number of rows stored for each data block with table compression enabled and must be done only for the first time. All subsequent operations, whether they affect compressed or uncompressed partitions, or change the compression attribute, behave identically for uncompressed, partially compressed, or fully compressed partitioned tables.

To avoid the recreation of any bitmap index structure, Oracle recommends creating every partitioned table with at least one compressed partition whenever you plan to partially or fully compress the partitioned table in the future. This compressed partition can stay empty or even can be dropped after the partition table creation.

Having a partitioned table with compressed partitions can lead to slightly larger bitmap index structures for the uncompressed partitions. The bitmap index structures for the compressed partitions, however, are in most cases smaller than the appropriate bitmap index structure before table compression. This highly depends on the achieved compression rates.

Note: Oracle Database will raise an error if compression is introduced to an object for the first time and there are usable bitmap index segments.

Example of Table Compression and Partitioning

The following statement moves and compresses an already existing partition `sales_q1_1998` of table `sales`:

```
ALTER TABLE sales
MOVE PARTITION sales_q1_1998 TABLESPACE ts_arch_q1_1998 COMPRESS;
```

If you use the `MOVE` statement, the local indexes for partition `sales_q1_1998` become unusable. You have to rebuild them afterward, as follows:

```
ALTER TABLE sales
MODIFY PARTITION sales_q1_1998 REBUILD UNUSABLE LOCAL INDEXES;
```

The following statement merges two existing partitions into a new, compressed partition, residing in a separate tablespace. The local bitmap indexes have to be rebuilt afterward, as follows:

```
ALTER TABLE sales MERGE PARTITIONS sales_q1_1998, sales_q2_1998
```

```
INTO PARTITION sales_1_1998 TABLESPACE ts_arch_1_1998
COMPRESS UPDATE INDEXES;
```

> **See Also:** *Oracle Database Performance Tuning Guide* for details regarding how to estimate the compression ratio when using table compression

Partition Pruning

Partition pruning is an essential performance feature for data warehouses. In partition pruning, the optimizer analyzes `FROM` and `WHERE` clauses in SQL statements to eliminate unneeded partitions when building the partition access list. This enables Oracle Database to perform operations only on those partitions that are relevant to the SQL statement. Oracle prunes partitions when you use range, `LIKE`, equality, and `IN`-list predicates on the range or list partitioning columns, and when you use equality and `IN`-list predicates on the hash partitioning columns.

Partition pruning dramatically reduces the amount of data retrieved from disk and shortens the use of processing time, improving query performance and resource utilization. If you partition the index and table on different columns (with a global, partitioned index), partition pruning also eliminates index partitions even when the partitions of the underlying table cannot be eliminated.

On composite partitioned objects, Oracle can prune at both the range partition level and at the hash or list subpartition level using the relevant predicates. Refer to the table `sales_range_hash` earlier, partitioned by range on the column `s_salesdate` and subpartitioned by hash on column `s_productid`, and consider the following example:

```
SELECT * FROM sales_range_hash
WHERE s_saledate BETWEEN (TO_DATE('01-JUL-1999', 'DD-MON-YYYY')) AND
 (TO_DATE('01-OCT-1999', 'DD-MON-YYYY')) AND s_productid = 1200;
```

Oracle uses the predicate on the partitioning columns to perform partition pruning as follows:

- When using range partitioning, Oracle accesses only partitions `sal99q2` and `sal99q3`.
- When using hash subpartitioning, Oracle accesses only the one subpartition in each partition that stores the rows with `s_productid=1200`. The mapping between the subpartition and the predicate is calculated based on Oracle's internal hash distribution function.

Pruning Using DATE Columns

In the earlier partitioning pruning example, the date value was fully specified as four digits for the year using the TO_DATE function, just as it was in the underlying table's range partitioning description. While this is the recommended format for specifying date values, the optimizer can prune partitions using the predicates on s_salesdate when you use other formats, as in the following example:

```
SELECT * FROM sales_range_hash
WHERE s_saledate BETWEEN TO_DATE('01-JUL-99', 'DD-MON-RR') AND
  TO_DATE('01-OCT-99', 'DD-MON-RR') AND s_productid = 1200;
```

Although this uses the DD-MON-RR format, which is not the same as the base partition, the optimizer can still prune properly.

If you execute an EXPLAIN PLAN statement on the query, the PARTITION_START and PARTITION_STOP columns of the output table do not specify which partitions Oracle is accessing. Instead, you see the keyword KEY for both columns. The keyword KEY for both columns means that partition pruning occurs at run-time. It can also affect the execution plan because the information about the pruned partitions is missing compared to the same statement using the same TO_DATE function than the partition table definition.

Avoiding I/O Bottlenecks

To avoid I/O bottlenecks, when Oracle is not scanning all partitions because some have been eliminated by pruning, spread each partition over several devices. On MPP systems, spread those devices over multiple nodes.

Partition-Wise Joins

Partition-wise joins reduce query response time by minimizing the amount of data exchanged among parallel execution servers when joins execute in parallel. This significantly reduces response time and improves the use of both CPU and memory resources. In Oracle Real Application Clusters environments, partition-wise joins also avoid or at least limit the data traffic over the interconnect, which is the key to achieving good scalability for massive join operations.

Partition-wise joins can be full or partial. Oracle decides which type of join to use.

Full Partition-Wise Joins

A full partition-wise join divides a large join into smaller joins between a pair of partitions from the two joined tables. To use this feature, you must equipartition both tables on their join keys. For example, consider a large join between a sales

table and a customer table on the column customerid. The query "find the records of all customers who bought more than 100 articles in Quarter 3 of 1999" is a typical example of a SQL statement performing such a join. The following is an example of this:

```
SELECT c.cust_last_name, COUNT(*)
FROM sales s, customers c
WHERE s.cust_id = c.cust_id AND
s.time_id BETWEEN TO_DATE('01-JUL-1999', 'DD-MON-YYYY') AND
     (TO_DATE('01-OCT-1999', 'DD-MON-YYYY'))
GROUP BY c.cust_last_name HAVING COUNT(*) > 100;
```

This large join is typical in data warehousing environments. The entire customer table is joined with one quarter of the sales data. In large data warehouse applications, this might mean joining millions of rows. The join method to use in that case is obviously a hash join. You can reduce the processing time for this hash join even more if both tables are equipartitioned on the `customerid` column. This enables a full partition-wise join.

When you execute a full partition-wise join in parallel, the granule of parallelism, as described under "Granules of Parallelism" on page 5-3, is a partition. As a result, the degree of parallelism is limited to the number of partitions. For example, you require at least 16 partitions to set the degree of parallelism of the query to 16.

You can use various partitioning methods to equipartition both tables on the column `customerid` with 16 partitions. These methods are described in these subsections.

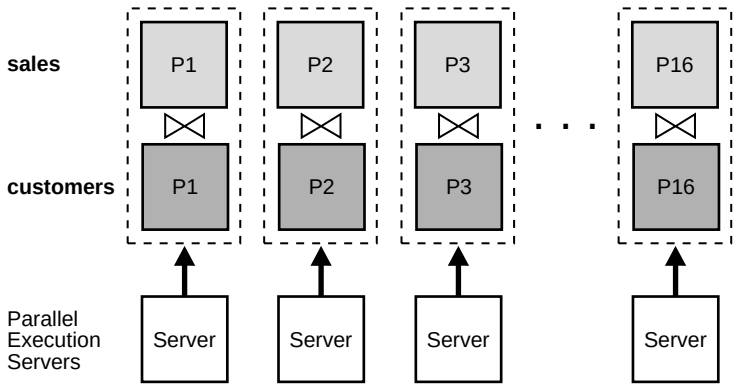
Hash-Hash This is the simplest method: the `customers` and `sales` tables are both partitioned by hash into 16 partitions, on the `s_customerid` and `c_customerid` columns. This partitioning method enables full partition-wise join when the tables are joined on `c_customerid` and `s_customerid`, both representing the same customer identification number. Because you are using the same hash function to distribute the same information (customer ID) into the same number of hash partitions, you can join the equivalent partitions. They are storing the same values.

In serial, this join is performed between pairs of matching hash partitions, one at a time. When one partition pair has been joined, the join of another partition pair begins. The join completes when the 16 partition pairs have been processed.

Note: A pair of matching hash partitions is defined as one partition with the same partition number from each table. For example, with full partition-wise joins we join partition 0 of `sales` with partition 0 of `customers`, partition 1 of `sales` with partition 1 of `customers`, and so on.

Parallel execution of a full partition-wise join is a straightforward parallelization of the serial execution. Instead of joining one partition pair at a time, 16 partition pairs are joined in parallel by the 16 query servers. Figure 5–1 illustrates the parallel execution of a full partition-wise join.

Figure 5–1 Parallel Execution of a Full Partition-wise Join



In Figure 5–1, assume that the degree of parallelism and the number of partitions are the same, in other words, 16 for both. Defining more partitions than the degree of parallelism may improve load balancing and limit possible skew in the execution. If you have more partitions than query servers, when one query server completes the join of one pair of partitions, it requests that the query coordinator give it another pair to join. This process repeats until all pairs have been processed. This method enables the load to be balanced dynamically when the number of partition pairs is greater than the degree of parallelism, for example, 64 partitions with a degree of parallelism of 16.

Note: To guarantee an equal work distribution, the number of partitions should always be a multiple of the degree of parallelism.

In Oracle Real Application Clusters environments running on shared-nothing or MPP platforms, placing partitions on nodes is critical to achieving good scalability. To avoid remote I/O, both matching partitions should have affinity to the same node. Partition pairs should be spread over all nodes to avoid bottlenecks and to use all CPU resources available on the system.

Nodes can host multiple pairs when there are more pairs than nodes. For example, with an 8-node system and 16 partition pairs, each node receives two pairs.

> **See Also:** *Oracle Real Application Clusters Deployment and Performance Guide* for more information on data affinity

(Composite-Hash)-Hash This method is a variation of the hash-hash method. The `sales` table is a typical example of a table storing historical data. For all the reasons mentioned under the heading "When to Use Range Partitioning" on page 5-9, range is the logical initial partitioning method.

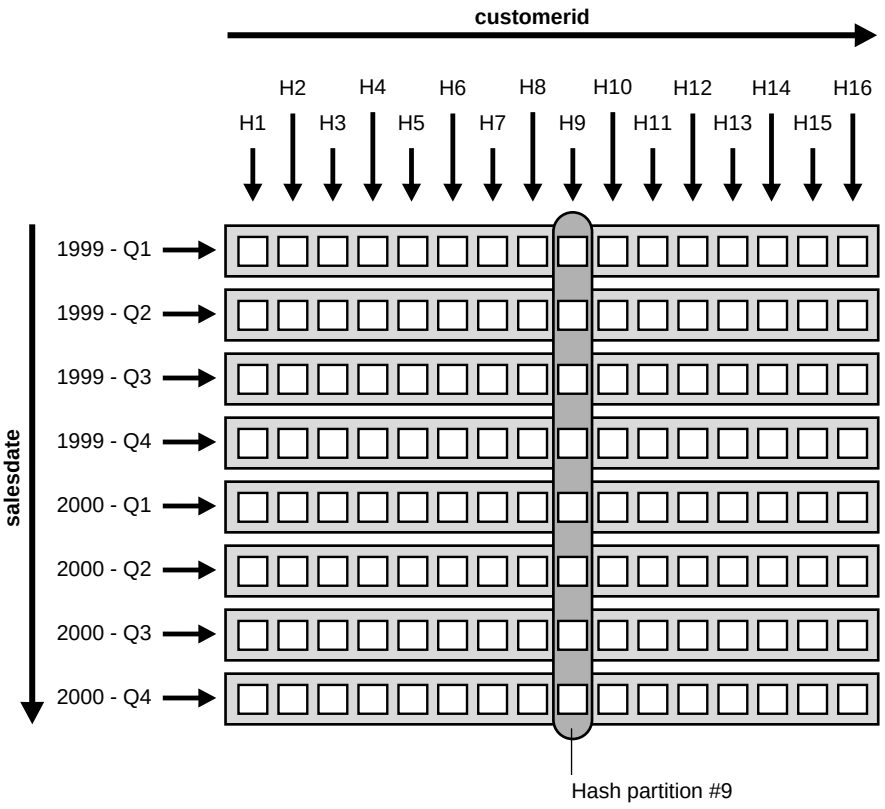
For example, assume you want to partition the `sales` table into eight partitions by range on the column `s_salesdate`. Also assume you have two years and that each partition represents a quarter. Instead of using range partitioning, you can use composite partitioning to enable a full partition-wise join while preserving the partitioning on `s_salesdate`. Partition the `sales` table by range on `s_salesdate` and then subpartition each partition by hash on `s_customerid` using 16 subpartitions for each partition, for a total of 128 subpartitions. The `customers` table can still use hash partitioning with 16 partitions.

When you use the method just described, a full partition-wise join works similarly to the one created by the hash-hash method. The join is still divided into 16 smaller joins between hash partition pairs from both tables. The difference is that now each hash partition in the `sales` table is composed of a set of 8 subpartitions, one from each range partition.

Figure 5–2 illustrates how the hash partitions are formed in the `sales` table. Each cell represents a subpartition. Each row corresponds to one range partition, for a total of 8 range partitions. Each range partition has 16 subpartitions. Each column corresponds to one hash partition for a total of 16 hash partitions; each hash partition has 8 subpartitions. Note that hash partitions can be defined only if all partitions have the same number of subpartitions, in this case, 16.

Hash partitions are implicit in a composite table. However, Oracle does not record them in the data dictionary, and you cannot manipulate them with DDL commands as you can range partitions.

Figure 5–2 Range and Hash Partitions of a Composite Table



(Composite-Hash)-Hash partitioning is effective because it lets you combine pruning (on `s_salesdate`) with a full partition-wise join (on `customerid`). In the previous example query, pruning is achieved by scanning only the subpartitions corresponding to Q3 of 1999, in other words, row number 3 in Figure 5–2. Oracle then joins these subpartitions with the customer table, using a full partition-wise join.

All characteristics of the hash-hash partition-wise join apply to the composite-hash partition-wise join. In particular, for this example, these two points are common to both methods:

- The degree of parallelism for this full partition-wise join cannot exceed 16. Even though the `sales` table has 128 subpartitions, it has only 16 hash partitions.

- The rules for data placement on MPP systems apply here. The only difference is that a hash partition is now a collection of subpartitions. You must ensure that all these subpartitions are placed on the same node as the matching hash partition from the other table. For example, in Figure 5–2, store hash partition 9 of the `sales` table shown by the eight circled subpartitions, on the same node as hash partition 9 of the `customers` table.

(Composite-List)-List The (Composite-List)-List method resembles that for (Composite-Hash)-Hash partition-wise joins.

Composite-Composite (Hash/List Dimension) If needed, you can also partition the `customer` table by the composite method. For example, you partition it by range on a postal code column to enable pruning based on postal code. You then subpartition it by hash on `customerid` using the same number of partitions (16) to enable a partition-wise join on the hash dimension.

Range-Range and List-List You can also join range partitioned tables with range partitioned tables and list partitioned tables with list partitioned tables in a partition-wise manner, but this is relatively uncommon. This is more complex to implement because you must know the distribution of the data before performing the join. Furthermore, if you do not correctly identify the partition bounds so that you have partitions of equal size, data skew during the execution may result.

The basic principle for using range-range and list-list is the same as for using hash-hash: you must equipartition both tables. This means that the number of partitions must be the same and the partition bounds must be identical. For example, assume that you know in advance that you have 10 million customers, and that the values for `customerid` vary from 1 to 10,000,000. In other words, you have 10 million possible different values. To create 16 partitions, you can range partition both tables, `sales` on `s_customerid` and `customers` on `c_customerid`. You should define partition bounds for both tables in order to generate partitions of the same size. In this example, partition bounds should be defined as 625001, 1250001, 1875001, ... 10000001, so that each partition contains 625000 rows.

Range-Composite, Composite-Composite (Range Dimension) Finally, you can also subpartition one or both tables on another column. Therefore, the range-composite and composite-composite methods on the range dimension are also valid for enabling a full partition-wise join on the range dimension.

Partial Partition-wise Joins

Oracle can perform partial partition-wise joins only in parallel. Unlike full partition-wise joins, partial partition-wise joins require you to partition only one table on the join key, not both tables. The partitioned table is referred to as the reference table. The other table may or may not be partitioned. Partial partition-wise joins are more common than full partition-wise joins.

To execute a partial partition-wise join, Oracle dynamically repartitions the other table based on the partitioning of the reference table. Once the other table is repartitioned, the execution is similar to a full partition-wise join.

The performance advantage that partial partition-wise joins have over joins in non-partitioned tables is that the reference table is not moved during the join operation. Parallel joins between non-partitioned tables require both input tables to be redistributed on the join key. This redistribution operation involves exchanging rows between parallel execution servers. This is a CPU-intensive operation that can lead to excessive interconnect traffic in Oracle Real Application Clusters environments. Partitioning large tables on a join key, either a foreign or primary key, prevents this redistribution every time the table is joined on that key. Of course, if you choose a foreign key to partition the table, which is the most common scenario, select a foreign key that is involved in many queries.

To illustrate partial partition-wise joins, consider the previous `sales/customer` example. Assume that sales is not partitioned or is partitioned on a column other than `s_customerid` . Because `sales` is often joined with `customers` on `customerid`, and because this join dominates our application workload, partition `sales` on `s_customerid` to enable partial partition-wise join every time `customers` and `sales` are joined. As in full partition-wise join, you have several alternatives:

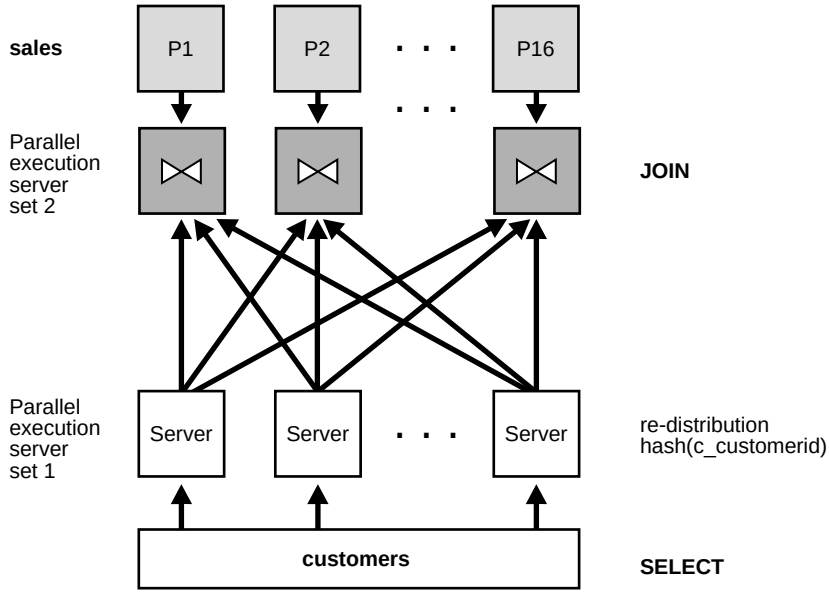
Hash/List The simplest method to enable a partial partition-wise join is to partition `sales` by hash on `s_customerid`. The number of partitions determines the maximum degree of parallelism, because the partition is the smallest granule of parallelism for partial partition-wise join operations.

The parallel execution of a partial partition-wise join is illustrated in Figure 5–3, which assumes that both the degree of parallelism and the number of partitions of `sales` are 16. The execution involves two sets of query servers: one set, labeled *set 1* in Figure 5–3, scans the `customers` table in parallel. The granule of parallelism for the scan operation is a range of blocks.

Rows from `customers` that are selected by the first set, in this case all rows, are redistributed to the second set of query servers by hashing `customerid`. For

example, all rows in `customers` that could have matching rows in partition `P1` of `sales` are sent to query server 1 in the second set. Rows received by the second set of query servers are joined with the rows from the corresponding partitions in `sales`. Query server number 1 in the second set joins all `customers` rows that it receives with partition `P1` of `sales`.

Figure 5–3 Partial Partition-Wise Join



Note: This section is based on range-hash, but it also applies for range-list partial partition-wise joins.

Considerations for full partition-wise joins also apply to partial partition-wise joins:

- The degree of parallelism does not need to equal the number of partitions. In Figure 5–3, the query executes with two sets of 16 query servers. In this case, Oracle assigns 1 partition to each query server of the second set. Again, the number of partitions should always be a multiple of the degree of parallelism.

- In Oracle Real Application Clusters environments on shared-nothing platforms (MPPs), each hash partition of `sales` should preferably have affinity to only one node in order to avoid remote I/Os. Also, spread partitions over all nodes to avoid bottlenecks and use all CPU resources available on the system. A node can host multiple partitions when there are more partitions than nodes.

  See Also: *Oracle Real Application Clusters Deployment and Performance Guide* for more information on data affinity

Composite As with full partition-wise joins, the prime partitioning method for the `sales` table is to use the range method on column `s_salesdate`. This is because `sales` is a typical example of a table that stores historical data. To enable a partial partition-wise join while preserving this range partitioning, subpartition `sales` by hash on column `s_customerid` using 16 subpartitions for each partition. Pruning and partial partition-wise joins can be used together if a query joins `customers` and `sales` and if the query has a selection predicate on `s_salesdate`.

When `sales` is composite, the granule of parallelism for a partial partition-wise join is a hash partition and not a subpartition. Refer to Figure 5–2 for an illustration of a hash partition in a composite table. Again, the number of hash partitions should be a multiple of the degree of parallelism. Also, on an MPP system, ensure that each hash partition has affinity to a single node. In the previous example, the eight subpartitions composing a hash partition should have affinity to the same node.

Note: This section is based on range-hash, but it also applies for range-list partial partition-wise joins.

Range Finally, you can use range partitioning on `s_customerid` to enable a partial partition-wise join. This works similarly to the hash method, but a side effect of range partitioning is that the resulting data distribution could be skewed if the size of the partitions differs. Moreover, this method is more complex to implement because it requires prior knowledge of the values of the partitioning column that is also a join key.

Benefits of Partition-Wise Joins

Partition-wise joins offer benefits described in this section:

- Reduction of Communications Overhead
- Reduction of Memory Requirements

Reduction of Communications Overhead When executed in parallel, partition-wise joins reduce communications overhead. This is because, in the default case, parallel execution of a join operation by a set of parallel execution servers requires the redistribution of each table on the join column into disjoint subsets of rows. These disjoint subsets of rows are then joined pair-wise by a single parallel execution server.

Oracle can avoid redistributing the partitions because the two tables are already partitioned on the join column. This enables each parallel execution server to join a pair of matching partitions.

This improved performance from using parallel execution is even more noticeable in Oracle Real Application Clusters configurations with internode parallel execution. Partition-wise joins dramatically reduce interconnect traffic. Using this feature is for large DSS configurations that use Oracle Real Application Clusters.

Currently, most Oracle Real Application Clusters platforms, such as MPP and SMP clusters, provide limited interconnect bandwidths compared with their processing powers. Ideally, interconnect bandwidth should be comparable to disk bandwidth, but this is seldom the case. As a result, most join operations in Oracle Real Application Clusters experience high interconnect latencies without parallel execution of partition-wise joins.

Reduction of Memory Requirements Partition-wise joins require less memory than the equivalent join operation of the complete data set of the tables being joined.

In the case of serial joins, the join is performed at the same time on a pair of matching partitions. If data is evenly distributed across partitions, the memory requirement is divided by the number of partitions. There is no skew.

In the parallel case, memory requirements depend on the number of partition pairs that are joined in parallel. For example, if the degree of parallelism is 20 and the number of partitions is 100, 5 times less memory is required because only 20 joins of two partitions are performed at the same time. The fact that partition-wise joins require less memory has a direct effect on performance. For example, the join probably does not need to write blocks to disk during the build phase of a hash join.

Performance Considerations for Parallel Partition-Wise Joins

The optimizer weighs the advantages and disadvantages when deciding whether or not to use partition-wise joins.

- In range partitioning where partition sizes differ, data skew increases response time; some parallel execution servers take longer than others to finish their

joins. Oracle recommends the use of hash (sub)partitioning to enable partition-wise joins because hash partitioning, if the number of partitions is a power of two, limits the risk of skew.

- The number of partitions used for partition-wise joins should, if possible, be a multiple of the number of query servers. With a degree of parallelism of 16, for example, you can have 16, 32, or even 64 partitions. If there is an even number of partitions, some parallel execution servers are used less than others. For example, if there are 17 evenly distributed partition pairs, only one pair will work on the last join, while the other pairs will have to wait. This is because, in the beginning of the execution, each parallel execution server works on a different partition pair. At the end of this first phase, only one pair is left. Thus, a single parallel execution server joins this remaining pair while all other parallel execution servers are idle.
- Sometimes, parallel joins can cause remote I/Os. For example, on Oracle Real Application Clusters environments running on MPP configurations, if a pair of matching partitions is not collocated on the same node, a partition-wise join requires extra internode communication due to remote I/O. This is because Oracle must transfer at least one partition to the node where the join is performed. In this case, it is better to explicitly redistribute the data than to use a partition-wise join.

Partitioning and Subpartitioning Columns and Keys

The partitioning columns (or subpartitioning columns) of a table or index consist of an ordered list of columns whose values determine how the data is partitioned or subpartitioned. This list can include up to 16 columns, and cannot include any of the following types of columns:

- A `LEVEL` or `ROWID` pseudocolumn
- A column of the `ROWID` datatype
- A nested table, `VARRAY`, object type, or `REF` column
- A LOB column (BLOB, CLOB, NCLOB, or BFILE datatype)

A row's partitioning key is an ordered list of its values for the partitioning columns. Similarly, in composite partitioning a row's subpartitioning key is an ordered list of its values for the subpartitioning columns. Oracle applies either the range, list, or hash method to each row's partitioning key or subpartitioning key to determine which partition or subpartition the row belongs in.

Partition Bounds for Range Partitioning

In a range-partitioned table or index, the partitioning key of each row is compared with a set of upper and lower bounds to determine which partition the row belongs in:

- Every partition of a range-partitioned table or index has a noninclusive upper bound, which is specified by the VALUES LESS THAN clause.
- Every partition except the first partition also has an inclusive lower bound, which is specified by the VALUES LESS THAN on the next-lower partition.

The partition bounds collectively define an ordering of the partitions in a table or index. The first partition is the partition with the lowest VALUES LESS THAN clause, and the last or highest partition is the partition with the highest VALUES LESS THAN clause.

Comparing Partitioning Keys with Partition Bounds

If you attempt to insert a row into a table and the row's partitioning key is greater than or equal to the partition bound for the highest partition in the table, the insert will fail.

When comparing character values in partitioning keys and partition bounds, characters are compared according to their binary values. However, if a character consists of more than one byte, Oracle compares the binary value of each byte, not of the character. The comparison also uses the comparison rules associated with the column data type. For example, blank-padded comparison is done for the ANSI CHAR data type. The NLS parameters, specifically the initialization parameters NLS_SORT and NLS_LANGUAGE and the environment variable NLS_LANG, have no effect on the comparison.

The binary value of character data varies depending on which character set is being used (for example, ASCII or EBCDIC). For example, ASCII defines the characters A through Z as less than the characters a through z, whereas EBCDIC defines A through Z as being greater than a through z. Thus, partitions designed for one sequence will not work with the other sequence. You must repartition the table after importing from a table using a different character set.

MAXVALUE

You can specify the keyword MAXVALUE for any value in the partition bound *value_list*. This keyword represents a virtual infinite value that sorts higher than any other value for the data type, including the NULL value.

For example, you might partition the OFFICE table on STATE (a CHAR(10) column) into three partitions with the following partition bounds:

- VALUES LESS THAN ('I'): States whose names start with A through H
- VALUES LESS THAN ('S'): States whose names start with I through R
- VALUES LESS THAN (MAXVALUE): States whose names start with S through Z, plus special codes for non-U.S. regions

Nulls

NULL cannot be specified as a value in a partition bound value_list. An empty string also cannot be specified as a value in a partition bound value_list, because it is treated as NULL within the database server.

For the purpose of assigning rows to partitions, Oracle Database sorts nulls greater than all other values except MAXVALUE. Nulls sort less than MAXVALUE.

This means that if a table is partitioned on a nullable column, and the column is to contain nulls, then the highest partition should have a partition bound of MAXVALUE for that column. Otherwise the rows that contain nulls will map above the highest partition in the table and the insert will fail.

DATE Datatypes

If the partition key includes a column that has the DATE datatype and the NLS date format does not specify the century with the year, you must specify partition bounds using the TO_DATE function with a 4-character format mask for the year. Otherwise, you will not be able to create the table or index. For example, with the sales_range table using a DATE column:

```
CREATE TABLE sales_range
(salesman_id  NUMBER(5),
salesman_name VARCHAR2(30),
sales_amount  NUMBER(10),
sales_date    DATE)
COMPRESS
PARTITION BY RANGE(sales_date)
(PARTITION sales_jan2000 VALUES LESS THAN(TO_DATE('02/01/2000','DD/MM/YYYY')),
 PARTITION sales_feb2000 VALUES LESS THAN(TO_DATE('03/01/2000','DD/MM/YYYY')),
 PARTITION sales_mar2000 VALUES LESS THAN(TO_DATE('04/01/2000','DD/MM/YYYY')),
 PARTITION sales_apr2000 VALUES LESS THAN(TO_DATE('05/01/2000','DD/MM/YYYY')));
```

When you query or modify data, it is recommended that you use the TO_DATE function in the WHERE clause so that the value of the date information can be

determined at compile time. However, the optimizer can prune partitions using a selection criterion on partitioning columns of type DATE when you use another format, as in the following examples:

```
SELECT * FROM sales_range
  WHERE sales_date BETWEEN TO_DATE('01-JUL-00', 'DD-MON-YY')
  AND TO_DATE('01-OCT-00', 'DD-MON-YY');

SELECT * FROM sales_range
  WHERE sales_date BETWEEN '01-JUL-2000' AND '01-OCT-2000';
```

In this case, the date value will be complete only at runtime. Therefore you will not be able to see which partitions Oracle is accessing as is usually shown on the partition_start and partition_stop columns of the EXPLAIN PLAN statement output on the SQL statement. Instead, you will see the keyword KEY for both columns.

Multicolumn Partitioning Keys

When a table or index is partitioned by range on multiple columns, each partition bound and partitioning key is a list (or vector) of values. The partition bounds and keys are ordered according to ANSI SQL2 vector comparison rules. This is also the way Oracle orders multicolumn index keys.

To compare a partitioning key with a partition bound, you compare the values of their corresponding columns until you find an unequal pair and then that pair determines which vector is greater. The values of any remaining columns have no effect on the comparison.

> **See Also:** *Oracle Database Administrator's Guide* for more information regarding multicolumn partitioning keys

Implicit Constraints Imposed by Partition Bounds

If you specify a partition bound other than MAXVALUE for the highest partition in a table, this imposes an implicit CHECK constraint on the table. This constraint is not recorded in the data dictionary, but the partition bound itself is recorded.

Index Partitioning

The rules for partitioning indexes are similar to those for tables:

- An index can be partitioned unless:
  - The index is a cluster index

  - The index is defined on a clustered table.
- You can mix partitioned and nonpartitioned indexes with partitioned and nonpartitioned tables:
  - A partitioned table can have partitioned or nonpartitioned indexes.
  - A nonpartitioned table can have partitioned or nonpartitioned B-tree indexes.
- Bitmap indexes on nonpartitioned tables cannot be partitioned.
- A bitmap index on a partitioned table must be a local index.

However, partitioned indexes are more complicated than partitioned tables because there are three types of partitioned indexes:

- Local prefixed
- Local nonprefixed
- Global prefixed

These types are described in the following section. Oracle supports all three types.

Local Partitioned Indexes

In a local index, all keys in a particular index partition refer only to rows stored in a single underlying table partition. A local index is created by specifying the `LOCAL` attribute.

Oracle constructs the local index so that it is equipartitioned with the underlying table. Oracle partitions the index on the same columns as the underlying table, creates the same number of partitions or subpartitions, and gives them the same partition bounds as corresponding partitions of the underlying table.

Oracle also maintains the index partitioning automatically when partitions in the underlying table are added, dropped, merged, or split, or when hash partitions or subpartitions are added or coalesced. This ensures that the index remains equipartitioned with the table.

A local index can be created `UNIQUE` if the partitioning columns form a subset of the index columns. This restriction guarantees that rows with identical index keys always map into the same partition, where uniqueness violations can be detected.

Local indexes have the following advantages:

- Only one index partition needs to be rebuilt when a maintenance operation other than SPLIT PARTITION or ADD PARTITION is performed on an underlying table partition.
- The duration of a partition maintenance operation remains proportional to partition size if the partitioned table has only local indexes.
- Local indexes support partition independence.
- Local indexes support smooth roll-out of old data and roll-in of new data in historical tables.
- Oracle can take advantage of the fact that a local index is equipartitioned with the underlying table to generate better query access plans.
- Local indexes simplify the task of tablespace incomplete recovery. In order to recover a partition or subpartition of a table to a point in time, you must also recover the corresponding index entries to the same point in time. The only way to accomplish this is with a local index. Then you can recover the corresponding table and index partitions or subpartitions together.

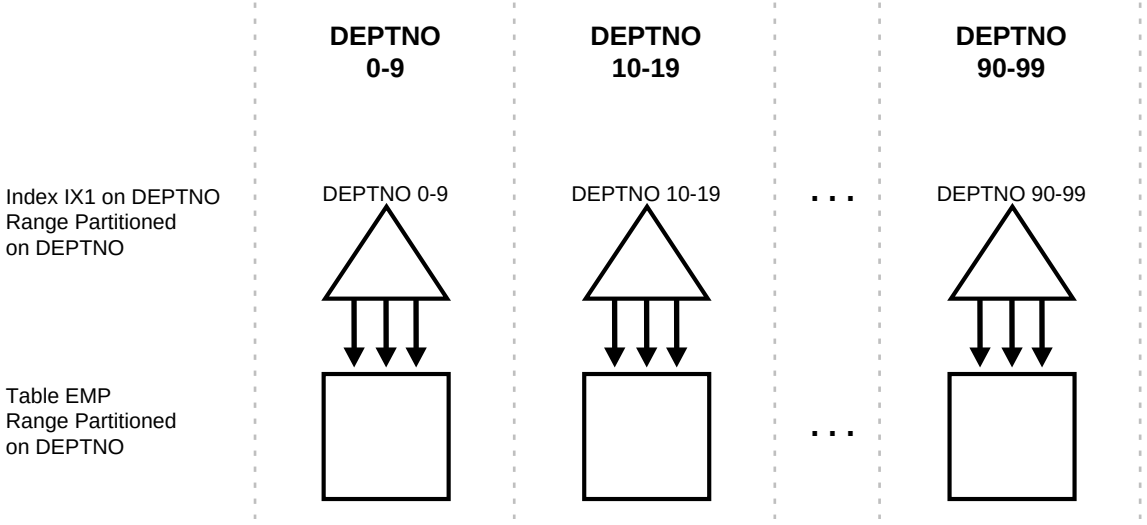
> **See Also:** *PL/SQL Packages and Types Reference* for a description of the DBMS_PCLXUTIL package

Local Prefixed Indexes A local index is prefixed if it is partitioned on a left prefix of the index columns. For example, if the sales table and its local index sales_ix are partitioned on the week_num column, then index sales_ix is local prefixed if it is defined on the columns (week_num, xaction_num). On the other hand, if index sales_ix is defined on column product_num then it is not prefixed.

Local prefixed indexes can be unique or nonunique.

Figure 5–4 illustrates another example of a local prefixed index.

Figure 5–4 Local Prefixed Index

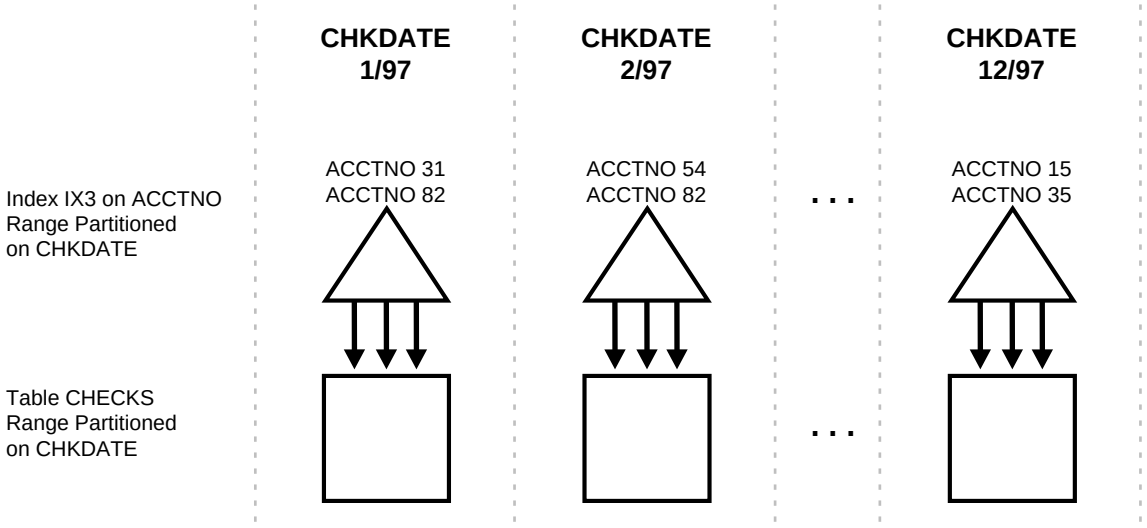


Local Nonprefixed Indexes A local index is nonprefixed if it is not partitioned on a left prefix of the index columns.

You cannot have a unique local nonprefixed index unless the partitioning key is a subset of the index key.

Figure 5–5 illustrates an example of a local nonprefixed index.

Figure 5–5 Local Nonprefixed Index



Global Partitioned Indexes

In a global partitioned index, the keys in a particular index partition may refer to rows stored in more than one underlying table partition or subpartition. A global index can be range or hash partitioned, though it can be defined on any type of partitioned table.

A global index is created by specifying the GLOBAL attribute. The database administrator is responsible for defining the initial partitioning of a global index at creation and for maintaining the partitioning over time. Index partitions can be merged or split as necessary.

Normally, a global index is not equipartitioned with the underlying table. There is nothing to prevent an index from being equipartitioned with the underlying table, but Oracle does not take advantage of the equipartitioning when generating query plans or executing partition maintenance operations. So an index that is equipartitioned with the underlying table should be created as LOCAL.

A global partitioned index contains a single B-tree with entries for all rows in all partitions. Each index partition may contain keys that refer to many different partitions or subpartitions in the table.

The highest partition of a global index must have a partition bound all of whose values are MAXVALUE. This insures that all rows in the underlying table can be represented in the index.

Prefixed and Nonprefixed Global Partitioned Indexes A global partitioned index is prefixed if it is partitioned on a left prefix of the index columns. See Figure 5–6 for an example. A global partitioned index is nonprefixed if it is not partitioned on a left prefix of the index columns. Oracle does not support global nonprefixed partitioned indexes.

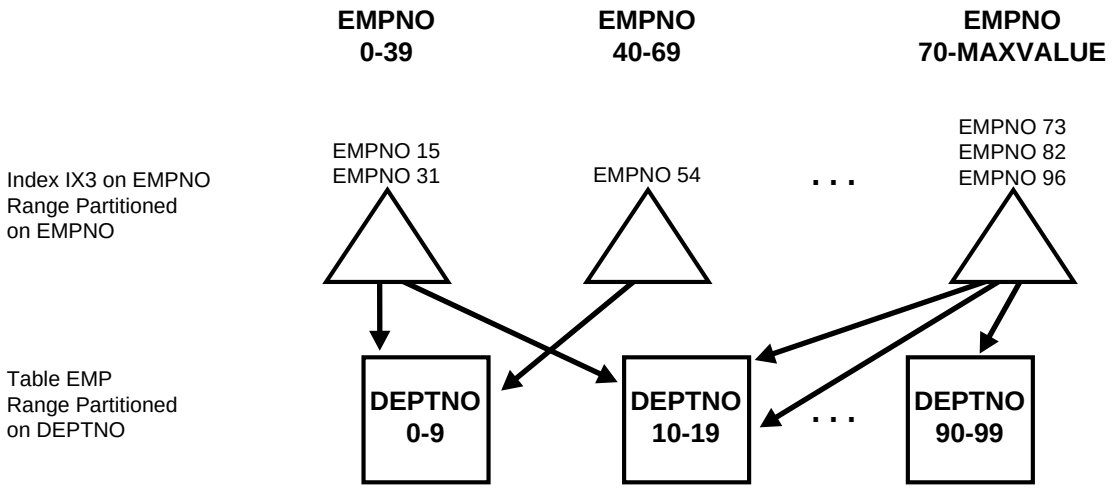
Global prefixed partitioned indexes can be unique or nonunique.

Nonpartitioned indexes are treated as global prefixed nonpartitioned indexes.

Management of Global Partitioned Indexes Global partitioned indexes are harder to manage than local indexes:

- When the data in an underlying table partition is moved or removed (SPLIT, MOVE, DROP, or TRUNCATE), all partitions of a global index are affected. Consequently global indexes do not support partition independence.
- When an underlying table partition or subpartition is recovered to a point in time, all corresponding entries in a global index must be recovered to the same point in time. Because these entries may be scattered across all partitions or subpartitions of the index, mixed in with entries for other partitions or subpartitions that are not being recovered, there is no way to accomplish this except by re-creating the entire global index.

Figure 5–6 Global Prefixed Partitioned Index



Summary of Partitioned Index Types

Table 5–2 summarizes the types of partitioned indexes that Oracle supports.

- If an index is local, it is equipartitioned with the underlying table. Otherwise, it is global.
- A prefixed index is partitioned on a left prefix of the index columns. Otherwise, it is nonprefixed.

Table 5–2 Types of Partitioned Indexes

Type of Index	Index Equipartitioned with Table	Index Partitioned on Left Prefix of Index Columns	UNIQUE Attribute Allowed	Example: Table Partitioning Key	Example: Index Columns	Example: Index Partitioning Key
Local Prefixed (any partitioning method)	Yes	Yes	Yes	A	A, B	A
Local Nonprefixed (any partitioning method)	Yes	No	Yes (Note1)	A	B, A	A
Global Prefixed (range partitioning only)	No (Note2)	Yes	Yes	A	B	B

Note 1: For a unique local nonprefixed index, the partitioning key must be a subset of the index key.

Note 2: Although a global partitioned index may be equipartitioned with the underlying table, Oracle does not take advantage of the partitioning or maintain equipartitioning after partition maintenance operations such as `DROP` or `SPLIT PARTITION`.

The Importance of Nonprefixed Indexes

Nonprefixed indexes are particularly useful in historical databases. In a table containing historical data, it is common for an index to be defined on one column to support the requirements of fast access by that column, but partitioned on another column (the same column as the underlying table) to support the time interval for rolling out old data and rolling in new data.

Consider a `sales` table partitioned by week. It contains a year's worth of data, divided into 13 partitions. It is range partitioned on `week_no`, four weeks to a partition. You might create a nonprefixed local index `sales_ix` on `sales`. The `sales_ix` index is defined on `acct_no` because there are queries that need fast access to the data by account number. However, it is partitioned on `week_no` to match the `sales` table. Every four weeks, the oldest partitions of `sales` and `sales_ix` are dropped and new ones are added.

Performance Implications of Prefixed and Nonprefixed Indexes

It is more expensive to probe into a nonprefixed index than to probe into a prefixed index.

If an index is prefixed (either local or global) and Oracle is presented with a predicate involving the index columns, then partition pruning can restrict application of the predicate to a subset of the index partitions.

For example, in Figure 5–4 on page 5-36, if the predicate is `deptno=15`, the optimizer knows to apply the predicate only to the second partition of the index. (If the predicate involves a bind variable, the optimizer will not know exactly which partition but it may still know there is only one partition involved, in which case at run time, only one index partition will be accessed.)

When an index is nonprefixed, Oracle often has to apply a predicate involving the index columns to all `N` index partitions. This is required to look up a single key, or to do an index range scan. For a range scan, Oracle must also combine information from `N` index partitions. For example, in Figure 5–5 on page 5-37, a local index is partitioned on `chkdate` with an index key on `acctno`. If the predicate is `acctno=31`, Oracle probes all 12 index partitions.

Of course, if there is also a predicate on the partitioning columns, then multiple index probes might not be necessary. Oracle takes advantage of the fact that a local index is equipartitioned with the underlying table to prune partitions based on the partition key. For example, if the predicate in Figure 5–4 on page 5-36 is `chkdate<3/97`, Oracle only has to probe two partitions.

So for a nonprefixed index, if the partition key is a part of the `WHERE` clause but not of the index key, then the optimizer determines which index partitions to probe based on the underlying table partition.

When many queries and DML statements using keys of local, nonprefixed, indexes have to probe all index partitions, this effectively reduces the degree of partition independence provided by such indexes.

Table 5–3 Comparing Prefixed Local, Nonprefixed Local, and Global Indexes

Index Characteristics	Prefixed Local	Nonprefixed Local	Global
Unique possible?	Yes	Yes	Yes. Must be global if using indexes on columns other than the partitioning columns
Manageability	Easy to manage	Easy to manage	Harder to manage
OLTP	Good	Bad	Good
Long Running (DSS)	Good	Good	Not Good

Guidelines for Partitioning Indexes

When deciding how to partition indexes on a table, consider the mix of applications that need to access the table. There is a trade-off between performance on the one hand and availability and manageability on the other. Here are some of the guidelines you should consider:

- For OLTP applications:
  - Global indexes and local prefixed indexes provide better performance than local nonprefixed indexes because they minimize the number of index partition probes.
  - Local indexes support more availability when there are partition or subpartition maintenance operations on the table. Local nonprefixed indexes are very useful for historical databases.

- For DSS applications, local nonprefixed indexes can improve performance because many index partitions can be scanned in parallel by range queries on the index key.

  For example, a query using the predicate "acctno between 40 and 45" on the table checks of Figure 5–4 on page 5-36 causes parallel scans of all the partitions of the nonprefixed index ix3. On the other hand, a query using the predicate "deptno BETWEEN 40 AND 45" on the table deptno of Figure 5–5 on page 5-37 cannot be parallelized because it accesses a single partition of the prefixed index ix1.

- For historical tables, indexes should be local if possible. This limits the impact of regularly scheduled drop partition operations.
- Unique indexes on columns other than the partitioning columns must be global because unique local nonprefixed indexes whose key does not contain the partitioning key are not supported.

Physical Attributes of Index Partitions

Default physical attributes are initially specified when a CREATE INDEX statement creates a partitioned index. Because there is no segment corresponding to the partitioned index itself, these attributes are only used in derivation of physical attributes of member partitions. Default physical attributes can later be modified using ALTER INDEX MODIFY DEFAULT ATTRIBUTES.

Physical attributes of partitions created by CREATE INDEX are determined as follows:

- Values of physical attributes specified (explicitly or by default) for the index are used whenever the value of a corresponding partition attribute is not specified. Handling of the TABLESPACE attribute of partitions of a LOCAL index constitutes an important exception to this rule in that in the absence of a user-specified TABLESPACE value (at both partition and index levels), that of the corresponding partition of the underlying table is used.
- Physical attributes (other than TABLESPACE, as explained in the preceding) of partitions of local indexes created in the course of processing ALTER TABLE ADD PARTITION are set to the default physical attributes of each index.

Physical attributes (other than TABLESPACE) of index partitions created by ALTER TABLE SPLIT PARTITION are determined as follows:

- Values of physical attributes of the index partition being split are used.

Physical attributes of an existing index partition can be modified by ALTER INDEX MODIFY PARTITION and ALTER INDEX REBUILD PARTITION. Resulting attributes are determined as follows:

- Values of physical attributes of the partition before the statement was issued are used whenever a new value is not specified. Note that ALTER INDEX REBUILD PARTITION can be used to change the tablespace in which a partition resides.

Physical attributes of global index partitions created by ALTER INDEX SPLIT PARTITION are determined as follows:

- Values of physical attributes of the partition being split are used whenever a new value is not specified.
- Physical attributes of all partitions of an index (along with default values) may be modified by ALTER INDEX, for example, ALTER INDEX indexname NOLOGGING changes the logging mode of all partitions of indexname to NOLOGGING.

> **See Also:** *Oracle Database Administrator's Guide* for more detailed examples of adding partitions and examples of rebuilding indexes

6

Indexes

This chapter describes how to use the following types of indexes in a data warehousing environment:

- Using Bitmap Indexes in Data Warehouses
- Using B-Tree Indexes in Data Warehouses
- Using Index Compression
- Choosing Between Local Indexes and Global Indexes

See Also: *Oracle Database Concepts* for general information regarding indexing

---

Note: Bitmap indexes are available only if you have purchased the Oracle Database Enterprise Edition.

Using Bitmap Indexes in Data Warehouses

Bitmap indexes are widely used in data warehousing environments. The environments typically have large amounts of data and ad hoc queries, but a low level of concurrent DML transactions. For such applications, bitmap indexing provides:

- Reduced response time for large classes of ad hoc queries.
- Reduced storage requirements compared to other indexing techniques.
- Dramatic performance gains even on hardware with a relatively small number of CPUs or a small amount of memory.
- Efficient maintenance during parallel DML and loads.

Fully indexing a large table with a traditional B-tree index can be prohibitively expensive in terms of space because the indexes can be several times larger than the data in the table. Bitmap indexes are typically only a fraction of the size of the indexed data in the table.

An index provides pointers to the rows in a table that contain a given key value. A regular index stores a list of rowids for each key corresponding to the rows with that key value. In a bitmap index, a bitmap for each key value replaces a list of rowids.

Each bit in the bitmap corresponds to a possible rowid, and if the bit is set, it means that the row with the corresponding rowid contains the key value. A mapping function converts the bit position to an actual rowid, so that the bitmap index provides the same functionality as a regular index. Bitmap indexes store the bitmaps in a compressed way. If the number of distinct key values is small, bitmap indexes compress better and the space saving benefit compared to a B-tree index becomes even better.

Bitmap indexes are most effective for queries that contain multiple conditions in the `WHERE` clause. Rows that satisfy some, but not all, conditions are filtered out before the table itself is accessed. This improves response time, often dramatically. If you are unsure of which indexes to create, the SQLAccess Advisor can generate recommendations on what to create.

Benefits for Data Warehousing Applications

Bitmap indexes are primarily intended for data warehousing applications where users query the data rather than update it. They are not suitable for OLTP applications with large numbers of concurrent transactions modifying the data.

Parallel query and parallel DML work with bitmap indexes. Bitmap indexing also supports parallel create indexes and concatenated indexes.

Bitmap indexes are required to take advantage of Oracle's star transformation capabilities.

> **See Also:** Chapter 19, "Schema Modeling Techniques" for further information about using bitmap indexes in data warehousing environments

Cardinality

The advantages of using bitmap indexes are greatest for columns in which the ratio of the number of distinct values to the number of rows in the table is small. We refer to this ratio as the **degree of cardinality**. A gender column, which has only two distinct values (male and female), is optimal for a bitmap index. However, data warehouse administrators also build bitmap indexes on columns with higher cardinalities.

For example, on a table with one million rows, a column with 10,000 distinct values is a candidate for a bitmap index. A bitmap index on this column can outperform a B-tree index, particularly when this column is often queried in conjunction with other indexed columns. In fact, in a typical data warehouse environments, a bitmap index can be considered for any non-unique column.

B-tree indexes are most effective for high-cardinality data: that is, for data with many possible values, such as `customer_name` or `phone_number`. In a data warehouse, B-tree indexes should be used only for unique columns or other columns with very high cardinalities (that is, columns that are almost unique). The majority of indexes in a data warehouse should be bitmap indexes.

In ad hoc queries and similar situations, bitmap indexes can dramatically improve query performance. `AND` and `OR` conditions in the `WHERE` clause of a query can be resolved quickly by performing the corresponding Boolean operations directly on the bitmaps before converting the resulting bitmap to rowids. If the resulting number of rows is small, the query can be answered quickly without resorting to a full table scan.

Example 6–1 Bitmap Index

The following shows a portion of a company's `customers` table.

```
SELECT cust_id, cust_gender, cust_marital_status, cust_income_level
FROM customers;
```

```
CUST_ID    C CUST_MARITAL_STATUS  CUST_INCOME_LEVEL
---------- - -------------------- --------------------
...
        70 F                      D: 70,000 - 89,999
        80 F married              H: 150,000 - 169,999
        90 M single               H: 150,000 - 169,999
       100 F                      I: 170,000 - 189,999
       110 F married              C: 50,000 - 69,999
       120 M single               F: 110,000 - 129,999
       130 M                      J: 190,000 - 249,999
       140 M married              G: 130,000 - 149,999
...
```

Because `cust_gender`, `cust_marital_status`, and cust_`income_level` are all low-cardinality columns (there are only three possible values for marital status and region, two possible values for gender, and 12 for income level), bitmap indexes are ideal for these columns. Do not create a bitmap index on `cust_id` because this is a unique column. Instead, a unique B-tree index on this column provides the most efficient representation and retrieval.

Table 6–1 illustrates the bitmap index for the `cust_gender` column in this example. It consists of two separate bitmaps, one for gender.

Table 6–1 Sample Bitmap Index

	gender='M'	gender='F'
cust_id 70	0	1
cust_id 80	0	1
cust_id 90	1	0
cust_id 100	0	1
cust_id 110	0	1
cust_id 120	1	0
cust_id 130	1	0
cust_id 140	1	0

Each entry (or bit) in the bitmap corresponds to a single row of the `customers` table. The value of each bit depends upon the values of the corresponding row in the table. For example, the bitmap `cust_gender='F'` contains a one as its first bit because the gender is `F` in the first row of the `customers` table. The bitmap `cust_gender='F'` has a zero for its third bit because the gender of the third row is not `F`.

An analyst investigating demographic trends of the company's customers might ask, "How many of our married customers have an income level of G or H?" This corresponds to the following SQL query:

```
SELECT COUNT(*) FROM customers
WHERE cust_marital_status = 'married'
AND cust_income_level IN ('H: 150,000 - 169,999', 'G: 130,000 - 149,999');
```

Bitmap indexes can efficiently process this query by merely counting the number of ones in the bitmap illustrated in Figure 6–1. The result set will be found by using bitmap or merge operations without the necessity of a conversion to rowids. To identify additional specific customer attributes that satisfy the criteria, use the resulting bitmap to access the table after a bitmap to rowid conversion.

Figure 6–1 Executing a Query Using Bitmap Indexes

Bitmap Indexes and Nulls

Unlike most other types of indexes, bitmap indexes include rows that have NULL values. Indexing of nulls can be useful for some types of SQL statements, such as queries with the aggregate function COUNT.

Example 6–2 Bitmap Index

```
SELECT COUNT(*) FROM customers WHERE cust_marital_status IS NULL;
```

This query uses a bitmap index on cust_marital_status. Note that this query would not be able to use a B-tree index, because B-tree indexes do not store the NULL values.

```
SELECT COUNT(*) FROM customers;
```

Any bitmap index can be used for this query because all table rows are indexed, including those that have NULL data. If nulls were not indexed, the optimizer would be able to use indexes only on columns with NOT NULL constraints.

Bitmap Indexes on Partitioned Tables

You can create bitmap indexes on partitioned tables but they must be local to the partitioned table—they cannot be global indexes. A partitioned table can only have global B-tree indexes, partitioned or non-partitioned. See "Index Partitioning" on page 5-9 for further information.

Using Bitmap Join Indexes in Data Warehouses

In addition to a bitmap index on a single table, you can create a bitmap join index, which is a bitmap index for the join of two or more tables. In a bitmap join index, the bitmap for the table to be indexed is built for values coming from the joined tables. In a data warehousing environment, the join condition is an equi-inner join between the primary key column or columns of the dimension tables and the foreign key column or columns in the fact table.

A bitmap join index can improve the performance by an order of magnitude. By storing the result of a join, the join can be avoided completely for SQL statements using a bitmap join index. Furthermore, since it is most likely to have a much smaller number of distinct values for a bitmap join index compared to a regular bitmap index on the join column, the bitmaps compress better, yielding to less space consumption than a regular bitmap join index on the join column.

Bitmap join indexes are much more efficient in storage than materialized join views, an alternative for materializing joins in advance. This is because the materialized join views do not compress the rowids of the fact tables.

Four Join Models for Bitmap Join Indexes

The most common usage of a bitmap join index is in star model environments, where a large table is indexed on columns joined by one or several smaller tables. We will refer to the large table as the fact table and to the smaller tables as dimension tables. The following section describes the four different join models supported by bitmap join indexes. See Chapter 19, "Schema Modeling Techniques" for schema modeling techniques.

Example 6–3 Bitmap Join Index: One Dimension Table Columns Joins One Fact Table

Unlike the example in "Bitmap Index" on page 6-3, where a bitmap index on the `cust_gender` column on the `customers` table was built, we now create a bitmap join index on the fact table `sales` for the joined column `customers(cust_gender)`. Table `sales` stores `cust_id` values only:

```
SELECT time_id, cust_id, amount_sold FROM sales;

TIME_ID   CUST_ID    AMOUNT_SOLD
--------- ---------- -----------
01-JAN-98      29700        2291
01-JAN-98       3380         114
01-JAN-98      67830         553
01-JAN-98     179330           0
01-JAN-98     127520         195
01-JAN-98      33030         280
...
```

To create such a bitmap join index, column `customers(cust_gender)` has to be joined with table `sales`. The join condition is specified as part of the `CREATE` statement for the bitmap join index as follows:

```
CREATE BITMAP INDEX sales_cust_gender_bjix
ON sales(customers.cust_gender)
FROM sales, customers
WHERE sales.cust_id = customers.cust_id
LOCAL;
```

The following query shows illustrates the join result that is used to create the bitmaps that are stored in the bitmap join index:

```
SELECT sales.time_id, customers.cust_gender, sales.amount_sold
FROM sales, customers
WHERE sales.cust_id = customers.cust_id;

TIME_ID   C AMOUNT_SOLD
--------- - -----------
01-JAN-98 M        2291
01-JAN-98 F         114
01-JAN-98 M         553
01-JAN-98 M           0
01-JAN-98 M         195
01-JAN-98 M         280
01-JAN-98 M          32
...
```

Table 6–2 illustrates the bitmap representation for the bitmap join index in this example.

Table 6–2 Sample Bitmap Join Index

	cust_gender='M'	cust_gender='F'
sales record 1	1	0
sales record 2	0	1
sales record 3	1	0
sales record 4	1	0
sales record 5	1	0
sales record 6	1	0
sales record 7	1	0

You can create other bitmap join indexes using more than one column or more than one table, as shown in these examples.

Example 6–4 Bitmap Join Index: Multiple Dimension Columns Join One Fact Table

You can create a bitmap join index on more than one column from a single dimension table, as in the following example, which uses `customers(cust_gender, cust_marital_status)` from the `sh` schema:

```
CREATE BITMAP INDEX sales_cust_gender_ms_bjix
ON sales(customers.cust_gender, customers.cust_marital_status)
FROM sales, customers
WHERE sales.cust_id = customers.cust_id
LOCAL NOLOGGING;
```

Example 6–5 Bitmap Join Index: Multiple Dimension Tables Join One Fact Table

You can create a bitmap join index on multiple dimension tables, as in the following, which uses `customers(gender)` and `products(category)`:

```
CREATE BITMAP INDEX sales_c_gender_p_cat_bjix
ON sales(customers.cust_gender, products.prod_category)
FROM sales, customers, products
WHERE sales.cust_id = customers.cust_id
AND sales.prod_id = products.prod_id
LOCAL NOLOGGING;
```

Example 6–6 Bitmap Join Index: Snowflake Schema

You can create a bitmap join index on more than one table, in which the indexed column is joined to the indexed table by using another table. For example, we can build an index on `countries.country_name`, even though the `countries` table is not joined directly to the `sales` table. Instead, the `countries` table is joined to the `customers` table, which is joined to the sales table. This type of schema is commonly called a **snowflake schema**.

```
CREATE BITMAP INDEX sales_co_country_name_bjix
ON sales(countries.country_name)
FROM sales, customers, countries
WHERE sales.cust_id = customers.cust_id
AND customers.country_id = countries.country_id
LOCAL NOLOGGING COMPUTE STATISTICS;
```

Bitmap Join Index Restrictions and Requirements

Join results must be stored, therefore, bitmap join indexes have the following restrictions:

- Parallel DML is currently only supported on the fact table. Parallel DML on one of the participating dimension tables will mark the index as unusable.
- Only one table can be updated concurrently by different transactions when using the bitmap join index.
- No table can appear twice in the join.
- You cannot create a bitmap join index on an index-organized table or a temporary table.
- The columns in the index must all be columns of the dimension tables.
- The dimension table join columns must be either primary key columns or have unique constraints.
- The dimension table column(s) participating the join with the fact table must be either the primary key column(s) or with the unique constraint.
- If a dimension table has composite primary key, each column in the primary key must be part of the join.
- The current restrictions for creating a regular bitmap index also apply to a bitmap join index. For example, you cannot create a bitmap index with the `UNIQUE` attribute. See *Oracle Database SQL Reference* for other restrictions.

Using B-Tree Indexes in Data Warehouses

A B-tree index is organized like an upside-down tree. The bottom level of the index holds the actual data values and pointers to the corresponding rows, much as the index in a book has a page number associated with each index entry.

In general, use B-tree indexes when you know that your typical query refers to the indexed column and retrieves a few rows. In these queries, it is faster to find the rows by looking at the index. However, using the book index analogy, if you plan to look at every single topic in a book, you might not want to look in the index for the topic and then look up the page. It might be faster to read through every chapter in the book. Similarly, if you are retrieving most of the rows in a table, it might not make sense to look up the index to find the table rows. Instead, you might want to read or scan the table.

B-tree indexes are most commonly used in a data warehouse to index unique or near-unique keys. In many cases, it may not be necessary to index these columns in a data warehouse, because unique constraints can be maintained without an index, and because typical data warehouse queries may not work better with such indexes. B-tree indexes are more common in environments using third normal form schemas. In general, bitmap indexes should be more common than B-tree indexes in most data warehouse environments.

Using Index Compression

Bitmap indexes are always stored in a patented, compressed manner without the need of any user intervention. B-tree indexes, however, can be stored specifically in a compressed manner to enable huge space savings, storing more keys in each index block, which also leads to less I/O and better performance.

Key compression lets you compress a b-tree index, which reduces the storage overhead of repeated values. In the case of a nonunique index, all index columns can be stored in a compressed format, whereas in the case of a unique index, at least one index column has to be stored uncompressed.

Generally, keys in an index have two pieces, a grouping piece and a unique piece. If the key is not defined to have a unique piece, Oracle provides one in the form of a rowid appended to the grouping piece. Key compression is a method of breaking off the grouping piece and storing it so it can be shared by multiple unique pieces. The cardinality of the chosen columns to be compressed determines the compression ratio that can be achieved. So, for example, if a unique index that consists of five columns provides the uniqueness mostly by the last two columns, it is most optimal to choose the three leading columns to be stored compressed. If you

choose to compress four columns, the repetitiveness will be almost gone, and the compression ratio will be worse.

Although key compression reduces the storage requirements of an index, it can increase the CPU time required to reconstruct the key column values during an index scan. It also incurs some additional storage overhead, because every prefix entry has an overhead of four bytes associated with it.

Choosing Between Local Indexes and Global Indexes

B-tree indexes on partitioned tables can be global or local. With Oracle8*i* and earlier releases, Oracle recommended that global indexes not be used in data warehouse environments because a partition DDL statement (for example, `ALTER TABLE ... DROP PARTITION`) would invalidate the entire index, and rebuilding the index is expensive. In Oracle Database 10*g*, global indexes can be maintained without Oracle marking them as unusable after DDL. This enhancement makes global indexes more effective for data warehouse environments.

However, local indexes will be more common than global indexes. Global indexes should be used when there is a specific requirement which cannot be met by local indexes (for example, a unique index on a non-partitioning key, or a performance requirement).

Bitmap indexes on partitioned tables are always local.

See Also: "Types of Partitioning" on page 5-4 for further details

7

Integrity Constraints

This chapter describes integrity constraints, and discusses:

- Why Integrity Constraints are Useful in a Data Warehouse
- Overview of Constraint States
- Typical Data Warehouse Integrity Constraints

Why Integrity Constraints are Useful in a Data Warehouse

Integrity constraints provide a mechanism for ensuring that data conforms to guidelines specified by the database administrator. The most common types of constraints include:

- `UNIQUE` constraints

  To ensure that a given column is unique

- `NOT NULL` constraints

  To ensure that no null values are allowed

- `FOREIGN KEY` constraints

  To ensure that two keys share a primary key to foreign key relationship

Constraints can be used for these purposes in a data warehouse:

- Data cleanliness

  Constraints verify that the data in the data warehouse conforms to a basic level of data consistency and correctness, preventing the introduction of dirty data.

- Query optimization

  The Oracle Database utilizes constraints when optimizing SQL queries. Although constraints can be useful in many aspects of query optimization, constraints are particularly important for query rewrite of materialized views.

Unlike data in many relational database environments, data in a data warehouse is typically added or modified under controlled circumstances during the extraction, transformation, and loading (ETL) process. Multiple users normally do not update the data warehouse directly, as they do in an OLTP system.

> **See Also:** Chapter 11, "Overview of Extraction, Transformation, and Loading"

Many significant constraint features have been introduced for data warehousing. Readers familiar with Oracle's constraint functionality in Oracle database version 7 and Oracle database version 8.x should take special note of the functionality described in this chapter. In fact, many Oracle database version7-based and Oracle database version8-based data warehouses lacked constraints because of concerns about constraint performance. Newer constraint functionality addresses these concerns.

Overview of Constraint States

To understand how best to use constraints in a data warehouse, you should first understand the basic purposes of constraints. Some of these purposes are:

- Enforcement

  In order to use a constraint for enforcement, the constraint must be in the ENABLE state. An enabled constraint ensures that all data modifications upon a given table (or tables) satisfy the conditions of the constraints. Data modification operations which produce data that violates the constraint fail with a constraint violation error.

- Validation

  To use a constraint for validation, the constraint must be in the VALIDATE state. If the constraint is validated, then all data that currently resides in the table satisfies the constraint.

  Note that validation is independent of enforcement. Although the typical constraint in an operational system is both enabled and validated, any constraint could be validated but not enabled or vice versa (enabled but not validated). These latter two cases are useful for data warehouses.

- Belief

  In some cases, you will know that the conditions for a given constraint are true, so you do not need to validate or enforce the constraint. However, you may wish for the constraint to be present anyway to improve query optimization and performance. When you use a constraint in this way, it is called a belief or RELY constraint, and the constraint must be in the RELY state. The RELY state provides you with a mechanism for telling Oracle that a given constraint is believed to be true.

  Note that the RELY state only affects constraints that have not been validated.

Typical Data Warehouse Integrity Constraints

This section assumes that you are familiar with the typical use of constraints. That is, constraints that are both enabled and validated. For data warehousing, many users have discovered that such constraints may be prohibitively costly to build and maintain. The topics discussed are:

- UNIQUE Constraints in a Data Warehouse
- FOREIGN KEY Constraints in a Data Warehouse

- RELY Constraints
- Integrity Constraints and Parallelism
- Integrity Constraints and Partitioning
- View Constraints

UNIQUE Constraints in a Data Warehouse

A UNIQUE constraint is typically enforced using a UNIQUE index. However, in a data warehouse whose tables can be extremely large, creating a unique index can be costly both in processing time and in disk space.

Suppose that a data warehouse contains a table sales, which includes a column sales_id. sales_id uniquely identifies a single sales transaction, and the data warehouse administrator must ensure that this column is unique within the data warehouse.

One way to create the constraint is as follows:

```
ALTER TABLE sales ADD CONSTRAINT sales_uk
UNIQUE (prod_id, cust_id, promo_id, channel_id, time_id);
```

By default, this constraint is both enabled and validated. Oracle implicitly creates a unique index on sales_id to support this constraint. However, this index can be problematic in a data warehouse for three reasons:

- The unique index can be very large, because the sales table can easily have millions or even billions of rows.
- The unique index is rarely used for query execution. Most data warehousing queries do not have predicates on unique keys, so creating this index will probably not improve performance.
- If sales is partitioned along a column other than sales_id, the unique index must be global. This can detrimentally affect all maintenance operations on the sales table.

A unique index is required for unique constraints to ensure that each individual row modified in the sales table satisfies the UNIQUE constraint.

For data warehousing tables, an alternative mechanism for unique constraints is illustrated in the following statement:

```
ALTER TABLE sales ADD CONSTRAINT sales_uk
UNIQUE (prod_id, cust_id, promo_id, channel_id, time_id) DISABLE VALIDATE;
```

This statement creates a unique constraint, but, because the constraint is disabled, a unique index is not required. This approach can be advantageous for many data warehousing environments because the constraint now ensures uniqueness without the cost of a unique index.

However, there are trade-offs for the data warehouse administrator to consider with DISABLE VALIDATE constraints. Because this constraint is disabled, no DML statements that modify the unique column are permitted against the sales table. You can use one of two strategies for modifying this table in the presence of a constraint:

- Use DDL to add data to this table (such as exchanging partitions). See the example in Chapter 15, "Maintaining the Data Warehouse".
- Before modifying this table, drop the constraint. Then, make all necessary data modifications. Finally, re-create the disabled constraint. Re-creating the constraint is more efficient than re-creating an enabled constraint. However, this approach does not guarantee that data added to the sales table while the constraint has been dropped is unique.

FOREIGN KEY Constraints in a Data Warehouse

In a star schema data warehouse, FOREIGN KEY constraints validate the relationship between the fact table and the dimension tables. A sample constraint might be:

```
ALTER TABLE sales ADD CONSTRAINT sales_time_fk
FOREIGN KEY (time_id) REFERENCES times (time_id)
ENABLE VALIDATE;
```

However, in some situations, you may choose to use a different state for the FOREIGN KEY constraints, in particular, the ENABLE NOVALIDATE state. A data warehouse administrator might use an ENABLE NOVALIDATE constraint when either:

- The tables contain data that currently disobeys the constraint, but the data warehouse administrator wishes to create a constraint for future enforcement.
- An enforced constraint is required immediately.

Suppose that the data warehouse loaded new data into the fact tables every day, but refreshed the dimension tables only on the weekend. During the week, the dimension tables and fact tables may in fact disobey the FOREIGN KEY constraints. Nevertheless, the data warehouse administrator might wish to maintain the enforcement of this constraint to prevent any changes that might affect the

FOREIGN KEY constraint outside of the ETL process. Thus, you can create the FOREIGN KEY constraints every night, after performing the ETL process, as shown here:

```
ALTER TABLE sales ADD CONSTRAINT sales_time_fk
FOREIGN KEY (time_id) REFERENCES times (time_id)
ENABLE NOVALIDATE;
```

ENABLE NOVALIDATE can quickly create an enforced constraint, even when the constraint is believed to be true. Suppose that the ETL process verifies that a FOREIGN KEY constraint is true. Rather than have the database re-verify this FOREIGN KEY constraint, which would require time and database resources, the data warehouse administrator could instead create a FOREIGN KEY constraint using ENABLE NOVALIDATE.

RELY Constraints

The ETL process commonly verifies that certain constraints are true. For example, it can validate all of the foreign keys in the data coming into the fact table. This means that you can trust it to provide clean data, instead of implementing constraints in the data warehouse. You create a RELY constraint as follows:

```
ALTER TABLE sales ADD CONSTRAINT sales_time_fk
FOREIGN KEY (time_id) REFERENCES times (time_id)
RELY DISABLE NOVALIDATE;
```

This statement assumes that the primary key is in the RELY state. RELY constraints, even though they are not used for data validation, can:

- Enable more sophisticated query rewrites for materialized views. See Chapter 18, "Query Rewrite" for further details.
- Enable other data warehousing tools to retrieve information regarding constraints directly from the Oracle data dictionary.

Creating a RELY constraint is inexpensive and does not impose any overhead during DML or load. Because the constraint is not being validated, no data processing is necessary to create it.

Integrity Constraints and Parallelism

All constraints can be validated in parallel. When validating constraints on very large tables, parallelism is often necessary to meet performance goals. The degree of

parallelism for a given constraint operation is determined by the default degree of parallelism of the underlying table.

Integrity Constraints and Partitioning

You can create and maintain constraints before you partition the data. Later chapters discuss the significance of partitioning for data warehousing. Partitioning can improve constraint management just as it does to management of many other operations. For example, Chapter 15, "Maintaining the Data Warehouse" provides a scenario creating `UNIQUE` and `FOREIGN KEY` constraints on a separate staging table, and these constraints are maintained during the `EXCHANGE PARTITION` statement.

View Constraints

You can create constraints on views. The only type of constraint supported on a view is a `RELY` constraint.

This type of constraint is useful when queries typically access views instead of base tables, and the database administrator thus needs to define the data relationships between views rather than tables. View constraints are particularly useful in OLAP environments, where they may enable more sophisticated rewrites for materialized views.

> **See Also:** Chapter 8, "Basic Materialized Views" and Chapter 18, "Query Rewrite"

8

Basic Materialized Views

This chapter introduces you to the use of materialized views, and discusses:

Overview of Data Warehousing with Materialized Views

Typically, data flows from one or more online transaction processing (OLTP) database into a data warehouse on a monthly, weekly, or daily basis. The data is normally processed in a **staging file** before being added to the data warehouse. Data warehouses commonly range in size from tens of gigabytes to a few terabytes. Usually, the vast majority of the data is stored in a few very large fact tables.

One technique employed in data warehouses to improve performance is the creation of summaries. Summaries are special types of aggregate views that improve query execution times by precalculating expensive joins and aggregation operations prior to execution and storing the results in a table in the database. For example, you can create a table to contain the sums of sales by region and by product.

The summaries or aggregates that are referred to in this book and in literature on data warehousing are created in Oracle Database using a schema object called a **materialized view**. Materialized views can perform a number of roles, such as improving query performance or providing replicated data.

In the past, organizations using summaries spent a significant amount of time and effort creating summaries manually, identifying which summaries to create, indexing the summaries, updating them, and advising their users on which ones to use. The introduction of summary management eased the workload of the database administrator and meant the user no longer needed to be aware of the summaries that had been defined. The database administrator creates one or more materialized views, which are the equivalent of a summary. The end user queries the tables and views at the detail data level. The query rewrite mechanism in the Oracle server automatically rewrites the SQL query to use the summary tables. This mechanism reduces response time for returning results from the query. Materialized views within the data warehouse are transparent to the end user or to the database application.

Although materialized views are usually accessed through the query rewrite mechanism, an end user or database application can construct queries that directly access the materialized views. However, serious consideration should be given to whether users should be allowed to do this because any change to the materialized views will affect the queries that reference them.

Materialized Views for Data Warehouses

In data warehouses, you can use materialized views to precompute and store aggregated data such as the sum of sales. Materialized views in these environments

are often referred to as summaries, because they store summarized data. They can also be used to precompute joins with or without aggregations. A materialized view eliminates the overhead associated with expensive joins and aggregations for a large or important class of queries.

Materialized Views for Distributed Computing

In distributed environments, you can use materialized views to replicate data at distributed sites and to synchronize updates done at those sites with conflict resolution methods. The materialized views as replicas provide local access to data that otherwise would have to be accessed from remote sites. Materialized views are also useful in remote data marts. See *Oracle Database Advanced Replication* and *Oracle Database Heterogeneous Connectivity Administrator's Guide* for details on distributed and mobile computing.

Materialized Views for Mobile Computing

You can also use materialized views to download a subset of data from central servers to mobile clients, with periodic refreshes and updates between clients and the central servers.

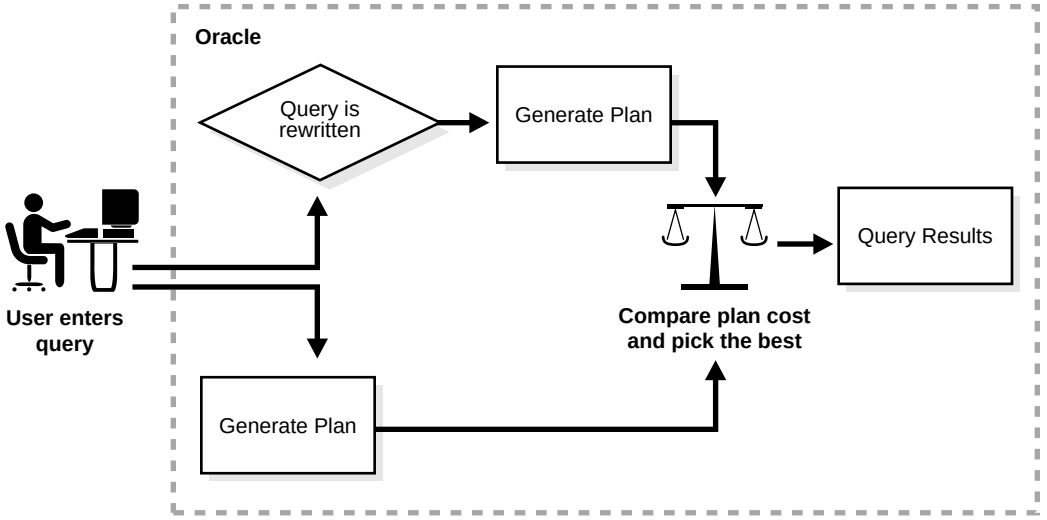
This chapter focuses on the use of materialized views in data warehouses. See *Oracle Database Advanced Replication* and *Oracle Database Heterogeneous Connectivity Administrator's Guide* for details on distributed and mobile computing.

The Need for Materialized Views

You can use materialized views to increase the speed of queries on very large databases. Queries to large databases often involve joins between tables, aggregations such as `SUM`, or both. These operations are expensive in terms of time and processing power. The type of materialized view you create determines how the materialized view is refreshed and used by query rewrite.

Materialized views improve query performance by precalculating expensive join and aggregation operations on the database prior to execution and storing the results in the database. The query optimizer automatically recognizes when an existing materialized view can and should be used to satisfy a request. It then transparently rewrites the request to use the materialized view. Queries go directly to the materialized view and not to the underlying detail tables. In general, rewriting queries to use materialized views rather than detail tables improves response. Figure 8–1 illustrates how query rewrite works.

Figure 8–1 Transparent Query Rewrite



When using query rewrite, create materialized views that satisfy the largest number of queries. For example, if you identify 20 queries that are commonly applied to the detail or fact tables, then you might be able to satisfy them with five or six well-written materialized views. A materialized view definition can include any number of aggregations (`SUM`, `COUNT(x)`, `COUNT(*)`, `COUNT(DISTINCT x)`, `AVG`, `VARIANCE`, `STDDEV`, `MIN`, and `MAX`). It can also include any number of joins. If you are unsure of which materialized views to create, Oracle provides the SQLAccess Advisor, which is a set of advisory procedures in the `DBMS_ADVISOR` package to help in designing and evaluating materialized views for query rewrite. See Chapter 17, "SQLAccess Advisor" for further details.

If a materialized view is to be used by query rewrite, it must be stored in the same database as the detail tables on which it relies. A materialized view can be partitioned, and you can define a materialized view on a partitioned table. You can also define one or more indexes on the materialized view.

Unlike indexes, materialized views can be accessed directly using a `SELECT` statement. However, it is recommended that you try to avoid writing SQL statements that directly reference the materialized view, because then it is difficult to change them without affecting the application. Instead, let query rewrite transparently rewrite your query to use the materialized view.

Note that the techniques shown in this chapter illustrate how to use materialized views in data warehouses. Materialized views can also be used by Oracle Replication. See *Oracle Database Advanced Replication* for further information.

Components of Summary Management

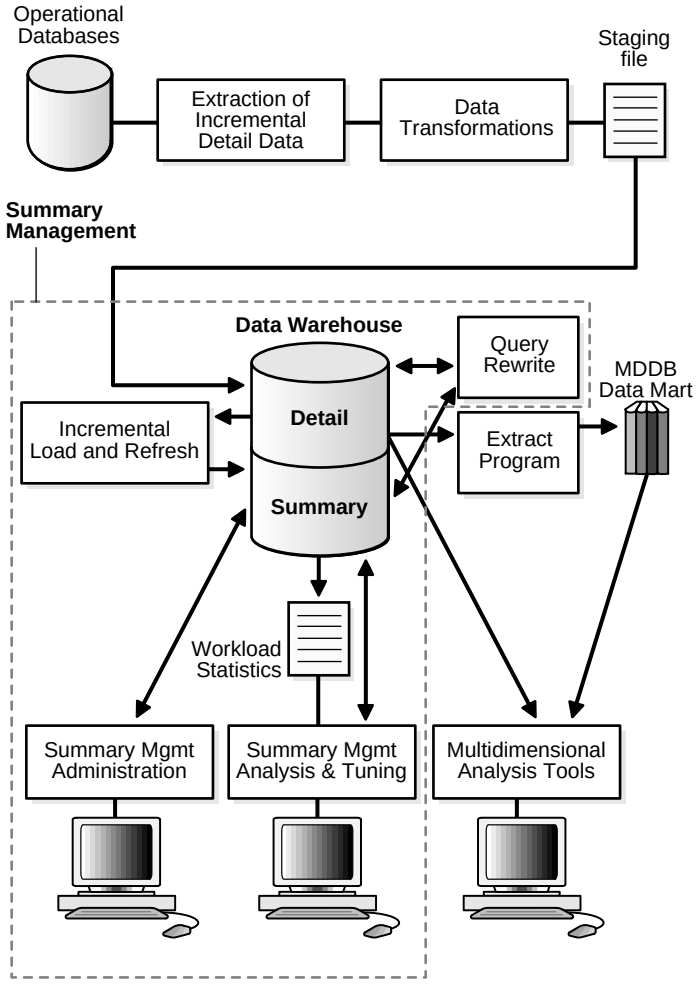
Summary management consists of:

- Mechanisms to define materialized views and dimensions.
- A refresh mechanism to ensure that all materialized views contain the latest data.
- A query rewrite capability to transparently rewrite a query to use a materialized view.
- The SQLAccess Advisor, which recommends materialized views and indexes to create. See Chapter 17, "SQLAccess Advisor" for more information.
- `TUNE_MVIEW`, which shows you how to make your materialized view fast refreshable and use general query rewrite.

The use of summary management features imposes no schema restrictions, and can enable some existing DSS database applications to improve performance without the need to redesign the database or the application.

Figure 8–2 illustrates the use of summary management in the warehousing cycle. After the data has been transformed, staged, and loaded into the detail data in the warehouse, you can invoke the summary management process. First, use the SQLAccess Advisor to plan how you will use materialized views. Then, create materialized views and design how queries will be rewritten. If you are having problems trying to get your materialized views to work then use `TUNE_MVIEW` to obtain an optimzed materialized view.

Figure 8–2 Overview of Summary Management



Understanding the summary management process during the earliest stages of data warehouse design can yield large dividends later in the form of higher performance, lower summary administration costs, and reduced storage requirements.

Data Warehousing Terminology

Some basic data warehousing terms are defined as follows:

- **Dimension tables** describe the business entities of an enterprise, represented as hierarchical, categorical information such as time, departments, locations, and products. Dimension tables are sometimes called lookup or reference tables.

  Dimension tables usually change slowly over time and are not modified on a periodic schedule. They are used in long-running decision support queries to aggregate the data returned from the query into appropriate levels of the dimension hierarchy.

- **Hierarchies** describe the business relationships and common access patterns in the database. An analysis of the dimensions, combined with an understanding of the typical work load, can be used to create materialized views. See Chapter 10, "Dimensions" for more information.

- **Fact tables** describe the business transactions of an enterprise.

  The vast majority of data in a data warehouse is stored in a few very large fact tables that are updated periodically with data from one or more operational OLTP databases.

  Fact tables include facts (also called measures) such as sales, units, and inventory.

  – A simple measure is a numeric or character column of one table such as `fact.sales`.

  – A computed measure is an expression involving measures of one table, for example, `fact.revenues - fact.expenses`.

  – A multitable measure is a computed measure defined on multiple tables, for example, `fact_a.revenues - fact_b.expenses`.

  Fact tables also contain one or more foreign keys that organize the business transactions by the relevant business entities such as time, product, and market. In most cases, these foreign keys are non-null, form a unique compound key of the fact table, and each foreign key joins with exactly one row of a **dimension table**.

- A materialized view is a precomputed table comprising aggregated and joined data from fact and possibly from dimension tables. Among builders of data warehouses, a materialized view is also known as a **summary**.

Materialized View Schema Design

Summary management can perform many useful functions, including query rewrite and materialized view refresh, even if your data warehouse design does not follow these guidelines. However, you will realize significantly greater query execution performance and materialized view refresh performance benefits and you will require fewer materialized views if your schema design complies with these guidelines.

A materialized view definition includes any number of aggregates, as well as any number of joins. In several ways, a materialized view behaves like an index:

- The purpose of a materialized view is to increase query execution performance.
- The existence of a materialized view is transparent to SQL applications, so that a database administrator can create or drop materialized views at any time without affecting the validity of SQL applications.
- A materialized view consumes storage space.
- The contents of the materialized view must be updated when the underlying detail tables are modified.

Schemas and Dimension Tables

In the case of normalized or partially normalized dimension tables (a dimension that is stored in more than one table), identify how these tables are joined. Note whether the joins between the dimension tables can guarantee that each child-side row joins with one and only one parent-side row. In the case of denormalized dimensions, determine whether the child-side columns uniquely determine the parent-side (or attribute) columns. These relationships can be enabled with constraints, using the `NOVALIDATE` and `RELY` options if the relationships represented by the constraints are guaranteed by other means. Note that if the joins between fact and dimension tables do not support the parent-child relationship described previously, you still gain significant performance advantages from defining the dimension with the `CREATE DIMENSION` statement. Another alternative, subject to some restrictions, is to use outer joins in the materialized view definition (that is, in the `CREATE MATERIALIZED VIEW` statement).

You must not create dimensions in any schema that does not satisfy these relationships. Incorrect results can be returned from queries otherwise.

Materialized View Schema Design Guidelines

Before starting to define and use the various components of summary management, you should review your schema design to abide by the following guidelines wherever possible.

Guidelines 1 and 2 are more important than guideline 3. If your schema design does not follow guidelines 1 and 2, it does not then matter whether it follows guideline 3. Guidelines 1, 2, and 3 affect both query rewrite performance and materialized view refresh performance.

Table 8–1 Schema Design Guidelines

Schema Guideline	Description
Guideline 1 Dimensions	Dimensions should either be denormalized (each dimension contained in one table) or the joins between tables in a normalized or partially normalized dimension should guarantee that each child-side row joins with exactly one parent-side row. The benefits of maintaining this condition are described in "Creating Dimensions" on page 10-4. You can enforce this condition by adding `FOREIGN KEY` and `NOT NULL` constraints on the child-side join keys and `PRIMARY KEY` constraints on the parent-side join keys.
Guideline 2 Dimensions	If dimensions are denormalized or partially denormalized, hierarchical integrity must be maintained between the key columns of the dimension table. Each child key value must uniquely identify its parent key value, even if the dimension table is denormalized. Hierarchical integrity in a denormalized dimension can be verified by calling the `VALIDATE_DIMENSION` procedure of the `DBMS_DIMENSION` package.
Guideline 3 Dimensions	Fact and dimension tables should similarly guarantee that each fact table row joins with exactly one dimension table row. This condition must be declared, and optionally enforced, by adding `FOREIGN KEY` and `NOT NULL` constraints on the fact key column(s) and `PRIMARY KEY` constraints on the dimension key column(s), or by using outer joins. In a data warehouse, constraints are typically enabled with the `NOVALIDATE` and `RELY` clauses to avoid constraint enforcement performance overhead. See *Oracle Database SQL Reference* for further details.
Guideline 4 Incremental Loads	Incremental loads of your detail data should be done using the SQL*Loader direct-path option, or any bulk loader utility that uses Oracle's direct-path interface. This includes `INSERT ... AS SELECT` with the `APPEND` or `PARALLEL` hints, where the hints cause the direct loader log to be used during the insert. See *Oracle Database SQL Reference* and "Types of Materialized Views" on page 8-12 for more information.
Guideline 5 Partitions	Range / composite partition your tables by a monotonically increasing time column if possible (preferably of type `DATE`).
Guideline 6 Dimensions	After each load and before refreshing your materialized view, use the `VALIDATE_DIMENSION` procedure of the `DBMS_DIMENSION` package to incrementally verify dimensional integrity.
Guideline 7 Time Dimensions	If a time dimension appears in the materialized view as a time column, partition and index the materialized view in the same manner as you have the fact tables.

If you are concerned with the time required to enable constraints and whether any constraints might be violated, then use the `ENABLE NOVALIDATE` with the `RELY`

clause to turn on constraint checking without validating any of the existing constraints. The risk with this approach is that incorrect query results could occur if any constraints are broken. Therefore, as the designer, you must determine how clean the data is and whether the risk of incorrect results is too great.

Loading Data into Data Warehouses

A popular and efficient way to load data into a data warehouse or data mart is to use SQL*Loader with the DIRECT or PARALLEL option, DataPump, or to use another loader tool that uses the Oracle direct-path API. See *Oracle Database Utilities* for the restrictions and considerations when using SQL*Loader with the DIRECT or PARALLEL keywords.

Loading strategies can be classified as one-phase or two-phase. In one-phase loading, data is loaded directly into the target table, quality assurance tests are performed, and errors are resolved by performing DML operations prior to refreshing materialized views. If a large number of deletions are possible, then storage utilization can be adversely affected, but temporary space requirements and load time are minimized.

In a two-phase loading process:

- Data is first loaded into a temporary table in the warehouse.
- Quality assurance procedures are applied to the data.
- Referential integrity constraints on the target table are disabled, and the local index in the target partition is marked unusable.
- The data is copied from the temporary area into the appropriate partition of the target table using INSERT AS SELECT with the PARALLEL or APPEND hint. The temporary table is then dropped. Alternatively, if the target table is partitioned, you can create a new (empty) partition in the target table and use ALTER TABLE ... EXCHANGE PARTITION to incorporate the temporary table into the target table. See *Oracle Database SQL Reference* for more information.
- The constraints are enabled, usually with the NOVALIDATE option.

Immediately after loading the detail data and updating the indexes on the detail data, the database can be opened for operation, if desired. You can disable query rewrite at the system level by issuing an ALTER SYSTEM SET QUERY_REWRITE_ENABLED = FALSE statement until all the materialized views are refreshed.

If QUERY_REWRITE_INTEGRITY is set to STALE_TOLERATED, access to the materialized view can be allowed at the session level to any users who do not require the materialized views to reflect the data from the latest load by issuing an

`ALTER SESSION SET QUERY_REWRITE_ENABLED=TRUE` statement. This scenario does not apply when `QUERY_REWRITE_INTEGRITY` is either `ENFORCED` or `TRUSTED` because the system ensures in these modes that only materialized views with updated data participate in a query rewrite.

Overview of Materialized View Management Tasks

The motivation for using materialized views is to improve performance, but the overhead associated with materialized view management can become a significant system management problem. When reviewing or evaluating some of the necessary materialized view management activities, consider some of the following:

- Identifying what materialized views to create initially.
- Indexing the materialized views.
- Ensuring that all materialized views and materialized view indexes are refreshed properly each time the database is updated.
- Checking which materialized views have been used.
- Determining how effective each materialized view has been on workload performance.
- Measuring the space being used by materialized views.
- Determining which new materialized views should be created.
- Determining which existing materialized views should be dropped.
- Archiving old detail and materialized view data that is no longer useful.

After the initial effort of creating and populating the data warehouse or data mart, the major administration overhead is the update process, which involves:

- Periodic extraction of incremental changes from the operational systems.
- Transforming the data.
- Verifying that the incremental changes are correct, consistent, and complete.
- Bulk-loading the data into the warehouse.
- Refreshing indexes and materialized views so that they are consistent with the detail data.

The update process must generally be performed within a limited period of time known as the **update window**. The update window depends on the **update**

frequency (such as daily or weekly) and the nature of the business. For a daily update frequency, an update window of two to six hours might be typical.

You need to know your update window for the following activities:

- Loading the detail data
- Updating or rebuilding the indexes on the detail data
- Performing quality assurance tests on the data
- Refreshing the materialized views
- Updating the indexes on the materialized views

Types of Materialized Views

The SELECT clause in the materialized view creation statement defines the data that the materialized view is to contain. Only a few restrictions limit what can be specified. Any number of tables can be joined together. However, they cannot be remote tables if you wish to take advantage of query rewrite. Besides tables, other elements such as views, inline views (subqueries in the FROM clause of a SELECT statement), subqueries, and materialized views can all be joined or referenced in the SELECT clause. You cannot, however, define a materialized with a subquery in the select list of the defining query. You can, however, include subqueries elsewhere in the defining query, such as in the WHERE clause.

The types of materialized views are:

- Materialized Views with Aggregates
- Materialized Views Containing Only Joins
- Nested Materialized Views

Materialized Views with Aggregates

In data warehouses, materialized views normally contain aggregates as shown in Example 8–1. For fast refresh to be possible, the SELECT list must contain all of the GROUP BY columns (if present), and there must be a COUNT(*) and a COUNT(column) on any aggregated columns. Also, materialized view logs must be present on all tables referenced in the query that defines the materialized view. The valid aggregate functions are: SUM, COUNT(x), COUNT(*), AVG, VARIANCE, STDDEV, MIN, and MAX, and the expression to be aggregated can be any SQL value expression. See "Restrictions on Fast Refresh on Materialized Views with Aggregates" on page 8-27.

Fast refresh for a materialized view containing joins and aggregates is possible after any type of DML to the base tables (direct load or conventional INSERT, UPDATE, or DELETE). It can be defined to be refreshed ON COMMIT or ON DEMAND. A REFRESH ON COMMIT materialized view will be refreshed automatically when a transaction that does DML to one of the materialized view's detail tables commits. The time taken to complete the commit may be slightly longer than usual when this method is chosen. This is because the refresh operation is performed as part of the commit process. Therefore, this method may not be suitable if many users are concurrently changing the tables upon which the materialized view is based.

Here are some examples of materialized views with aggregates. Note that materialized view logs are only created because this materialized view will be fast refreshed.

Example 8–1 Example 1: Creating a Materialized View

```
CREATE MATERIALIZED VIEW LOG ON products WITH SEQUENCE, ROWID
(prod_id, prod_name, prod_desc, prod_subcategory, prod_subcategory_desc,
prod_category, prod_category_desc, prod_weight_class, prod_unit_of_measure,
 prod_pack_size, supplier_id, prod_status, prod_list_price, prod_min_price)
INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW LOG ON sales
WITH SEQUENCE, ROWID
(prod_id, cust_id, time_id, channel_id, promo_id, quantity_sold, amount_sold)
INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW product_sales_mv
PCTFREE 0  TABLESPACE demo
STORAGE (INITIAL 8k NEXT 8k PCTINCREASE 0)
BUILD IMMEDIATE
REFRESH FAST
ENABLE QUERY REWRITE
AS SELECT p.prod_name, SUM(s.amount_sold) AS dollar_sales,
COUNT(*) AS cnt, COUNT(s.amount_sold) AS cnt_amt
FROM sales s, products p
WHERE s.prod_id = p.prod_id GROUP BY p.prod_name;
```

This example creates a materialized view product_sales_mv that computes total number and value of sales for a product. It is derived by joining the tables sales and products on the column prod_id. The materialized view is populated with data immediately because the build method is immediate and it is available for use by query rewrite. In this example, the default refresh method is FAST, which is

allowed because the appropriate materialized view logs have been created on tables product and sales.

Example 8–2 Example 2: Creating a Materialized View

```
CREATE MATERIALIZED VIEW product_sales_mv
PCTFREE 0 TABLESPACE demo
STORAGE (INITIAL 16k NEXT 16k PCTINCREASE 0)
BUILD DEFERRED
REFRESH COMPLETE ON DEMAND
ENABLE QUERY REWRITE AS
SELECT p.prod_name, SUM(s.amount_sold) AS dollar_sales
FROM sales s, products p WHERE s.prod_id = p.prod_id
GROUP BY p.prod_name;
```

This example creates a materialized view product_sales_mv that computes the sum of sales by prod_name. It is derived by joining the tables sales and products on the column prod_id. The materialized view does not initially contain any data, because the build method is DEFERRED. A complete refresh is required for the first refresh of a build deferred materialized view. When it is refreshed and once populated, this materialized view can be used by query rewrite.

Example 8–3 Example 3: Creating a Materialized View

```
CREATE MATERIALIZED VIEW LOG ON sales WITH SEQUENCE, ROWID
(prod_id, cust_id, time_id, channel_id, promo_id, quantity_sold, amount_sold)
INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW sum_sales
PARALLEL
BUILD IMMEDIATE
REFRESH FAST ON COMMIT AS
SELECT s.prod_id, s.time_id, COUNT(*) AS count_grp,
   SUM(s.amount_sold) AS sum_dollar_sales,
   COUNT(s.amount_sold) AS count_dollar_sales,
   SUM(s.quantity_sold) AS sum_quantity_sales,
   COUNT(s.quantity_sold) AS count_quantity_sales
FROM sales s
GROUP BY s.prod_id, s.time_id;
```

This example creates a materialized view that contains aggregates on a single table. Because the materialized view log has been created with all referenced columns in the materialized view's defining query, the materialized view is fast refreshable. If

DML is applied against the sales table, then the changes will be reflected in the materialized view when the commit is issued.

Requirements for Using Materialized Views with Aggregates

Table 8–2 illustrates the aggregate requirements for materialized views.

Table 8–2 Requirements for Materialized Views with Aggregates

If aggregate X is present, aggregate Y is required and aggregate Z is optional		
X	**Y**	**Z**
COUNT(expr)	-	-
SUM(expr)	COUNT(expr)	-
AVG(expr)	COUNT(expr)	SUM(expr)
STDDEV(expr)	COUNT(expr) SUM(expr)	SUM(expr * expr)
VARIANCE(expr)	COUNT(expr) SUM(expr)	SUM(expr * expr)

Note that COUNT(*) must always be present to guarantee all types of fast refresh. Otherwise, you may be limited to fast refresh after inserts only. Oracle recommends that you include the optional aggregates in column Z in the materialized view in order to obtain the most efficient and accurate fast refresh of the aggregates.

Materialized Views Containing Only Joins

Some materialized views contain only joins and no aggregates, such as in Example 8–4 on page 8-16, where a materialized view is created that joins the sales table to the times and customers tables. The advantage of creating this type of materialized view is that expensive joins will be precalculated.

Fast refresh for a materialized view containing only joins is possible after any type of DML to the base tables (direct-path or conventional INSERT, UPDATE, or DELETE).

A materialized view containing only joins can be defined to be refreshed ON COMMIT or ON DEMAND. If it is ON COMMIT, the refresh is performed at commit time of the transaction that does DML on the materialized view's detail table.

If you specify REFRESH FAST, Oracle performs further verification of the query definition to ensure that fast refresh can be performed if any of the detail tables change. These additional checks are:

- A materialized view log must be present for each detail table and the ROWID column must be present in each materialized view log.
- The rowids of all the detail tables must appear in the SELECT list of the materialized view query definition.
- If there are no outer joins, you may have arbitrary selections and joins in the WHERE clause. However, if there are outer joins, the WHERE clause cannot have any selections. Further, if there are outer joins, all the joins must be connected by ANDs and must use the equality (=) operator.
- If there are outer joins, unique constraints must exist on the join columns of the inner table. For example, if you are joining the fact table and a dimension table and the join is an outer join with the fact table being the outer table, there must exist unique constraints on the join columns of the dimension table.

If some of these restrictions are not met, you can create the materialized view as REFRESH FORCE to take advantage of fast refresh when it is possible. If one of the tables did not meet all of the criteria, but the other tables did, the materialized view would still be fast refreshable with respect to the other tables for which all the criteria are met.

Materialized Join Views FROM Clause Considerations

If the materialized view contains only joins, the ROWID columns for each table (and each instance of a table that occurs multiple times in the FROM list) must be present in the SELECT list of the materialized view.

If the materialized view has remote tables in the FROM clause, all tables in the FROM clause must be located on that same site. Further, ON COMMIT refresh is not supported for materialized view with remote tables. Materialized view logs must be present on the remote site for each detail table of the materialized view and ROWID columns must be present in the SELECT list of the materialized view.

To improve refresh performance, you should create indexes on the materialized view's columns that store the rowids of the fact table.

Example 8–4 Materialized View Containing Only Joins

```
CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID;
CREATE MATERIALIZED VIEW LOG ON times WITH ROWID;
CREATE MATERIALIZED VIEW LOG ON customers WITH ROWID;
```

```
CREATE MATERIALIZED VIEW detail_sales_mv
PARALLEL BUILD IMMEDIATE
REFRESH FAST AS
SELECT s.rowid "sales_rid", t.rowid "times_rid", c.rowid "customers_rid",
       c.cust_id, c.cust_last_name, s.amount_sold, s.quantity_sold, s.time_id
FROM sales s, times t, customers c
WHERE  s.cust_id = c.cust_id(+) AND s.time_id = t.time_id(+);
```

In this example, to perform a fast refresh, UNIQUE constraints should exist on `c.cust_id` and `t.time_id`. You should also create indexes on the columns `sales_rid`, `times_rid`, and `customers_rid`, as illustrated in the following. This will improve the refresh performance.

```
CREATE INDEX mv_ix_salesrid ON detail_sales_mv("sales_rid");
```

Alternatively, if the previous example did not include the columns `times_rid` and `customers_rid`, and if the refresh method was REFRESH FORCE, then this materialized view would be fast refreshable only if the sales table was updated but not if the tables `times` or `customers` were updated.

```
CREATE MATERIALIZED VIEW detail_sales_mv
PARALLEL
BUILD IMMEDIATE
REFRESH FORCE AS
SELECT s.rowid "sales_rid", c.cust_id, c.cust_last_name, s.amount_sold,
   s.quantity_sold, s.time_id
FROM sales s, times t, customers c
WHERE s.cust_id = c.cust_id(+) AND s.time_id = t.time_id(+);
```

Nested Materialized Views

A nested materialized view is a materialized view whose definition is based on another materialized view. A nested materialized view can reference other relations in the database in addition to referencing materialized views.

Why Use Nested Materialized Views?

In a data warehouse, you typically create many aggregate views on a single join (for example, rollups along different dimensions). Incrementally maintaining these distinct materialized aggregate views can take a long time, because the underlying join has to be performed many times.

Using nested materialized views, you can create multiple single-table materialized views based on a joins-only materialized view and the join is performed just once.

In addition, optimizations can be performed for this class of single-table aggregate materialized view and thus refresh is very efficient.

Example 8–5 Nested Materialized View

You can create a nested materialized view on materialized views that contain joins only or joins and aggregates. All the underlying objects (materialized views or tables) on which the materialized view is defined must have a materialized view log. All the underlying objects are treated as if they were tables. In addition, you can use all the existing options for materialized views.

Using the tables and their columns from the sh sample schema, the following materialized views illustrate how nested materialized views can be created.

```
CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID;
CREATE MATERIALIZED VIEW LOG ON customers WITH ROWID;
CREATE MATERIALIZED VIEW LOG ON times WITH ROWID;

/*create materialized view join_sales_cust_time as fast refreshable at
   COMMIT time */
CREATE MATERIALIZED VIEW join_sales_cust_time
REFRESH FAST ON COMMIT AS
SELECT c.cust_id, c.cust_last_name, s.amount_sold, t.time_id,
       t.day_number_in_week, s.rowid srid, t.rowid trid, c.rowid crid
FROM sales s, customers c, times t
WHERE s.time_id = t.time_id AND s.cust_id = c.cust_id;
```

To create a nested materialized view on the table join_sales_cust_time, you would have to create a materialized view log on the table. Because this will be a single-table aggregate materialized view on join_sales_cust_time, you need to log all the necessary columns and use the INCLUDING NEW VALUES clause.

```
/* create materialized view log on join_sales_cust_time */
CREATE MATERIALIZED VIEW LOG ON join_sales_cust_time
WITH ROWID (cust_last_name, day_number_in_week, amount_sold)
INCLUDING NEW VALUES;

/* create the single-table aggregate materialized view sum_sales_cust_time on
   join_sales_cust_time as fast refreshable at COMMIT time */
CREATE MATERIALIZED VIEW sum_sales_cust_time
REFRESH FAST ON COMMIT AS
SELECT COUNT(*) cnt_all, SUM(amount_sold) sum_sales, COUNT(amount_sold)
       cnt_sales, cust_last_name, day_number_in_week
FROM join_sales_cust_time
GROUP BY cust_last_name, day_number_in_week;
```

Nesting Materialized Views with Joins and Aggregates

Some types of nested materialized views cannot be fast refreshed. Use EXPLAIN_MVIEW to identify those types of materialized views. You can refresh a tree of nested materialized views in the appropriate dependency order by specifying the nested = TRUE parameter with the DBMS_MVIEW. REFRESH parameter. For example, if you call DBMS_MVIEW.REFRESH ('SUM_SALES_CUST_TIME', nested => TRUE), the REFRESH procedure will first refresh the join_sales_cust_time materialized view, and then refresh the sum_sales_cust_time materialized view.

Nested Materialized View Usage Guidelines

You should keep the following in mind when deciding whether to use nested materialized views:

- If you want to use fast refresh, you should fast refresh all the materialized views along any chain.
- If you want the highest level materialized view to be fresh with respect to the detail tables, you need to ensure that all materialized views in a tree are refreshed in the correct dependency order before refreshing the highest-level. You can automatically refesh intermediate materialized views in a nested hierarchy using the nested = TRUE parameter, as described in "Nesting Materialized Views with Joins and Aggregates" on page 8-19. If you do not specify nested = TRUE and the materialzied views under the highest-level materialized view are stale, refreshing only the highest-level will succeed, but makes it fresh only with respect to its underlying materialized view, not the detail tables at the base of the tree.
- When refreshing materialized views, you need to ensure that all materialized views in a tree are refreshed. If you only refresh the highest-level materialized view, the materialized views under it will be stale and you must explicitly refresh them. If you use the REFRESH procedure with the nested parameter value set to TRUE, only specified materialized views and their child materialized views in the tree are refreshed, and not their top-level materialized views. Use the REFRESH_DEPENDENT procedure with the nested parameter value set to TRUE if you want to ensure that all materialized views in a tree are refreshed.
- Freshness of a materialized view is calculated relative to the objects directly referenced by the materialized view. When a materialized view references another materialized view, the freshness of the topmost materialized view is calculated relative to changes in the materialized view it directly references, not

relative to changes in the tables referenced by the materialized view it references.

Restrictions When Using Nested Materialized Views

You cannot create both a materialized view and a prebuilt materialized view on the same table. For example, If you have a table `costs` with a materialized view `cost_mv` based on it, you cannot then create a prebuilt materialized view on table `costs`. The result would make `cost_mv` a nested materialized view and this method of conversion is not supported.

Creating Materialized Views

A materialized view can be created with the `CREATE MATERIALIZED VIEW` statement or using Enterprise Manager. Example 8–6 creates the materialized view `cust_sales_mv`.

Example 8–6 Creating a Materialized View

```
CREATE MATERIALIZED VIEW cust_sales_mv
PCTFREE 0 TABLESPACE demo
STORAGE (INITIAL 16k NEXT 16k PCTINCREASE 0)
PARALLEL
BUILD IMMEDIATE
REFRESH COMPLETE
ENABLE QUERY REWRITE AS
SELECT  c.cust_last_name, SUM(amount_sold) AS sum_amount_sold
FROM customers c, sales s WHERE s.cust_id = c.cust_id
GROUP BY c.cust_last_name;
```

It is not uncommon in a data warehouse to have already created summary or aggregation tables, and you might not wish to repeat this work by building a new materialized view. In this case, the table that already exists in the database can be registered as a prebuilt materialized view. This technique is described in "Registering Existing Materialized Views" on page 8-34.

Once you have selected the materialized views you want to create, follow these steps for each materialized view.

1. Design the materialized view. Existing user-defined materialized views do not require this step. If the materialized view contains many rows, then, if appropriate, the materialized view should be partitioned (if possible) and should match the partitioning of the largest or most frequently updated detail or fact table (if possible). Refresh performance benefits from partitioning,

because it can take advantage of parallel DML capabilities and possible PCT-based refresh.

2. Use the CREATE MATERIALIZED VIEW statement to create and, optionally, populate the materialized view. If a user-defined materialized view already exists, then use the ON PREBUILT TABLE clause in the CREATE MATERIALIZED VIEW statement. Otherwise, use the BUILD IMMEDIATE clause to populate the materialized view immediately, or the BUILD DEFERRED clause to populate the materialized view later. A BUILD DEFERRED materialized view is disabled for use by query rewrite until the first COMPLETE REFRESH, after which it will be automatically enabled, provided the ENABLE QUERY REWRITE clause has been specified.

   See Also: *Oracle Database SQL Reference* for descriptions of the SQL statements CREATE MATERIALIZED VIEW, ALTER MATERIALIZED VIEW, and DROP MATERIALIZED VIEW

Creating Materialized Views with Column Alias Lists

Currently, when a materialized view is created, if its defining query contains same-name columns in the SELECT list, the name conflicts need to be resolved by specifying unique aliases for those columns. Otherwise, the CREATE MATERIALIZED VIEW statement will fail with the error messages of columns ambiguously defined. However, the standard method of attaching aliases in the SELECT clause for name resolution restricts the use of the full text match query rewrite and it will occur only when the text of the materialized view's defining query and the text of user input query are identical. Thus, if the user specifies select aliases in the materialized view's defining query while there is no alias in the query, the full text match comparison will fail. This is particularly a problem for queries from Discoverer, which makes extensive use of column aliases.

The following is an example of the problem. sales_mv is created with column aliases in the SELECT clause but the input query Q1 does not have the aliases. The full text match rewrite will fail. The materialized view is as follows:

```
CREATE MATERIALIZED VIEW sales_mv
ENABLE QUERY REWRITE AS
SELECT s.time_id sales_tid, c.time_id costs_tid
FROM sales s, products p, costs c
WHERE s.prod_id = p.prod_id AND c.prod_id = p.prod_id AND
      p.prod_name IN (SELECT prod_name FROM products);
```

Input query statement Q1 is as follows:

```
SELECT s.time_id, c.time_id
FROM sales s, products p, costs c
WHERE s.prod_id = p.prod_id AND c.prod_id = p.prod_id AND
      p.prod_name IN (SELECT prod_name FROM products);
```

Even though the materialized view's defining query is almost identical and logically equivalent to the user's input query, query rewrite does not happen because of the failure of full text match that is the only rewrite possibility for some queries (for example, a subquery in the `WHERE` clause).

You can add a column alias list to a `CREATE MATERIALIZED VIEW` statement. The column alias list explicitly resolves any column name conflict without attaching aliases in the `SELECT` clause of the materialized view. The syntax of the materialized view column alias list is illustrated in the following example:

```
CREATE MATERIALIZED VIEW sales_mv (sales_tid, costs_tid)
ENABLE QUERY REWRITE AS
SELECT s.time_id, c.time_id
FROM sales s, products p, costs c
WHERE s.prod_id = p.prod_id AND c.prod_id = p.prod_id AND
      p.prod_name IN (SELECT prod_name FROM products);
```

In this example, the defining query of `sales_mv` now matches exactly with the user query `Q1`, so full text match rewrite will take place.

Note that when aliases are specified in both the `SELECT` clause and the new alias list clause, the alias list clause supersedes the ones in the `SELECT` clause.

Naming Materialized Views

The name of a materialized view must conform to standard Oracle naming conventions. However, if the materialized view is based on a user-defined prebuilt table, then the name of the materialized view must exactly match that table name.

If you already have a naming convention for tables and indexes, you might consider extending this naming scheme to the materialized views so that they are easily identifiable. For example, instead of naming the materialized view `sum_of_sales`, it could be called `sum_of_sales_mv` to denote that this is a materialized view and not a table or view.

Storage And Table Compression

Unless the materialized view is based on a user-defined prebuilt table, it requires and occupies storage space inside the database. Therefore, the storage needs for the

materialized view should be specified in terms of the tablespace where it is to reside and the size of the extents.

If you do not know how much space the materialized view will require, then the DBMS_MVIEW.ESTIMATE_SIZE package can estimate the number of bytes required to store this uncompressed materialized view. This information can then assist the design team in determining the tablespace in which the materialized view should reside.

You should use table compression with highly redundant data, such as tables with many foreign keys. This is particularly useful for materialized views created with the ROLLUP clause. Table compression reduces disk use and memory use (specifically, the buffer cache), often leading to a better scaleup for read-only operations. Table compression can also speed up query execution at the expense of update cost.

> **See Also:** *Oracle Database SQL Reference* for a complete description of STORAGE semantics, *Oracle Database Performance Tuning Guide*, and Chapter 5, "Parallelism and Partitioning in Data Warehouses" for table compression examples

Build Methods

Two build methods are available for creating the materialized view, as shown in Table 8–3. If you select BUILD IMMEDIATE, the materialized view definition is added to the schema objects in the data dictionary, and then the fact or detail tables are scanned according to the SELECT expression and the results are stored in the materialized view. Depending on the size of the tables to be scanned, this build process can take a considerable amount of time.

An alternative approach is to use the BUILD DEFERRED clause, which creates the materialized view without data, thereby enabling it to be populated at a later date using the DBMS_MVIEW.REFRESH package described in Chapter 15, "Maintaining the Data Warehouse".

Table 8–3 Build Methods

Build Method	Description
BUILD IMMEDIATE	Create the materialized view and then populate it with data.
BUILD DEFERRED	Create the materialized view definition but do not populate it with data.

Enabling Query Rewrite

Before creating a materialized view, you can verify what types of query rewrite are possible by calling the procedure DBMS_MVIEW.EXPLAIN_MVIEW, or use DBMS_ADVISOR.TUNE_MVIEW to optimize the materialized view so that a many types of query rewrite are possible. Once the materialized view has been created, you can use DBMS_MVIEW.EXPLAIN_REWRITE to find out if (or why not) it will rewrite a specific query.

Even though a materialized view is defined, it will not automatically be used by the query rewrite facility. Even though query rewrite is enabled by default, you also must specify the ENABLE QUERY REWRITE clause if the materialized view is to be considered available for rewriting queries.

If this clause is omitted or specified as DISABLE QUERY REWRITE when the materialized view is created, the materialized view can subsequently be enabled for query rewrite with the ALTER MATERIALIZED VIEW statement.

If you define a materialized view as BUILD DEFERRED, it is not eligible for query rewrite until it is populated with data.

Query Rewrite Restrictions

Query rewrite is not possible with all materialized views. If query rewrite is not occurring when expected, DBMS_MVIEW.EXPLAIN_REWRITE can help provide reasons why a specific query is not eligible for rewrite. If this shows that not all types of query rewrite are possible, use the procedure DBMS_ADVISOR.TUNE_MVIEW to see if the materialized view can be defined differently so that query rewrite is possible. Also, check to see if your materialized view satisfies all of the following conditions.

Materialized View Restrictions

You should keep in mind the following restrictions:

- The defining query of the materialized view cannot contain any non-repeatable expressions (ROWNUM, SYSDATE, non-repeatable PL/SQL functions, and so on).
- The query cannot contain any references to RAW or LONG RAW datatypes or object REFs.
- If the materialized view was registered as PREBUILT, the precision of the columns must agree with the precision of the corresponding SELECT expressions unless overridden by the WITH REDUCED PRECISION clause.

General Query Rewrite Restrictions

You should keep in mind the following restrictions:

- If a query has both local and remote tables, only local tables will be considered for potential rewrite.
- Neither the detail tables nor the materialized view can be owned by SYS.
- If a column or expression is present in the GROUP BY clause of the materialized view, it must also be present in the SELECT list.
- Aggregate functions must occur only as the outermost part of the expression. That is, aggregates such as AVG(AVG(x)) or AVG(x)+ AVG(x) are not allowed.
- CONNECT BY clauses are not allowed.

Refresh Options

When you define a materialized view, you can specify three refresh options: how to refresh, what type of refresh, and can trusted constraints be used. If unspecified, the defaults are assumed as ON DEMAND, FORCE, and ENFORCED constraints respectively.

The two refresh execution modes are ON COMMIT and ON DEMAND. Depending on the materialized view you create, some of the options may not be available. Table 8–4 describes the refresh modes.

Table 8–4 Refresh Modes

Refresh Mode	Description
ON COMMIT	Refresh occurs automatically when a transaction that modified one of the materialized view's detail tables commits. This can be specified as long as the materialized view is fast refreshable (in other words, not complex). The ON COMMIT privilege is necessary to use this mode
ON DEMAND	Refresh occurs when a user manually executes one of the available refresh procedures contained in the DBMS_MVIEW package (REFRESH, REFRESH_ALL_MVIEWS, REFRESH_DEPENDENT)

When a materialized view is maintained using the ON COMMIT method, the time required to complete the commit may be slightly longer than usual. This is because the refresh operation is performed as part of the commit process. Therefore this method may not be suitable if many users are concurrently changing the tables upon which the materialized view is based.

If you anticipate performing insert, update or delete operations on tables referenced by a materialized view concurrently with the refresh of that materialized view, and

that materialized view includes joins and aggregation, Oracle recommends you use ON COMMIT fast refresh rather than ON DEMAND fast refresh.

If you think the materialized view did not refresh, check the alert log or trace file.

If a materialized view fails during refresh at COMMIT time, you must explicitly invoke the refresh procedure using the DBMS_MVIEW package after addressing the errors specified in the trace files. Until this is done, the materialized view will no longer be refreshed automatically at commit time.

You can specify how you want your materialized views to be refreshed from the detail tables by selecting one of four options: COMPLETE, FAST, FORCE, and NEVER. Table 8–5 describes the refresh options.

Table 8–5 Refresh Options

Refresh Option	Description
COMPLETE	Refreshes by recalculating the materialized view's defining query.
FAST	Applies incremental changes to refresh the materialized view using the information logged in the materialized view logs, or from a SQL*Loader direct-path or a partition maintenance operation.
FORCE	Applies FAST refresh if possible; otherwise, it applies COMPLETE refresh.
NEVER	Indicates that the materialized view will not be refreshed with refresh mechanisms.

Whether the fast refresh option is available depends upon the type of materialized view. You can call the procedure DBMS_MVIEW.EXPLAIN_MVIEW to determine whether fast refresh is possible.

You can also specify if it is acceptable to use trusted constraints and REWRITE_INTEGRITY = TRUSTED during refresh. Any nonvalidated RELY constraint is a trusted constraint. For example, nonvalidated foreign key / primary key relationships, functional dependencies defined in dimensions or a materialized view in the UNKNOWN state. If query rewrite is enabled during refresh, these can improve the performance of refresh by enabling more performant query rewrites. Any materialized view that can uses TRUSTED constraints for refresh is left in a state of trusted freshness (the UNKNOWN state) after refresh.

This is reflected in the column STALENESS in the view USER_MVIEWS. The column UNKNOWN_TRUSTED_FD in the same view is also set to Y, which means yes.

You can define this property of the materialized either during create time by specifying REFRESH USING TRUSTED [ENFORCED] CONSTRAINTS or by using ALTER MATERIALIZED VIEW DDL.

Table 8–6 Constraints

Constraints to Use	Description
TRUSTED CONSTRAINTS	Refresh can use trusted constraints and REWRITE_INTEGRITY = TRUSTED during refresh. This allows use of non-validated RELY constraints and rewrite against materialized views in UNKNOWN or FRESH state during refresh.
ENFORCED CONSTRAINTS	Refresh can use validated constraints and REFRESH_INTEGRITY=ENFORCED during refresh. This allows use of only validated, enforced constraints and rewrite against materialized views in FRESH state during refresh.

General Restrictions on Fast Refresh

The defining query of the materialized view is restricted as follows:

- The materialized view must not contain references to non-repeating expressions like SYSDATE and ROWNUM.
- The materialized view must not contain references to RAW or LONG RAW data types.

Restrictions on Fast Refresh on Materialized Views with Joins Only

Defining queries for materialized views with joins only and no aggregates have the following restrictions on fast refresh:

- All restrictions from "General Restrictions on Fast Refresh" on page 8-27.
- They cannot have GROUP BY clauses or aggregates.
- If the WHERE clause of the query contains outer joins, then unique constraints must exist on the join columns of the inner join table.
- If there are no outer joins, you can have arbitrary selections and joins in the WHERE clause. However, if there are outer joins, the WHERE clause cannot have any selections. Furthermore, if there are outer joins, all the joins must be connected by ANDs and must use the equality (=) operator.
- Rowids of all the tables in the FROM list must appear in the SELECT list of the query.
- Materialized view logs must exist with rowids for all the base tables in the FROM list of the query.

Restrictions on Fast Refresh on Materialized Views with Aggregates

Defining queries for materialized views with aggregates or joins have the following restrictions on fast refresh:

- All restrictions from "General Restrictions on Fast Refresh" on page 8-27.

Fast refresh is supported for both `ON COMMIT` and `ON DEMAND` materialized views, however the following restrictions apply:

- All tables in the materialized view must have materialized view logs, and the materialized view logs must:
  - Contain all columns from the table referenced in the materialized view.
  - Specify with `ROWID` and `INCLUDING NEW VALUES`.
  - Specify the `SEQUENCE` clause if the table is expected to have a mix of inserts/direct-loads, deletes, and updates.
- Only `SUM`, `COUNT`, `AVG`, `STDDEV`, `VARIANCE`, `MIN` and `MAX` are supported for fast refresh.
- `COUNT(*)` must be specified.
- For each aggregate such as `AVG(expr)`, the corresponding `COUNT(expr)` must be present.
- If `VARIANCE(expr)` or `STDDEV(expr)` is specified, `COUNT(expr)` and `SUM(expr)` must be specified. Oracle recommends that `SUM(expr *expr)` be specified. See Table 8–2 on page 8-15 for further details.
- The `SELECT` list must contain all `GROUP BY` columns.
- If the materialized view has one of the following, then fast refresh is supported only on conventional DML inserts and direct loads.
  - Materialized views with `MIN` or `MAX` aggregates
  - Materialized views which have `SUM(expr)` but no `COUNT(expr)`
  - Materialized views without `COUNT(*)`

  Such a materialized view is called an insert-only materialized view.
- A materialized view with `MAX` or `MIN` is fast refreshable after delete or mixed DML statements if it does not have a `WHERE` clause.
- Materialized views with named views or subqueries in the `FROM` clause can be fast refreshed provided the views can be completely merged. For information on which views will merge, refer to the *Oracle Database Performance Tuning Guide*.
- If there are no outer joins, you may have arbitrary selections and joins in the `WHERE` clause.

- Materialized aggregate views with outer joins are fast refreshable after conventional DML and direct loads, provided only the outer table has been modified. Also, unique constraints must exist on the join columns of the inner join table. If there are outer joins, all the joins must be connected by ANDs and must use the equality (=) operator.
- For materialized views with CUBE, ROLLUP, grouping sets, or concatenation of them, the following restrictions apply:
  - The SELECT list should contain grouping distinguisher that can either be a GROUPING_ID function on all GROUP BY expressions or GROUPING functions one for each GROUP BY expression. For example, if the GROUP BY clause of the materialized view is "GROUP BY CUBE(a, b)", then the SELECT list should contain either "GROUPING_ID(a, b)" or "GROUPING(a) AND GROUPING(b)" for the materialized view to be fast refreshable.
  - GROUP BY should not result in any duplicate groupings. For example, "GROUP BY a, ROLLUP(a, b)" is not fast refreshable because it results in duplicate groupings "(a), (a, b), AND (a)".

Restrictions on Fast Refresh on Materialized Views with UNION ALL

Materialized views with the UNION ALL set operator support the REFRESH FAST option if the following conditions are satisfied:

- The defining query must have the UNION ALL operator at the top level.

  The UNION ALL operator cannot be embedded inside a subquery, with one exception: The UNION ALL can be in a subquery in the FROM clause provided the defining query is of the form SELECT * FROM (view or subquery with UNION ALL) as in the following example:

```
CREATE VIEW view_with_unionall_mv AS
(SELECT c.rowid crid, c.cust_id, 2 umarker
 FROM customers c WHERE c.cust_last_name = 'Smith'
 UNION ALL
 SELECT c.rowid crid, c.cust_id, 3 umarker
 FROM customers c WHERE c.cust_last_name = 'Jones');

CREATE MATERIALIZED VIEW unionall_inside_view_mv
REFRESH FAST ON DEMAND AS
SELECT * FROM view_with_unionall;
```

Note that the view view_with_unionall_mv satisfies all requirements for fast refresh.

- Each query block in the UNION ALL query must satisfy the requirements of a fast refreshable materialized view with aggregates or a fast refreshable materialized view with joins.

  The appropriate materialized view logs must be created on the tables as required for the corresponding type of fast refreshable materialized view.

  Note that the Oracle Database also allows the special case of a single table materialized view with joins only provided the ROWID column has been included in the SELECT list and in the materialized view log. This is shown in the defining query of the view view_with_unionall_mv.

- The SELECT list of each query must include a maintenance column, called a UNION ALL marker. The UNION ALL column must have a distinct constant numeric or string value in each UNION ALL branch. Further, the marker column must appear in the same ordinal position in the SELECT list of each query block.

- Some features such as outer joins, insert-only aggregate materialized view queries and remote tables are not supported for materialized views with UNION ALL.

- PCT-based refresh is not supported for UNION ALL materialized views.

- The compatibility initialization parameter must be set to 9.2.0 or higher to create a fast refreshable materialized view with UNION ALL.

Achieving Refresh Goals

In addition to the EXPLAIN_MVIEW procedure, which is discussed throughout this chapter, you can use the DBMS_ADVISOR.TUNE_MVIEW procedure to optimize a CREATE MATERIALIZED VIEW statement to achieve REFRESH FAST and ENABLE QUERY REWRITE goals. The procedure is described in "Tuning Materialized Views for Fast Refresh and Query Rewrite" on page 17-47.

Refreshing Nested Materialized Views

A nested materialized view is considered to be fresh as long as its data is synchronized with the data in its detail tables, even if some of its detail tables could be stale materialized views.

You can refresh nested materialized views in two ways: DBMS_MVIEW.REFRESH with the nested flag set to TRUE and REFRESH_DEPENDENT with the nested flag

set to TRUE on the base tables. If you use DBMS_MVIEW.REFRESH, the entire materialized view chain is refreshed from the top down. With DBMS_MVIEW.REFRESH_DEPENDENT, the entire chain is refreshed from the bottom up.

Example 8–7 Example of Refreshing a Nested Materialized View

The following statement shows an example of refreshing a nested materialized view:

```
DBMS_MVIEW.REFRESH('SALES_MV,COST_MV', nested => TRUE);
```

This statement will first refresh all child materialized views of sales_mv and cost_mv based on the dependency analysis and then refresh the two specified materialized views.

You can query the STALE_SINCE column in the *_MVIEWS views to find out when a materialized view became stale.

ORDER BY Clause

An ORDER BY clause is allowed in the CREATE MATERIALIZED VIEW statement. It is used only during the initial creation of the materialized view. It is not used during a full refresh or a fast refresh.

To improve the performance of queries against large materialized views, store the rows in the materialized view in the order specified in the ORDER BY clause. This initial ordering provides physical clustering of the data. If indexes are built on the columns by which the materialized view is ordered, accessing the rows of the materialized view using the index often reduces the time for disk I/O due to the physical clustering.

The ORDER BY clause is not considered part of the materialized view definition. As a result, there is no difference in the manner in which Oracle Database detects the various types of materialized views (for example, materialized join views with no aggregates). For the same reason, query rewrite is not affected by the ORDER BY clause. This feature is similar to the CREATE TABLE ... ORDER BY capability.

Materialized View Logs

Materialized view logs are required if you want to use fast refresh, with the exception of PCT refresh,where there a few situations where they are not necessary. As a general rule, though, you should create materialized view logs if you want to use fast refresh. Materialized view logs are defined using a CREATE MATERIALIZED VIEW LOG statement on the base table that is to be changed. They

are not created on the materialized view. For fast refresh of materialized views, the definition of the materialized view logs must normally specify the ROWID clause. In addition, for aggregate materialized views, it must also contain every column in the table referenced in the materialized view, the INCLUDING NEW VALUES clause and the SEQUENCE clause.

An example of a materialized view log is shown as follows where one is created on the table sales.

```
CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID
(prod_id, cust_id, time_id, channel_id, promo_id, quantity_sold, amount_sold)
INCLUDING NEW VALUES;
```

Alternatively, a materialized view log can be amended to include the rowid, as in the following:

```
ALTER MATERIALIZED VIEW LOG ON sales ADD ROWID
(prod_id, cust_id, time_id, channel_id, promo_id, quantity_sold, amount_sold)
INCLUDING NEW VALUES;
```

Oracle recommends that the keyword SEQUENCE be included in your materialized view log statement unless you are sure that you will never perform a mixed DML operation (a combination of INSERT, UPDATE, or DELETE operations on multiple tables). The SEQUENCE column is required in the materialized view log to support fast refresh with a combination of INSERT, UPDATE, or DELETE statements on multiple tables. You can, however, add the SEQUENCE number to the materialized view log after it has been created.

The boundary of a mixed DML operation is determined by whether the materialized view is ON COMMIT or ON DEMAND.

- For ON COMMIT, the mixed DML statements occur within the same transaction because the refresh of the materialized view will occur upon commit of this transaction.
- For ON DEMAND, the mixed DML statements occur between refreshes. The following example of a materialized view log illustrates where one is created on the table sales that includes the SEQUENCE keyword:

  ```
  CREATE MATERIALIZED VIEW LOG ON sales WITH SEQUENCE, ROWID
  (prod_id, cust_id, time_id, channel_id, promo_id,
   quantity_sold, amount_sold) INCLUDING NEW VALUES;
  ```

Using the FORCE Option with Materialized View Logs

If you specify FORCE and any items specified with the ADD clause have already been specified for the materialized view log, Oracle does not return an error, but silently ignores the existing elements and adds to the materialized view log any items that do not already exist in the log. For example, if you used a filter column such as `cust_id` and this column already existed, Oracle Database ignores the redundancy and does not return an error.

Using Oracle Enterprise Manager

A materialized view can also be created using Enterprise Manager by selecting the materialized view object type. There is no difference in the information required if this approach is used. See *Oracle Enterprise Manager Advanced Configuration* for further information.

Using Materialized Views with NLS Parameters

When using certain materialized views, you must ensure that your NLS parameters are the same as when you created the materialized view. Materialized views with this restriction are as follows:

- Expressions that may return different values, depending on NLS parameter settings. For example, (date > "01/02/03") or (rate <= "2.150") are NLS parameter dependent expressions.
- Equijoins where one side of the join is character data. The result of this equijoin depends on collation and this can change on a session basis, giving an incorrect result in the case of query rewrite or an inconsistent materialized view after a refresh operation.
- Expressions that generate internal conversion to character data in the SELECT list of a materialized view, or inside an aggregate of a materialized aggregate view. This restriction does not apply to expressions that involve only numeric data, for example, a+b where a and b are numeric fields.

Adding Comments to Materialized Views

You can add a comment to a materialized view. For example, the following statement adds a comment to data dictionary views for the existing materialized view:

```
COMMENT ON MATERIALIZED VIEW sales_mv IS 'sales materialized view';
```

To view the comment after the preceding statement execution, the user can query the catalog views, {USER, DBA} ALL_MVIEW_COMMENTS. For example:

```
SELECT MVIEW_NAME, COMMENTS
FROM USER_MVIEW_COMMENTS WHERE MVIEW_NAME = 'SALES_MV';
```

The output will resemble the following:

```
MVIEW_NAME      COMMENTS
-----------       ----------------------
SALES_MV         sales materialized view
```

Note: If the compatibility is set to 10.0.1 or higher, COMMENT ON TABLE will not be allowed for the materialized view container table. The following error message will be thrown if it is issued.

```
ORA-12098: cannot comment on the materialized view.
```

In the case of a prebuilt table, if it has an existing comment, the comment will be inherited by the materialized view after it has been created. The existing comment will be prefixed with '(from table)'. For example, table sales_summary was created to contain sales summary information. An existing comment 'Sales summary data' was associated with the table. A materialized view of the same name is created to use the prebuilt table as its container table. After the materialized view creation, the comment becomes '(from table) Sales summary data'.

However, if the prebuilt table, sales_summary, does not have any comment, the following comment is added: 'Sales summary data'. Then, if we drop the materialized view, the comment will be passed to the prebuilt table with the comment: '(from materialized view) Sales summary data'.

Registering Existing Materialized Views

Some data warehouses have implemented materialized views in ordinary user tables. Although this solution provides the performance benefits of materialized views, it does not:

- Provide query rewrite to all SQL applications.
- Enable materialized views defined in one application to be transparently accessed in another application.
- Generally support fast parallel or fast materialized view refresh.

Because of these limitations, and because existing materialized views can be extremely large and expensive to rebuild, you should register your existing

materialized view tables whenever possible. You can register a user-defined materialized view with the CREATE MATERIALIZED VIEW ... ON PREBUILT TABLE statement. Once registered, the materialized view can be used for query rewrites or maintained by one of the refresh methods, or both.

The contents of the table must reflect the materialization of the defining query at the time you register it as a materialized view, and each column in the defining query must correspond to a column in the table that has a matching datatype. However, you can specify WITH REDUCED PRECISION to allow the precision of columns in the defining query to be different from that of the table columns.

The table and the materialized view must have the same name, but the table retains its identity as a table and can contain columns that are not referenced in the defining query of the materialized view. These extra columns are known as unmanaged columns. If rows are inserted during a refresh operation, each unmanaged column of the row is set to its default value. Therefore, the unmanaged columns cannot have NOT NULL constraints unless they also have default values.

Materialized views based on prebuilt tables are eligible for selection by query rewrite provided the parameter QUERY_REWRITE_INTEGRITY is set to STALE_TOLERATED or TRUSTED. See Chapter 18, "Query Rewrite" for details about integrity levels.

When you drop a materialized view that was created on a prebuilt table, the table still exists—only the materialized view is dropped.

The following example illustrates the two steps required to register a user-defined table. First, the table is created, then the materialized view is defined using exactly the same name as the table. This materialized view sum_sales_tab is eligible for use in query rewrite.

```
CREATE TABLE sum_sales_tab
PCTFREE 0  TABLESPACE demo
STORAGE (INITIAL 16k NEXT 16k PCTINCREASE 0) AS
SELECT s.prod_id, SUM(amount_sold) AS dollar_sales,
       SUM(quantity_sold) AS unit_sales
FROM sales s GROUP BY s.prod_id;

CREATE MATERIALIZED VIEW sum_sales_tab
ON PREBUILT TABLE WITHOUT REDUCED PRECISION
ENABLE QUERY REWRITE AS
SELECT s.prod_id, SUM(amount_sold) AS dollar_sales,
    SUM(quantity_sold) AS unit_sales
FROM sales s GROUP BY s.prod_id;
```

You could have compressed this table to save space. See "Storage And Table Compression" on page 8-22 for details regarding table compression.

In some cases, user-defined materialized views are refreshed on a schedule that is longer than the update cycle. For example, a monthly materialized view might be updated only at the end of each month, and the materialized view values always refer to complete time periods. Reports written directly against these materialized views implicitly select only data that is not in the current (incomplete) time period. If a user-defined materialized view already contains a time dimension:

- It should be registered and then fast refreshed each update cycle.
- You can create a view that selects the complete time period of interest.
- The reports should be modified to refer to the view instead of referring directly to the user-defined materialized view.

If the user-defined materialized view does not contain a time dimension, then:

- Create a new materialized view that does include the time dimension (if possible).
- The view should aggregate over the time column in the new materialized view.

Choosing Indexes for Materialized Views

The two most common operations on a materialized view are query execution and fast refresh, and each operation has different performance requirements. Query execution might need to access any subset of the materialized view key columns, and might need to join and aggregate over a subset of those columns. Consequently, query execution usually performs best if a single-column bitmap index is defined on each materialized view key column.

In the case of materialized views containing only joins using fast refresh, Oracle recommends that indexes be created on the columns that contain the rowids to improve the performance of the refresh operation.

If a materialized view using aggregates is fast refreshable, then an index appropriate for the fast refresh procedure is created unless `USING NO INDEX` is specified in the `CREATE MATERIALIZED VIEW` statement.

See Chapter 17, "SQLAccess Advisor" for information on using the SQLAccess Advisor to determine what indexes are appropriate for your materialized view.

Dropping Materialized Views

Use the DROP MATERIALIZED VIEW statement to drop a materialized view. For example. the following statement:

```
DROP MATERIALIZED VIEW sales_sum_mv;
```

This statement drops the materialized view sales_sum_mv. If the materialized view was prebuilt on a table, then the table is not dropped, but it can no longer be maintained with the refresh mechanism or used by query rewrite. Alternatively, you can drop a materialized view using Oracle Enterprise Manager.

Analyzing Materialized View Capabilities

You can use the DBMS_MVIEW.EXPLAIN_MVIEW procedure to learn what is possible with a materialized view or potential materialized view. In particular, this procedure enables you to determine:

- If a materialized view is fast refreshable
- What types of query rewrite you can perform with this materialized view
- Whether PCT refresh is possible

Using this procedure is straightforward. You simply call DBMS_MVIEW.EXPLAIN_MVIEW, passing in as a single parameter the schema and materialized view name for an existing materialized view. Alternatively, you can specify the SELECT string for a potential materialized view or the complete CREATE MATERIALIZED VIEW statement. The materialized view or potential materialized view is then analyzed and the results are written into either a table called MV_CAPABILITIES_TABLE, which is the default, or to an array called MSG_ARRAY.

Note that you must run the utlxmv.sql script prior to calling EXPLAIN_MVIEW except when you are placing the results in MSG_ARRAY. The script is found in the admin directory. It is to create the MV_CAPABILITIES_TABLE in the current schema. An explanation of the various capabilities is in Table 8–7 on page 8-41, and all the possible messages are listed in Table 8–8 on page 8-43.

Using the DBMS_MVIEW.EXPLAIN_MVIEW Procedure

The EXPLAIN_MVIEW procedure has the following parameters:

- stmt_id

An optional parameter. A client-supplied unique identifier to associate output rows with specific invocations of EXPLAIN_MVIEW.

- mv

  The name of an existing materialized view or the query definition or the entire CREATE MATERIALIZED VIEW statement of a potential materialized view you want to analyze.

- msg-array

  The PL/SQL varray that receives the output.

EXPLAIN_MVIEW analyzes the specified materialized view in terms of its refresh and rewrite capabilities and inserts its results (in the form of multiple rows) into MV_CAPABILITIES_TABLE or MSG_ARRAY.

> **See Also:** *PL/SQL Packages and Types Reference* for further information about the DBMS_MVIEW package

DBMS_MVIEW.EXPLAIN_MVIEW Declarations

The following PL/SQL declarations that are made for you in the DBMS_MVIEW package show the order and datatypes of these parameters for explaining an existing materialized view and a potential materialized view with output to a table and to a VARRAY.

Explain an existing or potential materialized view with output to MV_CAPABILITIES_TABLE:

```
DBMS_MVIEW.EXPLAIN_MVIEW (mv           IN VARCHAR2,
                          stmt_id IN VARCHAR2:= NULL);
```

Explain an existing or potential materialized view with output to a VARRAY:

```
DBMS_MVIEW.EXPLAIN_MVIEW (mv          IN VARCHAR2,
                          msg_array   OUT SYS.ExplainMVArrayType);
```

Using MV_CAPABILITIES_TABLE

One of the simplest ways to use DBMS_MVIEW.EXPLAIN_MVIEW is with the MV_CAPABILITIES_TABLE, which has the following structure:

```
CREATE TABLE MV_CAPABILITIES_TABLE
(STMT_ID           VARCHAR(30),    -- client-supplied unique statement identifier
 MV                VARCHAR(30),    -- NULL for SELECT based EXPLAIN_MVIEW
 CAPABILITY_NAME   VARCHAR(30),    -- A descriptive name of particular
                                   -- capabilities, such as REWRITE.
```

```
                                -- See Table 8-7
 POSSIBLE          CHARACTER(1),  -- Y = capability is possible
                                -- N = capability is not possible
 RELATED_TEXT      VARCHAR(2000), -- owner.table.column, and so on related to
                                -- this message
 RELATED_NUM       NUMBER,        -- When there is a numeric value
                                -- associated with a row, it goes here.
 MSGNO             INTEGER,       -- When available, message # explaining
                                -- why disabled or more details when
                                -- enabled.
 MSGTXT            VARCHAR(2000), -- Text associated with MSGNO
 SEQ               NUMBER);       -- Useful in ORDER BY clause when
                                -- selecting from this table.
```

You can use the `utlxmv.sql` script found in the `admin` directory to create `MV_CAPABILITIES_TABLE`.

Example 8–8 DBMS_MVIEW.EXPLAIN_MVIEW

First, create the materialized view. Alternatively, you can use `EXPLAIN_MVIEW` on a potential materialized view using its `SELECT` statement or the complete `CREATE MATERIALIZED VIEW` statement.

```
CREATE MATERIALIZED VIEW cal_month_sales_mv
BUILD IMMEDIATE
REFRESH FORCE
ENABLE QUERY REWRITE AS
SELECT t.calendar_month_desc,  SUM(s.amount_sold) AS dollars
FROM sales s,  times t WHERE s.time_id = t.time_id
GROUP BY t.calendar_month_desc;
```

Then, you invoke `EXPLAIN_MVIEW` with the materialized view to explain. You need to use the `SEQ` column in an `ORDER BY` clause so the rows will display in a logical order. If a capability is not possible, `N` will appear in the `P` column and an explanation in the `MSGTXT` column. If a capability is not possible for more than one reason, a row is displayed for each reason.

```
EXECUTE DBMS_MVIEW.EXPLAIN_MVIEW ('SH.CAL_MONTH_SALES_MV');

SELECT capability_name,  possible, SUBSTR(related_text,1,8)
  AS rel_text, SUBSTR(msgtxt,1,60) AS msgtxt
FROM MV_CAPABILITIES_TABLE
ORDER BY seq;
```

```
CAPABILITY_NAME                P    REL_TEXT     MSGTXT
-------------------------      -    --------     ------
PCT                            N
REFRESH_COMPLETE               Y
REFRESH_FAST                   N
REWRITE                        Y
PCT_TABLE                      N    SALES        no partition key or PMARKER in select list
PCT_TABLE                      N    TIMES        relation is not a partitioned table
REFRESH_FAST_AFTER_INSERT      N    SH.TIMES     mv log must have new values
REFRESH_FAST_AFTER_INSERT      N    SH.TIMES     mv log must have ROWID
REFRESH_FAST_AFTER_INSERT      N    SH.TIMES     mv log does not have all necessary columns
REFRESH_FAST_AFTER_INSERT      N    SH.SALES     mv log must have new values
REFRESH_FAST_AFTER_INSERT      N    SH.SALES     mv log must have ROWID
REFRESH_FAST_AFTER_INSERT      N    SH.SALES     mv log does not have all necessary columns
REFRESH_FAST_AFTER_ONETAB_DML  N    DOLLARS      SUM(expr) without COUNT(expr)
REFRESH_FAST_AFTER_ONETAB_DML  N                 see the reason why
                                                 REFRESH_FAST_AFTER_INSERT is disabled
REFRESH_FAST_AFTER_ONETAB_DML  N                 COUNT(*) is not present in the select list
REFRESH_FAST_AFTER_ONETAB_DML  N                 SUM(expr) without COUNT(expr)
REFRESH_FAST_AFTER_ANY_DML     N                 see the reason why
                                                 REFRESH_FAST_AFTER_ONETAB_DML is disabled
REFRESH_FAST_AFTER_ANY_DML     N    SH.TIMES     mv log must have sequence
REFRESH_FAST_AFTER_ANY_DML     N    SH.SALES     mv log must have sequence
REFRESH_PCT                    N                 PCT is not possible on any of the detail
                                                 tables in the materialized view
REWRITE_FULL_TEXT_MATCH        Y
REWRITE_PARTIAL_TEXT_MATCH     Y
REWRITE_GENERAL                Y
REWRITE_PCT                    N                 PCT is not possible on any detail tables
```

See Also: Chapter 15, "Maintaining the Data Warehouse" and Chapter 18, "Query Rewrite" for further details about PCT

MV_CAPABILITIES_TABLE.CAPABILITY_NAME Details

Table 8–7 lists explanations for values in the CAPABILITY_NAME column.

Table 8–7 CAPABILITY_NAME Column Details

CAPABILITY_NAME	Description
PCT	If this capability is possible, Partition Change Tracking (PCT) is possible on at least one detail relation. If this capability is not possible, PCT is not possible with any detail relation referenced by the materialized view.
REFRESH_COMPLETE	If this capability is possible, complete refresh of the materialized view is possible.
REFRESH_FAST	If this capability is possible, fast refresh is possible at least under certain circumstances.
REWRITE	If this capability is possible, at least full text match query rewrite is possible. If this capability is not possible, no form of query rewrite is possible.
PCT_TABLE	If this capability is possible, it is possible with respect to a particular partitioned table in the top level FROM list. When possible, PCT applies to the partitioned table named in the RELATED_TEXT column. PCT is needed to support fast fresh after partition maintenance operations on the table named in the RELATED_TEXT column. PCT may also support fast refresh with regard to updates to the table named in the RELATED_TEXT column when fast refresh from a materialized view log is not possible. PCT is also needed to support query rewrite in the presence of partial staleness of the materialized view with regard to the table named in the RELATED_TEXT column. When disabled, PCT does not apply to the table named in the RELATED_TEXT column. In this case, fast refresh is not possible after partition maintenance operations on the table named in the RELATED_TEXT column. In addition, PCT-based refresh of updates to the table named in the RELATED_TEXT column is not possible. Finally, query rewrite cannot be supported in the presence of partial staleness of the materialized view with regard to the table named in the RELATED_TEXT column.
PCT_TABLE_REWRITE	If this capability is possible, it is possible with respect to a particular partitioned table in the top level FROM list. When possible, PCT applies to the partitioned table named in the RELATED_TEXT column. This capability is needed to support query rewrite against this materialized view in partial stale state with regard to the table named in the RELATED_TEXT column. When disabled, query rewrite cannot be supported if this materialized view is in partial stale state with regard to the table named in the RELATED_TEXT column.
REFRESH_FAST_ AFTER_INSERT	If this capability is possible, fast refresh from a materialized view log is possible at least in the case where the updates are restricted to INSERT operations; complete refresh is also possible. If this capability is not possible, no form of fast refresh from a materialized view log is possible.

Table 8–7 (Cont.) CAPABILITY_NAME Column Details

CAPABILITY_NAME	Description
REFRESH_FAST_AFTER_ONETAB_DML	If this capability is possible, fast refresh from a materialized view log is possible regardless of the type of update operation, provided all update operations are performed on a single table. If this capability is not possible, fast refresh from a materialized view log may not be possible when the update operations are performed on multiple tables.
REFRESH_FAST_AFTER_ANY_DML	If this capability is possible, fast refresh from a materialized view log is possible regardless of the type of update operation or the number of tables updated. If this capability is not possible, fast refresh from a materialized view log may not be possible when the update operations (other than INSERT) affect multiple tables.
REFRESH_FAST_PCT	If this capability is possible, fast refresh using PCT is possible. Generally, this means that refresh is possible after partition maintenance operations on those detail tables where PCT is indicated as possible.
REWRITE_FULL_TEXT_MATCH	If this capability is possible, full text match query rewrite is possible. If this capability is not possible, full text match query rewrite is not possible.
REWRITE_PARTIAL_TEXT_MATCH	If this capability is possible, at least full and partial text match query rewrite are possible. If this capability is not possible, at least partial text match query rewrite and general query rewrite are not possible.
REWRITE_GENERAL	If this capability is possible, all query rewrite capabilities are possible, including general query rewrite and full and partial text match query rewrite. If this capability is not possible, at least general query rewrite is not possible.
REWRITE_PCT	If this capability is possible, query rewrite can use a partially stale materialized view even in QUERY_REWRITE_INTEGRITY = ENFORCED or TRUSTED modes. When this capability is not possible, query rewrite can use a partially stale materialized view only in QUERY_REWRITE_INTEGRITY = STALE_TOLERATED mode.

MV_CAPABILITIES_TABLE Column Details

Table 8–8 lists the semantics for RELATED_TEXT and RELATED_NUM columns.

Table 8–8 MV_CAPABILITIES_TABLE Column Details

MSGNO	MSGTXT	RELATED_NUM	RELATED_TEXT
NULL	NULL		For PCT capability only: [*owner.*]*name* of the table upon which PCT is enabled
2066	This statement resulted in an Oracle error	Oracle error number that occurred	
2067	No partition key or PMARKER or join dependent expression in select list in select list		[*owner.*]*name* of relation for which PCT is not supported
2068	Relation is not partitioned		[*owner.*]*name* of relation for which PCT is not supported
2069	PCT not supported with multicolumn partition key		[*owner.*]*name* of relation for which PCT is not supported
2070	PCT not supported with this type of partitioning		[*owner.*]*name* of relation for which PCT is not supported
2071	Internal error: undefined PCT failure code	The unrecognized numeric PCT failure code	[*owner.*]*name* of relation for which PCT is not supported
2072	Requirements not satisfied for fast refresh of nested mv		
2077	Mv log is newer than last full refresh		[*owner.*]*table_name* of table upon which the mv log is needed
2078	Mv log must have new values		[*owner.*]*table_name* of table upon which the mv log is needed
2079	Mv log must have ROWID		[*owner.*]*table_name* of table upon which the mv log is needed
2080	Mv log must have primary key		[*owner.*]*table_name* of table upon which the mv log is needed
2081	Mv log does not have all necessary columns		[*owner.*]*table_name* of table upon which the mv log is needed
2082	Problem with mv log		[*owner.*]*table_name* of table upon which the mv log is needed
2099	Mv references a remote table or view in the FROM list	Offset from the SELECT keyword to the table or view in question	[*owner.*]*name* of the table or view in question
2126	Multiple master sites		Name of the first different node, or NULL if the first different node is local

Table 8–8 (Cont.) MV_CAPABILITIES_TABLE Column Details

MSGNO	MSGTXT	RELATED_NUM	RELATED_TEXT
2129	Join or filter condition(s) are complex		[*owner.*]*name* of the table involved with the join or filter condition (or NULL when not available)
2130	Expression not supported for fast refresh	Offset from the SELECT keyword to the expression in question	The alias name in the select list of the expression in question
2150	Select lists must be identical across the UNION operator	Offset from the SELECT keyword to the first different select item in the SELECT list	The alias name of the first different select item in the SELECT list
2182	PCT is enabled through a join dependency		[*owner.*]*name* of relation for which PCT_TABLE_REWRITE is not enabled
2183	Expression to enable PCT not in PARTITION BY of analytic function or spreadsheet	The unrecognized numeric PCT failure code	[*owner.*]*name* of relation for which PCT is not enabled
2184	Expression to enable PCT cannot be rolled up		[*owner.*]*name* of relation for which PCT is not enabled
2185	No partition key or PMARKER in the SELECT list		[*owner.*]*name* of relation for which PCT_TABLE_REWRITE is not enabled
2186	GROUP OUTER JOIN is present		
2187	materialized view on external table		

9

Advanced Materialized Views

This chapter discusses advanced topics in using materialized views:

- Partitioning and Materialized Views
- Materialized Views in OLAP Environments
- Materialized Views and Models
- Invalidating Materialized Views
- Security Issues with Materialized Views
- Altering Materialized Views

Partitioning and Materialized Views

Because of the large volume of data held in a data warehouse, partitioning is an extremely useful option when designing a database. Partitioning the fact tables improves scalability, simplifies system administration, and makes it possible to define local indexes that can be efficiently rebuilt. Partitioning the fact tables also improves the opportunity of fast refreshing the materialized view because this may enable Partition Change Tracking (PCT) refresh on the materialized view. Partitioning a materialized view also has benefits for refresh, because the refresh procedure can then use parallel DML in more scenarios and PCT-based refresh can use truncate partition to efficiently maintain the materialized view. See Chapter 5, "Parallelism and Partitioning in Data Warehouses" for further details about partitioning.

Partition Change Tracking

It is possible and advantageous to track freshness to a finer grain than the entire materialized view. The ability to identify which rows in a materialized view are affected by a certain detail table partition, is known as Partition Change Tracking (PCT). When one or more of the detail tables are partitioned, it may be possible to identify the specific rows in the materialized view that correspond to a modified detail partition(s); those rows become stale when a partition is modified while all other rows remain fresh.

You can use PCT to identify which materialized view rows correspond to a particular partition. PCT is also used to support fast refresh after partition maintenance operations on detail tables. For instance, if a detail table partition is truncated or dropped, the affected rows in the materialized view are identified and deleted.

Identifying which materialized view rows are fresh or stale, rather than considering the entire materialized view as stale, allows query rewrite to use those rows that are fresh while in `QUERY_REWRITE_INTEGRITY=ENFORCED` or `TRUSTED` modes. Oracle does not rewrite against partial stale materialized view if partition change tracking on the changed table is enabled by the presence of join dependent expression in the materialized view. See "Join Dependent Expression" on page 9-4.

To support PCT, a materialized view must satisfy the following requirements:

- At least one of the detail tables referenced by the materialized view must be partitioned.
- Partitioned tables must use either range, list or composite partitioning.

- The top level partition key must consist of only a single column.
- The materialized view must contain either the partition key column or a partition marker or ROWID or join dependent expression of the detail table. See *PL/SQL Packages and Types Reference* for details regarding the DBMS_MVIEW.PMARKER function.
- If you use a GROUP BY clause, the partition key column or the partition marker or ROWID or join dependent expression must be present in the GROUP BY clause.
- If you use an analytic window function or the MODEL clause, the partition key column or the partition marker or ROWID or join dependent expression must be present in their respective PARTITION BY subclauses.
- Data modifications can only occur on the partitioned table. If PCT refresh is being done for a table which has join dependent expression in the materialized view, then data modifications should not have occurred in any of the join dependent tables.
- The COMPATIBILITY initialization parameter must be a minimum of 9.0.0.0.0.
- PCT is not supported for a materialized view that refers to views, remote tables, or outer joins.
- PCT-based refresh is not supported for UNION ALL materialized views.

Partition Key

Partition change tracking requires sufficient information in the materialized view to be able to correlate a detail row in the source partitioned detail table to the corresponding materialized view row. This can be accomplished by including the detail table partition key columns in the SELECT list and, if GROUP BY is used, in the GROUP BY list.

Consider an example of a materialized view storing daily customer sales. The following example uses the sh sample schema and the three detail tables sales, products, and times to create the materialized view. sales table is partitioned by time_id column and products is partitioned by the prod_id column. times is not a partitioned table.

Example 9–1 Partition Key

The detail tables must have materialized view logs for FAST REFRESH. The following is an example:

```
CREATE MATERIALIZED VIEW LOG ON SALES WITH ROWID
   (prod_id, time_id, quantity_sold, amount_sold) INCLUDING NEW VALUES;
```

```
CREATE MATERIALIZED VIEW LOG ON PRODUCTS WITH ROWID
   (prod_id, prod_name, prod_desc) INCLUDING NEW VALUES;
CREATE MATERIALIZED VIEW LOG ON TIMES WITH ROWID
   (time_id, calendar_month_name, calendar_year) INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW cust_dly_sales_mv
BUILD DEFERRED REFRESH FAST ON DEMAND
ENABLE QUERY REWRITE AS
SELECT s.time_id, p.prod_id, p.prod_name, COUNT(*),
         SUM(s.quantity_sold), SUM(s.amount_sold),
         COUNT(s.quantity_sold), COUNT(s.amount_sold)
FROM sales s, products p, times t
WHERE  s.time_id = t.time_id AND s.prod_id = p.prod_id
GROUP BY s.time_id, p.prod_id, p.prod_name;
```

For `cust_dly_sales_mv`, PCT is enabled on both the sales table and products table because their respective partitioning key columns `time_id` and `prod_id` are in the materialized view.

Join Dependent Expression

An expression consisting of columns from tables directly or indirectly joined through equijoins to the partitioned detail table on the partitioning key and which is either a dimensional attribute or a dimension hierarchical parent of the joining key is called a join dependent expression. The set of tables in the path to detail table are called join dependent tables.

```
SELECT s.time_id, t.calendar_month_name
FROM sales s, times t WHERE s.time_id = t.time_id;
```

In this query, `times` table is a join dependent table since it is joined to `sales` table on the partitioning key column `time_id`. Moreover, `calendar_month_name` is a dimension hierarchical attribute of `times.time_id`, because `calendar_month_name` is an attribute of `times.mon_id` and `times.mon_id` is a dimension hierarchical parent of `times.time_id`. Hence, the expression `calendar_month_name` from `times` tables is a join dependent expression. Let's look at another example:

```
SELECT s.time_id, y.calendar_year_name
FROM sales s, times_d d, times_m m, times_y y
WHERE s.time_id = d.time_id AND d.day_id = m.day_id AND m.mon_id = y.mon_id;
```

Here, `times` table is denormalized into `times_d`, `times_m` and `times_y` tables. The expression `calendar_year_name` from `times_y` table is a join dependent

expression and the tables times_d, times_m and times_y are join dependent tables. This is because times_y table is joined indirectly through times_m and times_d tables to sales table on its partitioning key column time_id.

This lets users create materialized views containing aggregates on some level higher than the partitioning key of the detail table. Consider the following example of materialized view storing monthly customer sales.

Example 9–2 Join Dependent Expression

Assuming the presence of materialized view logs defined earlier, the materialized view can be created using the following DDL:

```
CREATE MATERIALIZED VIEW cust_mth_sales_mv
BUILD DEFERRED REFRESH FAST ON DEMAND
ENABLE QUERY REWRITE AS
SELECT  t.calendar_month_name, p.prod_id, p.prod_name, COUNT(*),
    SUM(s.quantity_sold), SUM(s.amount_sold),
    COUNT(s.quantity_sold), COUNT(s.amount_sold)
FROM sales s, products p, times t
WHERE  s.time_id = t.time_id AND s.prod_id = p.prod_id
GROUP BY t.calendar_month_name, p.prod_id, p.prod_name;
```

Here, you can correlate a detail table row to its corresponding materialized view row using the join dependent table times and the relationship that times.calendar_month_name is a dimensional attribute determined by times.time_id. This enables partition change tracking on sales table. In addition to this, PCT is enabled on products table because of presence of its partitioning key column prod_id in the materialized view.

Partition Marker

The DBMS_MVIEW.PMARKER function is designed to significantly reduce the cardinality of the materialized view (see Example 9–3 on page 9-6 for an example). The function returns a partition identifier that uniquely identifies the partition for a specified row within a specified partition table. Therefore, the DBMS_MVIEW.PMARKER function is used instead of the partition key column in the SELECT and GROUP BY clauses.

Unlike the general case of a PL/SQL function in a materialized view, use of the DBMS_MVIEW.PMARKER does not prevent rewrite with that materialized view even when the rewrite mode is QUERY_REWRITE_INTEGRITY=ENFORCED.

As an example of using the PMARKER function, consider calculating a typical number, such as revenue generated by a product category during a given year. If

there were 1000 different products sold each month, it would result in 12,000 rows in the materialized view.

Example 9–3 Partition Marker

Consider an example of a materialized view storing the yearly sales revenue for each product category. With approximately hundreds of different products in each product category, including the partitioning key column prod_id of products table in the materialized view would substantially increase the cardinality. Instead, this materialized view uses the DBMS_MVIEW.PMARKER function, which increases the cardinality of materialized view by a factor of the number of partitions in the products table.

```
CREATE MATERIALIZED VIEW prod_yr_sales_mv
BUILD DEFERRED
REFRESH FAST ON DEMAND
ENABLE QUERY REWRITE AS
SELECT DBMS_MVIEW.PMARKER(p.rowid), p.prod_category, t.calendar_year, COUNT(*),
       SUM(s.amount_sold), SUM(s.quantity_sold),
       COUNT(s.amount_sold), COUNT(s.quantity_sold)
FROM   sales s, products p, times t
WHERE  s.time_id = t.time_id AND s.prod_id = p.prod_id
GROUP BY  DBMS_MVIEW.PMARKER (p.rowid), p.prod_category, t.calendar_year;
```

prod_yr_sales_mv includes the DBMS_MVIEW.PMARKER function on the products table in its SELECT list. This enables partition change tracking on products table with significantly less cardinality impact than grouping by the partition key column prod_id. In this example, the desired level of aggregation for the prod_yr_sales_mv is to group by products.prod_category. Using the DBMS_MVIEW.PMARKER function, the materialized view cardinality is increased only by a factor of the number of partitions in the products table. This would generally be significantly less than the cardinality impact of including the partition key columns.

Please note that partition change tracking is enabled on sales table because of presence of join dependent expression calendar_year in the SELECT list.

Partial Rewrite

A subsequent INSERT statement adds a new row to the sales_part3 partition of table sales. At this point, because cust_dly_sales_mv has PCT available on table sales using a partition key, Oracle can identify the stale rows in the materialized view cust_dly_sales_mv corresponding to sales_part3 partition (The other rows are unchanged in their freshness state). Query rewrite cannot

identify the fresh portion of materialized views cust_mth_sales_mv and prod_yr_sales_mv because PCT is available on table sales using join dependent expressions. Query rewrite can determine the fresh portion of a materialized view on changes to a detail table only if PCT is available on the detail table using a partition key or partition marker.

Partitioning a Materialized View

Partitioning a materialized view involves defining the materialized view with the standard Oracle partitioning clauses, as illustrated in the following example. This statement creates a materialized view called part_sales_mv, which uses three partitions, can be fast refreshed, and is eligible for query rewrite.

```
CREATE MATERIALIZED VIEW part_sales_mv
PARALLEL PARTITION BY RANGE (time_id)
(PARTITION month1
      VALUES LESS THAN (TO_DATE('31-12-1998', 'DD-MM-YYYY'))
      PCTFREE 0 PCTUSED 99
      STORAGE (INITIAL 64k NEXT 16k PCTINCREASE 0)
      TABLESPACE sf1,
 PARTITION month2
      VALUES LESS THAN (TO_DATE('31-12-1999', 'DD-MM-YYYY'))
      PCTFREE 0 PCTUSED 99
      STORAGE (INITIAL 64k NEXT 16k PCTINCREASE 0)
      TABLESPACE sf2,
 PARTITION month3
      VALUES LESS THAN (TO_DATE('31-12-2000', 'DD-MM-YYYY'))
      PCTFREE 0 PCTUSED 99
      STORAGE (INITIAL 64k NEXT 16k PCTINCREASE 0)
      TABLESPACE sf3)
BUILD DEFERRED
REFRESH FAST
ENABLE QUERY REWRITE AS
SELECT s.cust_id, s.time_id,
  SUM(s.amount_sold) AS sum_dol_sales, SUM(s.quantity_sold) AS sum_unit_sales
  FROM sales s GROUP BY s.time_id, s.cust_id;
```

Partitioning a Prebuilt Table

Alternatively, a materialized view can be registered to a partitioned prebuilt table as illustrated in the following example:

```
CREATE TABLE part_sales_tab(time_id, cust_id, sum_dollar_sales, sum_unit_sale)
PARALLEL PARTITION BY RANGE (time_id)
```

```
(PARTITION month1
      VALUES LESS THAN (TO_DATE('31-12-1998', 'DD-MM-YYYY'))
      PCTFREE 0 PCTUSED 99
      STORAGE (INITIAL 64k NEXT 16k PCTINCREASE 0)
      TABLESPACE sf1,
 PARTITION month2
      VALUES LESS THAN (TO_DATE('31-12-1999', 'DD-MM-YYYY'))
      PCTFREE 0 PCTUSED 99
      STORAGE (INITIAL 64k NEXT 16k PCTINCREASE 0)
      TABLESPACE sf2,
PARTITION month3
      VALUES LESS THAN (TO_DATE('31-12-2000', 'DD-MM-YYYY'))
      PCTFREE 0 PCTUSED 99
      STORAGE (INITIAL 64k NEXT 16k PCTINCREASE 0)
      TABLESPACE sf3)  AS
SELECT s.time_id, s.cust_id, SUM(s.amount_sold) AS sum_dollar_sales,
  SUM(s.quantity_sold) AS sum_unit_sales
FROM sales s GROUP BY s.time_id, s.cust_id;

CREATE MATERIALIZED VIEW part_sales_tab_mv
ON PREBUILT TABLE
ENABLE QUERY REWRITE AS
SELECT s.time_id,  s.cust_id, SUM(s.amount_sold) AS sum_dollar_sales,
     SUM(s.quantity_sold) AS sum_unit_sales
FROM sales s GROUP BY s.time_id, s.cust_id;
```

In this example, the table part_sales_tab has been partitioned over three months and then the materialized view was registered to use the prebuilt table. This materialized view is eligible for query rewrite because the ENABLE QUERY REWRITE clause has been included.

Benefits of Partitioning a Materialized View

When a materialized view is partitioned on the partitioning key column or join dependent expressions of the detail table, it is more efficient to use a TRUNCATE PARTITION statement to remove one or more partitions of the materialized view during refresh and then repopulate the partition with new data. Oracle Database uses this variant of fast refresh (called PCT refresh) with partition truncation if the following conditions are satisfied in addition to other conditions described in "Partition Change Tracking" on page 9-2.

- The materialized view is partitioned on the partitioning key column or join dependent expressions of the detail table.

- If PCT is enabled using either the partitioning key column or join expressions, both the materialized view should be range or list partitioned.
- PCT refresh is non-atomic.

Rolling Materialized Views

When a data warehouse or data mart contains a time dimension, it is often desirable to archive the oldest information and then reuse the storage for new information. This is called the rolling window scenario. If the fact tables or materialized views include a time dimension and are horizontally partitioned by the time attribute, then management of rolling materialized views can be reduced to a few fast partition maintenance operations provided the unit of data that is rolled out equals, or is at least aligned with, the range partitions.

If you plan to have rolling materialized views in your data warehouse, you should determine how frequently you plan to perform partition maintenance operations, and you should plan to partition fact tables and materialized views to reduce the amount of system administration overhead required when old data is aged out. An additional consideration is that you might want to use data compression on your infrequently updated partitions.

You are not restricted to using range partitions. For example, a composite partition using both a time value and a key value could result in a good partition solution for your data.

See Chapter 15, "Maintaining the Data Warehouse" for further details regarding `CONSIDER FRESH` and for details regarding compression.

Materialized Views in OLAP Environments

This section discusses certain OLAP concepts and how relational databases can handle OLAP queries. Next, it recommends an approach for using materialized views to meet OLAP performance needs. Finally, it discusses using materialized views with set operators, a common scenario for OLAP environments.

OLAP Cubes

While data warehouse environments typically view data in the form of a star schema, OLAP environments view data in the form of a hierarchical cube. A hierarchical cube includes the data aggregated along the rollup hierarchy of each of its dimensions and these aggregations are combined across dimensions. It includes the typical set of aggregations needed for business intelligence queries.

Example 9–4 Hierarchical Cube

Consider a sales data set with two dimensions, each of which has a 4-level hierarchy:

- Time, which contains (all times), year, quarter, and month.
- Product, which contains (all products), division, brand, and item.

This means there are 16 aggregate groups in the hierarchical cube. This is because the four levels of time are multiplied by four levels of product to produce the cube. Table 9–1 shows the four levels of each dimension.

Table 9–1 ROLLUP By Time and Product

ROLLUP By Time	ROLLUP By Product
year, quarter, month	division, brand, item
year, quarter	division, brand
year	division
all times	all products

Note that as you increase the number of dimensions and levels, the number of groups to calculate increases dramatically. This example involves 16 groups, but if you were to add just two more dimensions with the same number of levels, you would have 4 x 4 x 4 x 4 = 256 different groups. Also, consider that a similar increase in groups occurs if you have multiple hierarchies in your dimensions. For example, the time dimension might have an additional hierarchy of fiscal month rolling up to fiscal quarter and then fiscal year. Handling the explosion of groups has historically been the major challenge in data storage for OLAP systems.

Typical OLAP queries **slice and dice** different parts of the cube comparing aggregations from one level to aggregation from another level. For instance, a query might find sales of the grocery division for the month of January, 2002 and compare them with total sales of the grocery division for all of 2001.

Partitioning Materialized Views for OLAP

Materialized views with multiple aggregate groups will give their best performance for refresh and query rewrite when partitioned appropriately.

PCT refresh in a rolling window scenario requires partitioning at the top level on some level from the time dimension. And, partition pruning for queries rewritten against this materialized view requires partitioning on `GROUPING_ID` column.

Hence, the most effective partitioning scheme for these materialized views is to use composite partitioning (range-list on (time, GROUPING_ID) columns). By partitioning the materialized views this way, you enable:

- PCT refresh, thereby improving refresh performance.
- Partition pruning: only relevant aggregate groups will be accessed, thereby greatly reducing the query processing cost.

If you do not want to use PCT refresh, you can just partition by list on GROUPING_ID column.

Compressing Materialized Views for OLAP

You should consider data compression when using highly redundant data, such as tables with many foreign keys. In particular, materialized views created with the ROLLUP clause are likely candidates. See *Oracle Database SQL Reference* for data compression syntax and restrictions and "Storage And Table Compression" on page 8-22 for details regarding compression.

Materialized Views with Set Operators

Oracle provides support for materialized views whose defining query involves set operators. Materialized views with set operators can now be created enabled for query rewrite. You can refresh the materialized view using either ON COMMIT or ON DEMAND refresh.

Fast refresh is supported if the defining query has the UNION ALL operator at the top level and each query block in the UNION ALL, meets the requirements of a materialized view with aggregates or materialized view with joins only. Further, the materialized view must include a constant column (known as a UNION ALL marker) that has a distinct value in each query block, which, in the following example, is columns 1 marker and 2 marker.

See "Restrictions on Fast Refresh on Materialized Views with UNION ALL" on page 8-29 for detailed restrictions on fast refresh for materialized views with UNION ALL.

Examples of Materialized Views Using UNION ALL

The following examples illustrate creation of fast refreshable materialized views involving UNION ALL.

Example 9–5 Materialized View Using UNION ALL with Two Join Views

To create a UNION ALL materialized view with two join views, the materialized view logs must have the rowid column and, in the following example, the UNION ALL marker is the columns, `1 marker` and `2 marker`.

```
CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID;
CREATE MATERIALIZED VIEW LOG ON customers WITH ROWID;

CREATE MATERIALIZED VIEW unionall_sales_cust_joins_mv
REFRESH FAST ON COMMIT
ENABLE QUERY REWRITE AS
(SELECT c.rowid crid, s.rowid srid, c.cust_id, s.amount_sold, 1 marker
FROM sales s, customers c
WHERE s.cust_id = c.cust_id AND c.cust_last_name = 'Smith')
UNION ALL
(SELECT c.rowid crid, s.rowid srid, c.cust_id, s.amount_sold, 2 marker
FROM sales s, customers c
WHERE s.cust_id = c.cust_id AND c.cust_last_name = 'Brown');
```

Example 9–6 Materialized View Using UNION ALL with Joins and Aggregates

The following example shows a UNION ALL of a materialized view with joins and a materialized view with aggregates. A couple of things can be noted in this example. Nulls or constants can be used to ensure that the data types of the corresponding SELECT list columns match. Also the UNION ALL marker column can be a string literal, which is `'Year' umarker`, `'Quarter' umarker`, or `'Daily' umarker` in the following example:

```
CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID, SEQUENCE
(amount_sold, time_id)
INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW LOG ON times WITH ROWID, SEQUENCE
  (time_id, fiscal_year, fiscal_quarter_number, day_number_in_week)
INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW unionall_sales_mix_mv
REFRESH FAST ON DEMAND AS
(SELECT 'Year' umarker, NULL, NULL, t.fiscal_year,
        SUM(s.amount_sold) amt, COUNT(s.amount_sold), COUNT(*)
 FROM sales s, times t
 WHERE s.time_id = t.time_id
 GROUP BY t.fiscal_year)
UNION ALL
(SELECT 'Quarter' umarker, NULL, NULL, t.fiscal_quarter_number,
```

```
       SUM(s.amount_sold) amt, COUNT(s.amount_sold), COUNT(*)
FROM sales s, times t
WHERE s.time_id = t.time_id and t.fiscal_year = 2001
GROUP BY t.fiscal_quarter_number)
UNION ALL
(SELECT 'Daily' umarker, s.rowid rid, t.rowid rid2, t.day_number_in_week,
       s.amount_sold amt, 1, 1
FROM sales s, times t
WHERE s.time_id = t.time_id
AND t.time_id between '01-Jan-01' AND '01-Dec-31');
```

Materialized Views and Models

Models, which provide array-based computations in SQL, can be used in materialized views. Because the MODEL clause calculations can be expensive, you may want to use two separate materialized views: one for the model calculations and one for the SELECT ... GROUP BY query. For example, instead of using one, long materialized view, you could create the following materialized views:

```
CREATE MATERIALIZED VIEW my_groupby_mv
REFRESH FAST
ENABLE QUERY REWRITE AS
SELECT country_name country, prod_name prod, calendar_year year,
  SUM(amount_sold) sale, COUNT(amount_sold) cnt, COUNT(*) cntstr
FROM sales, times, customers, countries, products
WHERE sales.time_id = times.time_id AND
      sales.prod_id = products.prod_id AND
      sales.cust_id = customers.cust_id AND
      customers.country_id = countries.country_id
GROUP BY country_name, prod_name, calendar_year;

CREATE MATERIALIZED VIEW my_model_mv
ENABLE QUERY REWRITE AS
SELECT country, prod, year, sale, cnt
FROM my_groupby_mv
MODEL PARTITION BY(country) DIMENSION BY(prod, year)
   MEASURES(sale s) IGNORE NAV
(s['Shorts', 2000] =  0.2 * AVG(s)[CURRENTV(), year BETWEEN 1996 AND 1999],
s['Kids Pajama', 2000] = 0.5 * AVG(s)[CURRENTV(), year BETWEEN 1995 AND 1999],
s['Boys Pajama', 2000] = 0.6 * AVG(s)[CURRENTV(), year BETWEEN 1994 AND 1999],
...
<hundreds of other update rules>);
```

By using two materialized views, you can incrementally maintain the materialized view my_groupby_mv. The materialized view my_model_mv is on a much smaller data set because it is built on my_groupby_mv and can be maintained by a complete refresh.

Materialized views with models can use complete refresh or PCT refresh only, and are available for partial text query rewrite only.

See Chapter 22, "SQL for Modeling" for further details about model calculations.

Invalidating Materialized Views

Dependencies related to materialized views are automatically maintained to ensure correct operation. When a materialized view is created, the materialized view depends on the detail tables referenced in its definition. Any DML operation, such as a INSERT, or DELETE, UPDATE, or DDL operation on any dependency in the materialized view will cause it to become invalid. To revalidate a materialized view, use the ALTER MATERIALIZED VIEW COMPILE statement.

A materialized view is automatically revalidated when it is referenced. In many cases, the materialized view will be successfully and transparently revalidated. However, if a column has been dropped in a table referenced by a materialized view or the owner of the materialized view did not have one of the query rewrite privileges and that privilege has now been granted to the owner, you should use the following statement to revalidate the materialized view:

```
ALTER MATERIALIZED VIEW mview_name COMPILE;
```

The state of a materialized view can be checked by querying the data dictionary views USER_MVIEWS or ALL_MVIEWS. The column STALENESS will show one of the values FRESH, STALE, UNUSABLE, UNKNOWN, UNDEFINED, or NEEDS_COMPILE to indicate whether the materialized view can be used. The state is maintained automatically. However, if the staleness of a materialized view is marked as NEEDS_COMPILE, you could issue an ALTER MATERIALIZED VIEW ... COMPILE statement to validate the materialized view and get the correct staleness state.

Security Issues with Materialized Views

To create a materialized view in your own schema, you must have the CREATE MATERIALIZED VIEW privilege and the SELECT privilege to any tables referenced that are in another schema. To create a materialized view in another schema, you must have the CREATE ANY MATERIALIZED VIEW privilege and the owner of the materialized view needs SELECT privileges to the tables referenced if they are from

another schema. Moreover, if you enable query rewrite on a materialized view that references tables outside your schema, you must have the GLOBAL QUERY REWRITE privilege or the QUERY REWRITE object privilege on each table outside your schema.

If the materialized view is on a prebuilt container, the creator, if different from the owner, must have SELECT WITH GRANT privilege on the container table.

If you continue to get a privilege error while trying to create a materialized view and you believe that all the required privileges have been granted, then the problem is most likely due to a privilege not being granted explicitly and trying to inherit the privilege from a role instead. The owner of the materialized view must have explicitly been granted SELECT access to the referenced tables if the tables are in a different schema.

If the materialized view is being created with ON COMMIT REFRESH specified, then the owner of the materialized view requires an additional privilege if any of the tables in the defining query are outside the owner's schema. In that case, the owner requires the ON COMMIT REFRESH system privilege or the ON COMMIT REFRESH object privilege on each table outside the owner's schema.

Querying Materialized Views with Virtual Private Database

For all security concerns, a materialized view serves as a view that happens to be materialized when you are directly querying the materialized view. When creating a view or materialized view, the owner needs to have the necessary permissions to access the underlying base relations of the view or materialized view that they are creating. With these permissions, the owner can publish a view or materialized view that other users can access, assuming they have been granted access to the view or materialized view.

Using materialized views with Virtual Private Database (VPD) is similar. When you create a materialized view, there must not be any VPD policies in effect against the base relations of the materialized view for the owner of the materialized view. However, the owner of the materialized view may establish a VPD policy on the new materialized view. Users who access the materialized view are subject to the VPD policy on the materialized view. However, they are not additionally subject to the VPD policies of the underlying base relations of the materialized view, since security processing of the underlying base relations is performed against the owner of the materialized view.

Using Query Rewrite with Virtual Private Database

When you access a materialized view using query rewrite, the materialized view serves as an access structure much like an index. As such, the security implications for materialized views accessed in this way are much the same as for indexes: all security checks are performed against the relations specified in the request query. The index or materialized view is used to speed the performance of accessing the data, not provide any additional security checks. Thus, the presence of the index or materialized view presents no additional security checking.

This holds true when you are accessing a materialized view using query rewrite in the presence of Virtual Private Database (VPD). The request query is subject to any VPD policies that are present against the relations specified in the query. Query rewrite may rewrite the query to use a materialize view instead of accessing the detail relations, but only if it can guarantee to deliver exactly the same rows as if the rewrite had not occurred. Specifically, query rewrite must retain and respect any VPD policies against the relations specified in the request query. However, any VPD policies against the materialized view itself do not have effect when the materialized view is accessed using query rewrite. This is because the data is already protected by the VPD policies against the relations in the request query.

Restrictions with Materialized Views and Virtual Private Database

Query rewrite does not use its full and partial text match modes with request queries that include relations with active VPD policies, but it does use general rewrite methods. This is because VPD transparently transforms the request query to affect the VPD policy. If query rewrite were to perform a text match transformation against a request query with a VPD policy, the effect would be to negate the VPD policy.

In addition, when you create or refresh a materialized view, the owner of the materialized view must not have any active VPD policies in effect against the base relations of the materialized view, or an error is returned. The materialized view owner must either have no such VPD policies, or any such policy must return `NULL`. This is because VPD would transparently modify the defining query of the materialized view such that the set of rows contained by the materialized view would not match the set of rows indicated by the materialized view definition.

One way to work around this restriction yet still create a materialized view containing the desired VPD-specified subset of rows is to create the materialized view in a user account that has no active VPD policies against the detail relations of the materialized view. In addition, you can include a predicate in the `WHERE` clause of the materialized view that embodies the effect of the VPD policy. When query rewrite attempts to rewrite a request query that has that VPD policy, it will match

up the VPD-generated predicate on the request query with the predicate you directly specify when you create the materialized view.

Altering Materialized Views

Six modifications can be made to a materialized view. You can:

- Change its refresh option (FAST/FORCE/COMPLETE/NEVER).
- Change its refresh mode (ON COMMIT/ON DEMAND).
- Recompile it.
- Enable or disable its use for query rewrite.
- Consider it fresh.
- Partition maintenance operations.

All other changes are achieved by dropping and then re-creating the materialized view.

The COMPILE clause of the ALTER MATERIALIZED VIEW statement can be used when the materialized view has been invalidated. This compile process is quick, and allows the materialized view to be used by query rewrite again.

> **See Also:** *Oracle Database SQL Reference* for further information about the ALTER MATERIALIZED VIEW statement and "Invalidating Materialized Views" on page 9-14

10

Dimensions

The following sections will help you create and manage a data warehouse:

What are Dimensions?

A **dimension** is a structure that categorizes data in order to enable users to answer business questions. Commonly used dimensions are customers, products, and time. For example, each sales channel of a clothing retailer might gather and store data regarding sales and reclamations of their Cloth assortment. The retail chain management can build a data warehouse to analyze the sales of its products across all stores over time and help answer questions such as:

- What is the effect of promoting one product on the sale of a related product that is not promoted?
- What are the sales of a product before and after a promotion?
- How does a promotion affect the various distribution channels?

The data in the retailer's data warehouse system has two important components: dimensions and facts. The dimensions are products, customers, promotions, channels, and time. One approach for identifying your dimensions is to review your reference tables, such as a product table that contains everything about a product, or a promotion table containing all information about promotions. The facts are sales (units sold) and profits. A data warehouse contains facts about the sales of each product at on a daily basis.

A typical relational implementation for such a data warehouse is a star schema. The fact information is stored in what is called a fact table, whereas the dimensional information is stored in dimension tables. In our example, each sales transaction record is uniquely defined as for each customer, for each product, for each sales channel, for each promotion, and for each day (time).

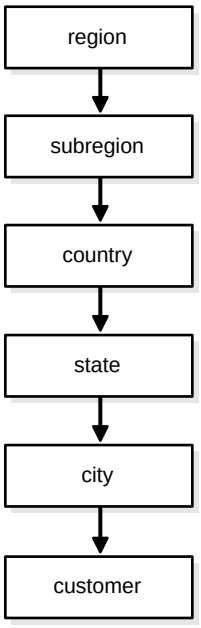
> **See Also:** Chapter 19, "Schema Modeling Techniques" for further details

In Oracle Database, the dimensional information itself is stored in a dimension table. In addition, the database object dimension helps to organize and group dimensional information into hierarchies. This represents natural `1:n` relationships between columns or column groups (the levels of a hierarchy) that cannot be represented with constraint conditions. Going up a level in the hierarchy is called rolling up the data and going down a level in the hierarchy is called drilling down the data. In the retailer example:

- Within the `time` dimension, months roll up to quarters, quarters roll up to years, and years roll up to all years.

- Within the `product` dimension, products roll up to subcategories, subcategories roll up to categories, and categories roll up to all products.
- Within the `customer` dimension, customers roll up to `city`. Then cities roll up to `state`. Then states roll up to `country`. Then countries roll up to `subregion`. Finally, subregions roll up to `region`, as shown in Figure 10–1.

Figure 10–1 Sample Rollup for a Customer Dimension



Data analysis typically starts at higher levels in the dimensional hierarchy and gradually drills down if the situation warrants such analysis.

Dimensions do not have to be defined. However, if your application uses dimensional modeling, it is worth spending time creating them as it can yield significant benefits, because they help query rewrite perform more complex types of rewrite. Dimensions are also beneficial to certain types of materialized view refresh operations and with the SQLAccess Advisor. They are only mandatory if you use the SQLAccess Advisor (a GUI tool for materialized view and index management) without a workload to recommend which materialized views and indexes to create, drop, or retain.

> **See Also:** Chapter 18, "Query Rewrite" for further details regarding query rewrite and Chapter 17, "SQLAccess Advisor" for further details regarding the SQLAccess Advisor

In spite of the benefits of dimensions, you must not create dimensions in any schema that does not fully satisfy the dimensional relationships described in this chapter. Incorrect results can be returned from queries otherwise.

Creating Dimensions

Before you can create a dimension object, the dimension tables must exist in the database possibly containing the dimension data. For example, if you create a customer dimension, one or more tables must exist that contain the city, state, and country information. In a star schema data warehouse, these dimension tables already exist. It is therefore a simple task to identify which ones will be used.

Now you can draw the hierarchies of a dimension as shown in Figure 10–1. For example, `city` is a child of `state` (because you can aggregate city-level data up to state), and `country`. This hierarchical information will be stored in the database object dimension.

In the case of normalized or partially normalized dimension representation (a dimension that is stored in more than one table), identify how these tables are joined. Note whether the joins between the dimension tables can guarantee that each child-side row joins with one and only one parent-side row. In the case of denormalized dimensions, determine whether the child-side columns uniquely determine the parent-side (or attribute) columns. If you use constraints to represent these relationships, they can be enabled with the `NOVALIDATE` and `RELY` clauses if the relationships represented by the constraints are guaranteed by other means.

You create a dimension using either the `CREATE DIMENSION` statement or the Dimension Wizard in Oracle Enterprise Manager. Within the `CREATE DIMENSION` statement, use the `LEVEL` clause to identify the names of the dimension levels.

> **See Also:** *Oracle Database SQL Reference* for a complete description of the `CREATE DIMENSION` statement

This customer dimension contains a single hierarchy with a geographical rollup, with arrows drawn from the child level to the parent level, as shown in Figure 10–1 on page 10-3.

Each arrow in this graph indicates that for any child there is one and only one parent. For example, each city must be contained in exactly one state and each state

must be contained in exactly one country. States that belong to more than one country, or that belong to no country, violate hierarchical integrity. Hierarchical integrity is necessary for the correct operation of management functions for materialized views that include aggregates.

For example, you can declare a dimension `products_dim`, which contains levels product, subcategory, and category:

```
CREATE DIMENSION products_dim
       LEVEL product           IS (products.prod_id)
       LEVEL subcategory       IS (products.prod_subcategory)
       LEVEL category          IS (products.prod_category) ...
```

Each level in the dimension must correspond to one or more columns in a table in the database. Thus, level product is identified by the column `prod_id` in the products table and level subcategory is identified by a column called `prod_subcategory` in the same table.

In this example, the database tables are denormalized and all the columns exist in the same table. However, this is not a prerequisite for creating dimensions. "Using Normalized Dimension Tables" on page 10-10 shows how to create a dimension `customers_dim` that has a normalized schema design using the `JOIN KEY` clause.

The next step is to declare the relationship between the levels with the `HIERARCHY` statement and give that hierarchy a name. A hierarchical relationship is a functional dependency from one level of a hierarchy to the next level in the hierarchy. Using the level names defined previously, the `CHILD OF` relationship denotes that each child's level value is associated with one and only one parent level value. The following statement declares a hierarchy `prod_rollup` and defines the relationship between products, subcategory, and category.

```
HIERARCHY prod_rollup
 (product         CHILD OF
  subcategory     CHILD OF
  category)
```

In addition to the `1:n` hierarchical relationships, dimensions also include `1:1` attribute relationships between the hierarchy levels and their dependent, determined dimension attributes. For example, the dimension `times_dim`, as defined in *Oracle Database Sample Schemas*, has columns `fiscal_month_desc`, `fiscal_month_name`, and `days_in_fiscal_month`. Their relationship is defined as follows:

```
LEVEL fis_month   IS TIMES.FISCAL_MONTH_DESC
...
```

```
ATTRIBUTE fis_month DETERMINES
      (fiscal_month_name, days_in_fiscal_month)
```

The ATTRIBUTE ... DETERMINES clause relates fis_month to fiscal_month_name and days_in_fiscal_month. Note that this is a unidirectional determination. It is only guaranteed, that for a specific fiscal_month, for example, 1999-11, you will find exactly one matching values for fiscal_month_name, for example, November and days_in_fiscal_month, for example, 28. You cannot determine a specific fiscal_month_desc based on the fiscal_month_name, which is November for every fiscal year.

In this example, suppose a query were issued that queried by fiscal_month_name instead of fiscal_month_desc. Because this 1:1 relationship exists between the attribute and the level, an already aggregated materialized view containing fiscal_month_desc can be joined back to the dimension information and used to identify the data.

> **See Also:** Chapter 18, "Query Rewrite" for further details of using dimensional information

A sample dimension definition follows:

```
CREATE DIMENSION products_dim
    LEVEL product           IS (products.prod_id)
    LEVEL subcategory       IS (products.prod_subcategory)
    LEVEL category          IS (products.prod_category)
    HIERARCHY prod_rollup (
         product         CHILD OF
         subcategory     CHILD OF
         category)
    ATTRIBUTE product DETERMINES
     (products.prod_name, products.prod_desc,
      prod_weight_class, prod_unit_of_measure,
      prod_pack_size, prod_status, prod_list_price, prod_min_price)
    ATTRIBUTE subcategory DETERMINES
     (prod_subcategory, prod_subcategory_desc)
    ATTRIBUTE category DETERMINES
     (prod_category, prod_category_desc);
```

Alternatively, the *extended_attribute_clause* could have been used instead of the *attribute_clause*, as shown in the following example:

```
CREATE DIMENSION products_dim
    LEVEL product           IS (products.prod_id)
    LEVEL subcategory       IS (products.prod_subcategory)
```

```
LEVEL category          IS (products.prod_category)
HIERARCHY prod_rollup (
     product         CHILD OF
     subcategory     CHILD OF
     category
    )
ATTRIBUTE product_info LEVEL product DETERMINES
 (products.prod_name, products.prod_desc,
  prod_weight_class, prod_unit_of_measure,
  prod_pack_size, prod_status, prod_list_price, prod_min_price)
ATTRIBUTE subcategory DETERMINES
 (prod_subcategory, prod_subcategory_desc)
ATTRIBUTE category DETERMINES
 (prod_category, prod_category_desc);
```

The design, creation, and maintenance of dimensions is part of the design, creation, and maintenance of your data warehouse schema. Once the dimension has been created, check that it meets these requirements:

- There must be a 1:n relationship between a parent and children. A parent can have one or more children, but a child can have only one parent.
- There must be a 1:1 attribute relationship between hierarchy levels and their dependent dimension attributes. For example, if there is a column `fiscal_month_desc`, then a possible attribute relationship would be `fiscal_month_desc` to `fiscal_month_name`.
- If the columns of a parent level and child level are in different relations, then the connection between them also requires a 1:n join relationship. Each row of the child table must join with one and only one row of the parent table. This relationship is stronger than referential integrity alone, because it requires that the child join key must be non-null, that referential integrity must be maintained from the child join key to the parent join key, and that the parent join key must be unique.
- You must ensure (using database constraints if necessary) that the columns of each hierarchy level are non-null and that hierarchical integrity is maintained.
- The hierarchies of a dimension can overlap or be disconnected from each other. However, the columns of a hierarchy level cannot be associated with more than one dimension.
- Join relationships that form cycles in the dimension graph are not supported. For example, a hierarchy level cannot be joined to itself either directly or indirectly.

Note: The information stored with a dimension objects is only declarative. The previously discussed relationships are not enforced with the creation of a dimension object. You should validate any dimension definition with the DBMS_DIMENSION.VALIDATE_DIMENSION procedure, as discussed on "Validating Dimensions" on page 10-12.

Dropping and Creating Attributes with Columns

You can use the attribute clause in a CREATE DIMENSION statement to specify one or multiple columns that are uniquely determined by a hierarchy level.

If you use the *extended_attribute_clause* to create multiple columns determined by a hierarchy level, you can drop one attribute column without dropping them all. Alternatively, you can specify an attribute name for each attribute clause CREATE or ALTER DIMENSION statement so that an attribute name is specified for each attribute clause where multiple level-to-column relationships can be individually specified.

The following statement illustrates how you can drop a single column without dropping all columns:

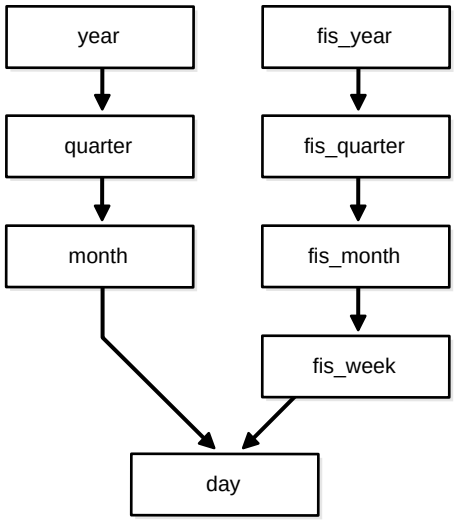
```
CREATE DIMENSION products_dim
LEVEL product         IS (products.prod_id)
LEVEL subcategory     IS (products.prod_subcategory)
LEVEL category        IS (products.prod_category)
HIERARCHY prod_rollup (
          product        CHILD OF
          subcategory    CHILD OF category)
ATTRIBUTE product DETERMINES
         (products.prod_name, products.prod_desc,
          prod_weight_class, prod_unit_of_measure,
          prod_pack_size,prod_status, prod_list_price, prod_min_price)
ATTRIBUTE subcategory_att DETERMINES
         (prod_subcategory, prod_subcategory_desc)
ATTRIBUTE category DETERMINES
         (prod_category, prod_category_desc);

ALTER DIMENSION products_dim
DROP ATTRIBUTE subcategory_att LEVEL subcategory COLUMN prod_subcategory;
```

See Also: *Oracle Database SQL Reference* for a complete description of the CREATE DIMENSION statement

Multiple Hierarchies

A single dimension definition can contain multiple hierarchies. Suppose our retailer wants to track the sales of certain items over time. The first step is to define the time dimension over which sales will be tracked. Figure 10–2 illustrates a dimension `times_dim` with two time hierarchies.

Figure 10–2 times_dim Dimension with Two Time Hierarchies



From the illustration, you can construct the hierarchy of the denormalized `time_dim` dimension's `CREATE DIMENSION` statement as follows. The complete `CREATE DIMENSION` statement as well as the `CREATE TABLE` statement are shown in *Oracle Database Sample Schemas*.

```
CREATE DIMENSION times_dim
   LEVEL day         IS times.time_id
   LEVEL month       IS times.calendar_month_desc
   LEVEL quarter     IS times.calendar_quarter_desc
   LEVEL year        IS times.calendar_year
   LEVEL fis_week    IS times.week_ending_day
   LEVEL fis_month   IS times.fiscal_month_desc
   LEVEL fis_quarter IS times.fiscal_quarter_desc
   LEVEL fis_year    IS times.fiscal_year
   HIERARCHY cal_rollup    (
             day     CHILD OF
```

```
          month   CHILD OF
          quarter CHILD OF
          year
   )
   HIERARCHY fis_rollup    (
          day         CHILD OF
          fis_week    CHILD OF
          fis_month   CHILD OF
          fis_quarter CHILD OF
          fis_year
   ) <attribute determination clauses>;
```

Using Normalized Dimension Tables

The tables used to define a dimension may be normalized or denormalized and the individual hierarchies can be normalized or denormalized. If the levels of a hierarchy come from the same table, it is called a fully denormalized hierarchy. For example, `cal_rollup` in the `times_dim` dimension is a denormalized hierarchy. If levels of a hierarchy come from different tables, such a hierarchy is either a fully or partially normalized hierarchy. This section shows how to define a normalized hierarchy.

Suppose the tracking of a customer's location is done by city, state, and country. This data is stored in the tables customers and countries. The customer dimension `customers_dim` is partially normalized because the data entities `cust_id` and `country_id` are taken from different tables. The clause `JOIN KEY` within the dimension definition specifies how to join together the levels in the hierarchy. The dimension statement is partially shown in the following. The complete `CREATE DIMENSION` statement as well as the `CREATE TABLE` statement are shown in *Oracle Database Sample Schemas*.

```
CREATE DIMENSION customers_dim
   LEVEL customer  IS (customers.cust_id)
   LEVEL city      IS (customers.cust_city)
   LEVEL state     IS (customers.cust_state_province)
   LEVEL country   IS (countries.country_id)
   LEVEL subregion IS (countries.country_subregion)
   LEVEL region IS (countries.country_region)
   HIERARCHY geog_rollup (
      customer        CHILD OF
      city            CHILD OF
      state           CHILD OF
      country         CHILD OF
      subregion       CHILD OF
```

```
    region
  JOIN KEY (customers.country_id) REFERENCES country);
```

Viewing Dimensions

Dimensions can be viewed through one of two methods:

- Using Oracle Enterprise Manager
- Using the DESCRIBE_DIMENSION Procedure

Using Oracle Enterprise Manager

All of the dimensions that exist in the data warehouse can be viewed using Oracle Enterprise Manager. Select the **Dimension** object from within the **Schema** icon to display all of the dimensions. Select a specific dimension to graphically display its hierarchy, levels, and any attributes that have been defined.

> **See Also:** *Oracle Enterprise Manager Administrator's Guide* for details regarding creating and using dimensions

Using the DESCRIBE_DIMENSION Procedure

To view the definition of a dimension, use the DESCRIBE_DIMENSION procedure in the DBMS_DIMENSION package. For example, if a dimension is created in the sh sample schema with the following statements:

```
CREATE DIMENSION channels_dim
        LEVEL channel       IS (channels.channel_id)
        LEVEL channel_class IS (channels.channel_class)
        HIERARCHY channel_rollup (
                channel CHILD OF channel_class)
        ATTRIBUTE channel DETERMINES (channel_desc)
        ATTRIBUTE channel_class DETERMINES (channel_class);
```

Execute the DESCRIBE_DIMENSION procedure as follows:

```
SET SERVEROUTPUT ON FORMAT WRAPPED;  --to improve the display of info
EXECUTE DBMS_DIMENSION.DESCRIBE_DIMENSION('SH.CHANNELS_DIM');
```

You then see the following output results:

```
EXECUTE DBMS_DIMENSION.DESCRIBE_DIMENSION('SH.CHANNELS_DIM');
  DIMENSION SH.CHANNELS_DIM
    LEVEL CHANNEL IS SH.CHANNELS.CHANNEL_ID
```

```
    LEVEL CHANNEL_CLASS IS SH.CHANNELS.CHANNEL_CLASS

    HIERARCHY CHANNEL_ROLLUP (
             CHANNEL CHILD OF
             CHANNEL_CLASS)

    ATTRIBUTE CHANNEL LEVEL CHANNEL DETERMINES
SH.CHANNELS.CHANNEL_DESC
    ATTRIBUTE CHANNEL_CLASS LEVEL CHANNEL_CLASS DETERMINES
SH.CHANNELS.CHANNEL_CLASS
```

Using Dimensions with Constraints

Constraints play an important role with dimensions. Full referential integrity is sometimes enabled in data warehouses, but not always. This is because operational databases normally have full referential integrity and you can ensure that the data flowing into your data warehouse never violates the already established integrity rules.

It is recommended that constraints be enabled and, if validation time is a concern, then the NOVALIDATE clause should be used as follows:

```
ENABLE NOVALIDATE CONSTRAINT pk_time;
```

Primary and foreign keys should be implemented also. Referential integrity constraints and NOT NULL constraints on the fact tables provide information that query rewrite can use to extend the usefulness of materialized views.

In addition, you should use the RELY clause to inform query rewrite that it can rely upon the constraints being correct as follows:

```
ALTER TABLE time MODIFY CONSTRAINT pk_time RELY;
```

This information is also used for query rewrite. See Chapter 18, "Query Rewrite" for more information.

Validating Dimensions

The information of a dimension object is declarative only and not enforced by the database. If the relationships described by the dimensions are incorrect, incorrect results could occur. Therefore, you should verify the relationships specified by CREATE DIMENSION using the DBMS_DIMENSION.VALIDATE_DIMENSION procedure periodically.

This procedure is easy to use and has only four parameters:

- Dimension owner and name.
- Set to TRUE to check only the new rows for tables of this dimension.
- Set to TRUE to verify that all columns are not null.
- A user-supplied unique identifier to identify the result of each run of the procedure.

The following example validates the dimension TIME_FN in the sh schema:

```
@utldim.sql
EXECUTE DBMS_DIMENSION.VALIDATE_DIMENSION ('SH.TIME_FN', FALSE, TRUE,
  'my 1st example');
```

Before running the VALIDATE_DIMENSION procedure, you need to create a local table, DIMENSION_EXCEPTIONS, by running the provided script utldim.sql. If the VALIDATE_DIMENSION procedure encounters any errors, they are placed in this table. Querying this table will identify the exceptions that were found. The following illustrates a sample:

```
SELECT * FROM dimension_exceptions
WHERE statement_id = 'my 1st example';

STATEMENT_ID     OWNER  TABLE_NAME  DIMENSION_NAME   RELATIONSHIP   BAD_ROWID
------------     -----  ----------  --------------   ------------   ---------
my 1st example   SH     MONTH       TIME_FN          FOREIGN KEY    AAAAuwAAJAAAARwAAA
```

However, rather than query this table, it may be better to query the rowid of the invalid row to retrieve the actual row that has violated the constraint. In this example, the dimension TIME_FN is checking a table called month. It has found a row that violates the constraints. Using the rowid, you can see exactly which row in the month table is causing the problem, as in the following:

```
SELECT * FROM month
WHERE rowid IN (SELECT bad_rowid
                FROM dimension_exceptions
                WHERE statement_id = 'my 1st example');

MONTH    QUARTER   FISCAL_QTR    YEAR    FULL_MONTH_NAME    MONTH_NUMB
------   -------   ----------    ----    ---------------    ----------
199903     19981        19981    1998              March             3
```

Altering Dimensions

You can modify the dimension using the ALTER DIMENSION statement. You can add or drop a level, hierarchy, or attribute from the dimension using this command.

Referring to the time dimension in Figure 10–2 on page 10-9, you can remove the attribute fis_year, drop the hierarchy fis_rollup, or remove the level fiscal_year. In addition, you can add a new level called f_year as in the following:

```
ALTER DIMENSION times_dim DROP ATTRIBUTE fis_year;
ALTER DIMENSION times_dim DROP HIERARCHY fis_rollup;
ALTER DIMENSION times_dim DROP LEVEL fis_year
ALTER DIMENSION times_dim ADD LEVEL f_year IS times.fiscal_year;
```

If you used the *extended_attribute_clause* when creating the dimension, you can drop one attribute column without dropping all attribute columns. This is illustrated in "Dropping and Creating Attributes with Columns" on page 10-8, which shows the following statement:

```
ALTER DIMENSION product_dim
DROP ATTRIBUTE size LEVEL prod_type COLUMN Prod_TypeSize;
```

If you try to remove anything with further dependencies inside the dimension, Oracle Database rejects the altering of the dimension. A dimension becomes invalid if you change any schema object that the dimension is referencing. For example, if the table on which the dimension is defined is altered, the dimension becomes invalid.

To check the status of a dimension, view the contents of the column invalid in the ALL_DIMENSIONS data dictionary view. To revalidate the dimension, use the COMPILE option as follows:

```
ALTER DIMENSION times_dim COMPILE;
```

Dimensions can also be modified or deleted using Oracle Enterprise Manager. See *Oracle Enterprise Manager Administrator's Guide* for more information.

Deleting Dimensions

A dimension is removed using the DROP DIMENSION statement. For example:

```
DROP DIMENSION times_dim;
```

Part IV

Managing the Data Warehouse Environment

This section deals with the tasks for managing a data warehouse.

It contains the following chapters:

11

Overview of Extraction, Transformation, and Loading

This chapter discusses the process of extracting, transporting, transforming, and loading data in a data warehousing environment, and includes the following:

Overview of ETL in Data Warehouses

You need to load your data warehouse regularly so that it can serve its purpose of facilitating business analysis. To do this, data from one or more operational systems needs to be extracted and copied into the data warehouse. The process of extracting data from source systems and bringing it into the data warehouse is commonly called **ETL**, which stands for extraction, transformation, and loading. The acronym ETL is perhaps too simplistic, because it omits the transportation phase and implies that each of the other phases of the process is distinct. We refer to the entire process, including data loading, as ETL. You should understand that ETL refers to a broad process, and not three well-defined steps.

The methodology and tasks of ETL have been well known for many years, and are not necessarily unique to data warehouse environments: a wide variety of proprietary applications and database systems are the IT backbone of any enterprise. Data has to be shared between applications or systems, trying to integrate them, giving at least two applications the same picture of the world. This data sharing was mostly addressed by mechanisms similar to what we now call ETL.

Data warehouse environments face the same challenge with the additional burden that they not only have to exchange but to integrate, rearrange and consolidate data over many systems, thereby providing a new unified information base for business intelligence. Additionally, the data volume in data warehouse environments tends to be very large.

What happens during the ETL process? During extraction, the desired data is identified and extracted from many different sources, including database systems and applications. Very often, it is not possible to identify the specific subset of interest, therefore more data than necessary has to be extracted, so the identification of the relevant data will be done at a later point in time. Depending on the source system's capabilities (for example, operating system resources), some transformations may take place during this extraction process. The size of the extracted data varies from hundreds of kilobytes up to gigabytes, depending on the source system and the business situation. The same is true for the time delta between two (logically) identical extractions: the time span may vary between days / hours and minutes to near real-time. Web server log files for example can easily become hundreds of megabytes in a very short period of time.

After extracting data, it has to be physically transported to the target system or an intermediate system for further processing. Depending on the chosen way of transportation, some transformations can be done during this process, too. For

example, a SQL statement which directly accesses a remote target through a gateway can concatenate two columns as part of the SELECT statement.

The emphasis in many of the examples in this section is scalability. Many long-time users of Oracle Database are experts in programming complex data transformation logic using PL/SQL. These chapters suggest alternatives for many such data manipulation operations, with a particular emphasis on implementations that take advantage of Oracle's new SQL functionality, especially for ETL and the parallel query infrastructure.

ETL Tools for Data Warehouses

Designing and maintaining the ETL process is often considered one of the most difficult and resource-intensive portions of a data warehouse project. Many data warehousing projects use ETL tools to manage this process. Oracle Warehouse Builder (OWB), for example, provides ETL capabilities and takes advantage of inherent database abilities. Other data warehouse builders create their own ETL tools and processes, either inside or outside the database.

Besides the support of extraction, transformation, and loading, there are some other tasks that are important for a successful ETL implementation as part of the daily operations of the data warehouse and its support for further enhancements. Besides the support for designing a data warehouse and the data flow, these tasks are typically addressed by ETL tools such as OWB.

Oracle is not an ETL tool and does not provide a complete solution for ETL. However, Oracle does provide a rich set of capabilities that can be used by both ETL tools and customized ETL solutions. Oracle offers techniques for transporting data between Oracle databases, for transforming large volumes of data, and for quickly loading new data into a data warehouse.

Daily Operations in Data Warehouses

The successive loads and transformations must be scheduled and processed in a specific order. Depending on the success or failure of the operation or parts of it, the result must be tracked and subsequent, alternative processes might be started. The control of the progress as well as the definition of a business workflow of the operations are typically addressed by ETL tools such as Oracle Warehouse Builder.

Evolution of the Data Warehouse

As the data warehouse is a living IT system, sources and targets might change. Those changes must be maintained and tracked through the lifespan of the system without overwriting or deleting the old ETL process flow information. To build and keep a level of trust about the information in the warehouse, the process flow of each individual record in the warehouse can be reconstructed at any point in time in the future in an ideal case.

12

Extraction in Data Warehouses

This chapter discusses extraction, which is the process of taking data from an operational system and moving it to your data warehouse or staging system. The chapter discusses:

- Overview of Extraction in Data Warehouses
- Introduction to Extraction Methods in Data Warehouses
- Data Warehousing Extraction Examples

Overview of Extraction in Data Warehouses

Extraction is the operation of extracting data from a source system for further use in a data warehouse environment. This is the first step of the ETL process. After the extraction, this data can be transformed and loaded into the data warehouse.

The source systems for a data warehouse are typically transaction processing applications. For example, one of the source systems for a sales analysis data warehouse might be an order entry system that records all of the current order activities.

Designing and creating the extraction process is often one of the most time-consuming tasks in the ETL process and, indeed, in the entire data warehousing process. The source systems might be very complex and poorly documented, and thus determining which data needs to be extracted can be difficult. The data has to be extracted normally not only once, but several times in a periodic manner to supply all changed data to the data warehouse and keep it up-to-date. Moreover, the source system typically cannot be modified, nor can its performance or availability be adjusted, to accommodate the needs of the data warehouse extraction process.

These are important considerations for extraction and ETL in general. This chapter, however, focuses on the technical considerations of having different kinds of sources and extraction methods. It assumes that the data warehouse team has already identified the data that will be extracted, and discusses common techniques used for extracting data from source databases.

Designing this process means making decisions about the following two main aspects:

- Which extraction method do I choose?

  This influences the source system, the transportation process, and the time needed for refreshing the warehouse.

- How do I provide the extracted data for further processing?

  This influences the transportation method, and the need for cleaning and transforming the data.

Introduction to Extraction Methods in Data Warehouses

The extraction method you should choose is highly dependent on the source system and also from the business needs in the target data warehouse environment. Very often, there is no possibility to add additional logic to the source systems to enhance

an incremental extraction of data due to the performance or the increased workload of these systems. Sometimes even the customer is not allowed to add anything to an out-of-the-box application system.

The estimated amount of the data to be extracted and the stage in the ETL process (initial load or maintenance of data) may also impact the decision of how to extract, from a logical and a physical perspective. Basically, you have to decide how to extract data logically and physically.

Logical Extraction Methods

There are two types of logical extraction:

- Full Extraction
- Incremental Extraction

Full Extraction

The data is extracted completely from the source system. Because this extraction reflects all the data currently available on the source system, there's no need to keep track of changes to the data source since the last successful extraction. The source data will be provided as-is and no additional logical information (for example, timestamps) is necessary on the source site. An example for a full extraction may be an export file of a distinct table or a remote SQL statement scanning the complete source table.

Incremental Extraction

At a specific point in time, only the data that has changed since a well-defined event back in history will be extracted. This event may be the last time of extraction or a more complex business event like the last booking day of a fiscal period. To identify this delta change there must be a possibility to identify all the changed information since this specific time event. This information can be either provided by the source data itself such as an application column, reflecting the last-changed timestamp or a change table where an appropriate additional mechanism keeps track of the changes besides the originating transactions. In most cases, using the latter method means adding extraction logic to the source system.

Many data warehouses do not use any change-capture techniques as part of the extraction process. Instead, entire tables from the source systems are extracted to the data warehouse or staging area, and these tables are compared with a previous extract from the source system to identify the changed data. This approach may not

have significant impact on the source systems, but it clearly can place a considerable burden on the data warehouse processes, particularly if the data volumes are large.

Oracle's Change Data Capture mechanism can extract and maintain such delta information. See Chapter 16, "Change Data Capture" for further details about the Change Data Capture framework.

Physical Extraction Methods

Depending on the chosen logical extraction method and the capabilities and restrictions on the source side, the extracted data can be physically extracted by two mechanisms. The data can either be extracted online from the source system or from an offline structure. Such an offline structure might already exist or it might be generated by an extraction routine.

There are the following methods of physical extraction:

- Online Extraction
- Offline Extraction

Online Extraction

The data is extracted directly from the source system itself. The extraction process can connect directly to the source system to access the source tables themselves or to an intermediate system that stores the data in a preconfigured manner (for example, snapshot logs or change tables). Note that the intermediate system is not necessarily physically different from the source system.

With online extractions, you need to consider whether the distributed transactions are using original source objects or prepared source objects.

Offline Extraction

The data is not extracted directly from the source system but is staged explicitly outside the original source system. The data already has an existing structure (for example, redo logs, archive logs or transportable tablespaces) or was created by an extraction routine.

You should consider the following structures:

- Flat files

  Data in a defined, generic format. Additional information about the source object is necessary for further processing.

- Dump files

Oracle-specific format. Information about the containing objects may or may not be included, depending on the chosen utility.

- Redo and archive logs

  Information is in a special, additional dump file.

- Transportable tablespaces

  A powerful way to extract and move large volumes of data between Oracle databases. A more detailed example of using this feature to extract and transport data is provided in Chapter 13, "Transportation in Data Warehouses". Oracle Corporation recommends that you use transportable tablespaces whenever possible, because they can provide considerable advantages in performance and manageability over other extraction techniques.

  See *Oracle Database Utilities* for more information on using export/import.

Change Data Capture

An important consideration for extraction is incremental extraction, also called Change Data Capture. If a data warehouse extracts data from an operational system on a nightly basis, then the data warehouse requires only the data that has changed since the last extraction (that is, the data that has been modified in the past 24 hours). Change Data Capture is also the key-enabling technology for providing near real-time, or on-time, data warehousing.

When it is possible to efficiently identify and extract only the most recently changed data, the extraction process (as well as all downstream operations in the ETL process) can be much more efficient, because it must extract a much smaller volume of data. Unfortunately, for many source systems, identifying the recently modified data may be difficult or intrusive to the operation of the system. Change Data Capture is typically the most challenging technical issue in data extraction.

Because change data capture is often desirable as part of the extraction process and it might not be possible to use the Change Data Capture mechanism, this section describes several techniques for implementing a self-developed change capture on Oracle Database source systems:

- Timestamps
- Partitioning
- Triggers

These techniques are based upon the characteristics of the source systems, or may require modifications to the source systems. Thus, each of these techniques must be carefully evaluated by the owners of the source system prior to implementation.

Each of these techniques can work in conjunction with the data extraction technique discussed previously. For example, timestamps can be used whether the data is being unloaded to a file or accessed through a distributed query. See Chapter 16, "Change Data Capture" for further details.

Timestamps

The tables in some operational systems have timestamp columns. The timestamp specifies the time and date that a given row was last modified. If the tables in an operational system have columns containing timestamps, then the latest data can easily be identified using the timestamp columns. For example, the following query might be useful for extracting today's data from an `orders` table:

```
SELECT * FROM orders
WHERE TRUNC(CAST(order_date AS date),'dd') =
  TO_DATE(SYSDATE,'dd-mon-yyyy');
```

If the timestamp information is not available in an operational source system, you will not always be able to modify the system to include timestamps. Such modification would require, first, modifying the operational system's tables to include a new timestamp column and then creating a trigger to update the timestamp column following every operation that modifies a given row.

Partitioning

Some source systems might use range partitioning, such that the source tables are partitioned along a date key, which allows for easy identification of new data. For example, if you are extracting from an `orders` table, and the `orders` table is partitioned by week, then it is easy to identify the current week's data.

Triggers

Triggers can be created in operational systems to keep track of recently updated records. They can then be used in conjunction with timestamp columns to identify the exact time and date when a given row was last modified. You do this by creating a trigger on each source table that requires change data capture. Following each DML statement that is executed on the source table, this trigger updates the timestamp column with the current time. Thus, the timestamp column provides the exact time and date when a given row was last modified.

A similar internalized trigger-based technique is used for Oracle materialized view logs. These logs are used by materialized views to identify changed data, and these logs are accessible to end users. However, the format of the materialized view logs is not documented and might change over time.

If you want to use a trigger-based mechanism, use synchronous change data capture. It is recommended that you use synchronous Change Data Capture for trigger based change capture, because CDC provides an externalized interface for accessing the change information and provides a framework for maintaining the distribution of this information to various clients.

Materialized view logs rely on triggers, but they provide an advantage in that the creation and maintenance of this change-data system is largely managed by the database.

However, Oracle Corporation recommends the usage of synchronous Change Data Capture for trigger-based change capture, since CDC provides an externalized interface for accessing the change information and provides a framework for maintaining the distribution of this information to various clients

Trigger-based techniques might affect performance on the source systems, and this impact should be carefully considered prior to implementation on a production source system.

Data Warehousing Extraction Examples

You can extract data in two ways:

- Extraction Using Data Files
- Extraction Through Distributed Operations

Extraction Using Data Files

Most database systems provide mechanisms for exporting or unloading data from the internal database format into flat files. Extracts from mainframe systems often use COBOL programs, but many databases, as well as third-party software vendors, provide export or unload utilities.

Data extraction does not necessarily mean that entire database structures are unloaded in flat files. In many cases, it may be appropriate to unload entire database tables or objects. In other cases, it may be more appropriate to unload only a subset of a given table such as the changes on the source system since the last

extraction or the results of joining multiple tables together. Different extraction techniques vary in their capabilities to support these two scenarios.

When the source system is an Oracle database, several alternatives are available for extracting data into files:

- Extracting into Flat Files Using SQL*Plus
- Extracting into Flat Files Using OCI or Pro*C Programs
- Exporting into Export Files Using the Export Utility
- Extracting into Export Files Using External Tables

Extracting into Flat Files Using SQL*Plus

The most basic technique for extracting data is to execute a SQL query in SQL*Plus and direct the output of the query to a file. For example, to extract a flat file, `country_city.log`, with the pipe sign as delimiter between column values, containing a list of the cities in the US in the tables countries and customers, the following SQL script could be run:

```
SET echo off SET pagesize 0 SPOOL country_city.log
SELECT distinct t1.country_name ||'|'|| t2.cust_city
FROM countries t1, customers t2 WHERE t1.country_id = t2.country_id
AND t1.country_name= 'United States of America';
SPOOL off
```

The exact format of the output file can be specified using SQL*Plus system variables.

This extraction technique offers the advantage of storing the result in a customized format. Note that using the external table data pump unload facility, you can also extract the result of an arbitrary SQL operation. The example previously extracts the results of a join.

This extraction technique can be parallelized by initiating multiple, concurrent SQL*Plus sessions, each session running a separate query representing a different portion of the data to be extracted. For example, suppose that you wish to extract data from an `orders` table, and that the `orders` table has been range partitioned by month, with partitions `orders_jan1998`, `orders_feb1998`, and so on. To extract a single year of data from the `orders` table, you could initiate 12 concurrent SQL*Plus sessions, each extracting a single partition. The SQL script for one such session could be:

```
SPOOL order_jan.dat
SELECT * FROM orders PARTITION (orders_jan1998);
```

```
SPOOL OFF
```

These 12 SQL*Plus processes would concurrently spool data to 12 separate files. You can then concatenate them if necessary (using operating system utilities) following the extraction. If you are planning to use SQL*Loader for loading into the target, these 12 files can be used as is for a parallel load with 12 SQL*Loader sessions. See Chapter 13, "Transportation in Data Warehouses" for an example.

Even if the `orders` table is not partitioned, it is still possible to parallelize the extraction either based on logical or physical criteria. The logical method is based on logical ranges of column values, for example:

```
SELECT ... WHERE order_date
BETWEEN TO_DATE('01-JAN-99') AND TO_DATE('31-JAN-99');
```

The physical method is based on a range of values. By viewing the data dictionary, it is possible to identify the Oracle Database data blocks that make up the `orders` table. Using this information, you could then derive a set of rowid-range queries for extracting data from the `orders` table:

```
SELECT * FROM orders WHERE rowid BETWEEN value1 and value2;
```

Parallelizing the extraction of complex SQL queries is sometimes possible, although the process of breaking a single complex query into multiple components can be challenging. In particular, the coordination of independent processes to guarantee a globally consistent view can be difficult. Unlike the SQL*Plus approach, using the new external table data pump unload functionality provides transparent parallel capabilities.

Note that all parallel techniques can use considerably more CPU and I/O resources on the source system, and the impact on the source system should be evaluated before parallelizing any extraction technique.

Extracting into Flat Files Using OCI or Pro*C Programs

OCI programs (or other programs using Oracle call interfaces, such as Pro*C programs), can also be used to extract data. These techniques typically provide improved performance over the SQL*Plus approach, although they also require additional programming. Like the SQL*Plus approach, an OCI program can extract the results of any SQL query. Furthermore, the parallelization techniques described for the SQL*Plus approach can be readily applied to OCI programs as well.

When using OCI or SQL*Plus for extraction, you need additional information besides the data itself. At minimum, you need information about the extracted

columns. It is also helpful to know the extraction format, which might be the separator between distinct columns.

Exporting into Export Files Using the Export Utility

The Export utility allows tables (including data) to be exported into Oracle Database export files. Unlike the SQL*Plus and OCI approaches, which describe the extraction of the results of a SQL statement, Export provides a mechanism for extracting database objects. Thus, Export differs from the previous approaches in several important ways:

- The export files contain metadata as well as data. An export file contains not only the raw data of a table, but also information on how to re-create the table, potentially including any indexes, constraints, grants, and other attributes associated with that table.
- A single export file may contain a subset of a single object, many database objects, or even an entire schema.
- Export cannot be directly used to export the results of a complex SQL query. Export can be used only to extract subsets of distinct database objects.
- The output of the Export utility must be processed using the Import utility.

Oracle provides the original Export and Import utilities for backward compatibility and the data pump export/import infrastructure for high-performant, scalable and parallel extraction. See *Oracle Database Utilities* for further details.

Extracting into Export Files Using External Tables

In addition to the Export Utility, you can use external tables to extract the results from any SELECT operation. The data is stored in the platform independent, Oracle-internal data pump format and can be processed as regular external table on the target system. The following example extracts the result of a join operation in parallel into the four specified files. The only allowed external table type for extracting data is the Oracle-internal format ORACLE_DATAPUMP.

```
CREATE DIRECTORY def_dir AS '/net/dlsun48/private/hbaer/WORK/FEATURES/et';
DROP TABLE extract_cust;
CREATE TABLE extract_cust
ORGANIZATION EXTERNAL
(TYPE ORACLE_DATAPUMP DEFAULT DIRECTORY def_dir ACCESS PARAMETERS
(NOBADFILE NOLOGFILE)
LOCATION ('extract_cust1.exp', 'extract_cust2.exp', 'extract_cust3.exp',
          'extract_cust4.exp'))
PARALLEL 4 REJECT LIMIT UNLIMITED AS
```

```
SELECT  c.*, co.country_name, co.country_subregion, co.country_region
FROM customers c, countries co where co.country_id=c.country_id;
```

The total number of extraction files specified limits the maximum degree of parallelism for the write operation. Note that the parallelizing of the extraction does not automatically parallelize the SELECT portion of the statement.

Unlike using any kind of export/import, the metadata for the external table is not part of the created files when using the external table data pump unload. To extract the appropriate metadata for the external table, use the DBMS_METADATA package, as illustrated in the following statement:

```
SET LONG 2000
SELECT DBMS_METADATA.GET_DDL('TABLE','EXTRACT_CUST') FROM DUAL;
```

Extraction Through Distributed Operations

Using distributed-query technology, one Oracle database can directly query tables located in various different source systems, such as another Oracle database or a legacy system connected with the Oracle gateway technology. Specifically, a data warehouse or staging database can directly access tables and data located in a connected source system. Gateways are another form of distributed-query technology. Gateways allow an Oracle database (such as a data warehouse) to access database tables stored in remote, non-Oracle databases. This is the simplest method for moving data between two Oracle databases because it combines the extraction and transformation into a single step, and requires minimal programming. However, this is not always feasible.

Suppose that you wanted to extract a list of employee names with department names from a source database and store this data into the data warehouse. Using an Oracle Net connection and distributed-query technology, this can be achieved using a single SQL statement:

```
CREATE TABLE country_city AS SELECT distinct t1.country_name, t2.cust_city
FROM countries@source_db t1, customers@source_db t2
WHERE t1.country_id = t2.country_id
AND t1.country_name='United States of America';
```

This statement creates a local table in a data mart, country_city, and populates it with data from the countries and customers tables on the source system.

This technique is ideal for moving small volumes of data. However, the data is transported from the source system to the data warehouse through a single Oracle Net connection. Thus, the scalability of this technique is limited. For larger data

volumes, file-based data extraction and transportation techniques are often more scalable and thus more appropriate.

See *Oracle Database Heterogeneous Connectivity Administrator's Guide* and *Oracle Database Concepts* for more information on distributed queries.

13

Transportation in Data Warehouses

The following topics provide information about transporting data into a data warehouse:

- Overview of Transportation in Data Warehouses
- Introduction to Transportation Mechanisms in Data Warehouses

Overview of Transportation in Data Warehouses

Transportation is the operation of moving data from one system to another system. In a data warehouse environment, the most common requirements for transportation are in moving data from:

- A source system to a staging database or a data warehouse database
- A staging database to a data warehouse
- A data warehouse to a data mart

Transportation is often one of the simpler portions of the ETL process, and can be integrated with other portions of the process. For example, as shown in Chapter 12, "Extraction in Data Warehouses", distributed query technology provides a mechanism for both extracting and transporting data.

Introduction to Transportation Mechanisms in Data Warehouses

You have three basic choices for transporting data in warehouses:

- Transportation Using Flat Files
- Transportation Through Distributed Operations
- Transportation Using Transportable Tablespaces

Transportation Using Flat Files

The most common method for transporting data is by the transfer of flat files, using mechanisms such as FTP or other remote file system access protocols. Data is unloaded or exported from the source system into flat files using techniques discussed in Chapter 12, "Extraction in Data Warehouses", and is then transported to the target platform using FTP or similar mechanisms.

Because source systems and data warehouses often use different operating systems and database systems, using flat files is often the simplest way to exchange data between heterogeneous systems with minimal transformations. However, even when transporting data between homogeneous systems, flat files are often the most efficient and most easy-to-manage mechanism for data transfer.

Transportation Through Distributed Operations

Distributed queries, either with or without gateways, can be an effective mechanism for extracting data. These mechanisms also transport the data directly to the target

systems, thus providing both extraction and transformation in a single step. Depending on the tolerable impact on time and system resources, these mechanisms can be well suited for both extraction and transformation.

As opposed to flat file transportation, the success or failure of the transportation is recognized immediately with the result of the distributed query or transaction. See Chapter 12, "Extraction in Data Warehouses" for further information.

Transportation Using Transportable Tablespaces

Oracle transportable tablespaces are the fastest way for moving large volumes of data between two Oracle databases. Previous to the introduction of transportable tablespaces, the most scalable data transportation mechanisms relied on moving flat files containing raw data. These mechanisms required that data be unloaded or exported into files from the source database, Then, after transportation, these files were loaded or imported into the target database. Transportable tablespaces entirely bypass the unload and reload steps.

Using transportable tablespaces, Oracle data files (containing table data, indexes, and almost every other Oracle database object) can be directly transported from one database to another. Furthermore, like import and export, transportable tablespaces provide a mechanism for transporting metadata in addition to transporting data.

Transportable tablespaces have some limitations: source and target systems must be running Oracle8*i* (or higher), must use the same character set, and, prior to Oracle Database 10*g*, must run on the same operating system. For details on how to transport tablespace between operating systems, see *Oracle Database Administrator's Guide*.

The most common applications of transportable tablespaces in data warehouses are in moving data from a staging database to a data warehouse, or in moving data from a data warehouse to a data mart.

Transportable Tablespaces Example

Suppose that you have a data warehouse containing sales data, and several data marts that are refreshed monthly. Also suppose that you are going to move one month of sales data from the data warehouse to the data mart.

Step 1 Place the Data to be Transported into its own Tablespace

The current month's data must be placed into a separate tablespace in order to be transported. In this example, you have a tablespace `ts_temp_sales`, which will

hold a copy of the current month's data. Using the CREATE TABLE ... AS SELECT statement, the current month's data can be efficiently copied to this tablespace:

```
CREATE TABLE temp_jan_sales NOLOGGING TABLESPACE ts_temp_sales
AS SELECT * FROM sales
WHERE time_id BETWEEN '31-DEC-1999' AND '01-FEB-2000';
```

Following this operation, the tablespace `ts_temp_sales` is set to read-only:

```
ALTER TABLESPACE ts_temp_sales READ ONLY;
```

A tablespace cannot be transported unless there are no active transactions modifying the tablespace. Setting the tablespace to read-only enforces this.

The tablespace `ts_temp_sales` may be a tablespace that has been especially created to temporarily store data for use by the transportable tablespace features. Following "Copy the Datafiles and Export File to the Target System", this tablespace can be set to read / write, and, if desired, the table `temp_jan_sales` can be dropped, or the tablespace can be re-used for other transportations or for other purposes.

In a given transportable tablespace operation, all of the objects in a given tablespace are transported. Although only one table is being transported in this example, the tablespace `ts_temp_sales` could contain multiple tables. For example, perhaps the data mart is refreshed not only with the new month's worth of sales transactions, but also with a new copy of the customer table. Both of these tables could be transported in the same tablespace. Moreover, this tablespace could also contain other database objects such as indexes, which would also be transported.

Additionally, in a given transportable-tablespace operation, multiple tablespaces can be transported at the same time. This makes it easier to move very large volumes of data between databases. Note, however, that the transportable tablespace feature can only transport a set of tablespaces which contain a complete set of database objects without dependencies on other tablespaces. For example, an index cannot be transported without its table, nor can a partition be transported without the rest of the table. You can use the DBMS_TTS package to check that a tablespace is transportable.

> **See Also:** *PL/SQL Packages and Types Reference* for detailed information about the DBMS_TTS package

In this step, we have copied the January sales data into a separate tablespace; however, in some cases, it may be possible to leverage the transportable tablespace feature without even moving data to a separate tablespace. If the sales table has

been partitioned by month in the data warehouse and if each partition is in its own tablespace, then it may be possible to directly transport the tablespace containing the January data. Suppose the January partition, sales_jan2000, is located in the tablespace ts_sales_jan2000. Then the tablespace ts_sales_jan2000 could potentially be transported, rather than creating a temporary copy of the January sales data in the ts_temp_sales.

However, the same conditions must be satisfied in order to transport the tablespace ts_sales_jan2000 as are required for the specially created tablespace. First, this tablespace must be set to READ ONLY. Second, because a single partition of a partitioned table cannot be transported without the remainder of the partitioned table also being transported, it is necessary to exchange the January partition into a separate table (using the ALTER TABLE statement) to transport the January data. The EXCHANGE operation is very quick, but the January data will no longer be a part of the underlying sales table, and thus may be unavailable to users until this data is exchanged back into the sales table after the export of the metadata. The January data can be exchanged back into the sales table after you complete step 3.

Step 2 Export the Metadata

The Export utility is used to export the metadata describing the objects contained in the transported tablespace. For our example scenario, the Export command could be:

```
EXP TRANSPORT_TABLESPACE=y TABLESPACES=ts_temp_sales FILE=jan_sales.dmp
```

This operation will generate an export file, jan_sales.dmp. The export file will be small, because it contains only metadata. In this case, the export file will contain information describing the table temp_jan_sales, such as the column names, column datatype, and all other information that the target Oracle database will need in order to access the objects in ts_temp_sales.

Step 3 Copy the Datafiles and Export File to the Target System

Copy the data files that make up ts_temp_sales, as well as the export file jan_sales.dmp to the data mart platform, using any transportation mechanism for flat files. Once the datafiles have been copied, the tablespace ts_temp_sales can be set to READ WRITE mode if desired.

Step 4 Import the Metadata

Once the files have been copied to the data mart, the metadata should be imported into the data mart:

```
IMP TRANSPORT_TABLESPACE=y DATAFILES='/db/tempjan.f'
```

```
    TABLESPACES=ts_temp_sales FILE=jan_sales.dmp
```

At this point, the tablespace `ts_temp_sales` and the table `temp_sales_jan` are accessible in the data mart. You can incorporate this new data into the data mart's tables.

You can insert the data from the `temp_sales_jan` table into the data mart's sales table in one of two ways:

```
INSERT /*+ APPEND */ INTO sales SELECT * FROM temp_sales_jan;
```

Following this operation, you can delete the `temp_sales_jan` table (and even the entire `ts_temp_sales` tablespace).

Alternatively, if the data mart's sales table is partitioned by month, then the new transported tablespace and the `temp_sales_jan` table can become a permanent part of the data mart. The `temp_sales_jan` table can become a partition of the data mart's sales table:

```
ALTER TABLE sales ADD PARTITION sales_00jan VALUES
  LESS THAN (TO_DATE('01-feb-2000','dd-mon-yyyy'));
ALTER TABLE sales EXCHANGE PARTITION sales_00jan
  WITH TABLE temp_sales_jan INCLUDING INDEXES WITH VALIDATION;
```

Other Uses of Transportable Tablespaces

The previous example illustrates a typical scenario for transporting data in a data warehouse. However, transportable tablespaces can be used for many other purposes. In a data warehousing environment, transportable tablespaces should be viewed as a utility (much like Import/Export or SQL*Loader), whose purpose is to move large volumes of data between Oracle databases. When used in conjunction with parallel data movement operations such as the CREATE TABLE ... AS SELECT and INSERT ... AS SELECT statements, transportable tablespaces provide an important mechanism for quickly transporting data for many purposes.

14

Loading and Transformation

This chapter helps you create and manage a data warehouse, and discusses:

Overview of Loading and Transformation in Data Warehouses

Data transformations are often the most complex and, in terms of processing time, the most costly part of the extraction, transformation, and loading (ETL) process. They can range from simple data conversions to extremely complex data scrubbing techniques. Many, if not all, data transformations can occur within an Oracle database, although transformations are often implemented outside of the database (for example, on flat files) as well.

This chapter introduces techniques for implementing scalable and efficient data transformations within the Oracle Database. The examples in this chapter are relatively simple. Real-world data transformations are often considerably more complex. However, the transformation techniques introduced in this chapter meet the majority of real-world data transformation requirements, often with more scalability and less programming than alternative approaches.

This chapter does not seek to illustrate all of the typical transformations that would be encountered in a data warehouse, but to demonstrate the types of fundamental technology that can be applied to implement these transformations and to provide guidance in how to choose the best techniques.

Transformation Flow

From an architectural perspective, you can transform your data in two ways:

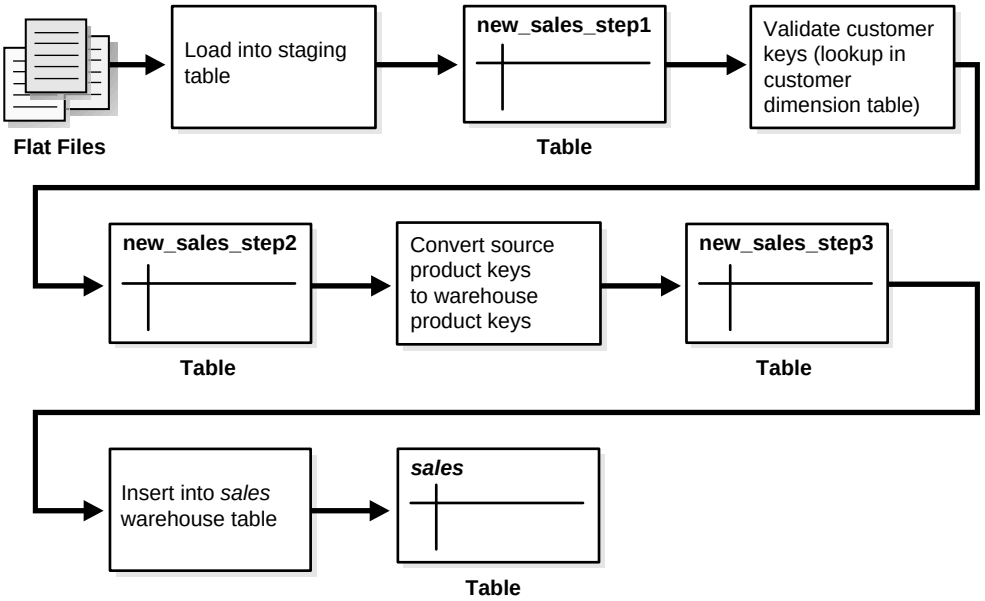
- Multistage Data Transformation
- Pipelined Data Transformation

Multistage Data Transformation

The data transformation logic for most data warehouses consists of multiple steps. For example, in transforming new records to be inserted into a sales table, there may be separate logical transformation steps to validate each dimension key.

Figure 14–1 offers a graphical way of looking at the transformation logic.

Figure 14–1 Multistage Data Transformation



When using Oracle Database as a transformation engine, a common strategy is to implement each transformation as a separate SQL operation and to create a separate, temporary staging table (such as the tables `new_sales_step1` and `new_sales_step2` in Figure 14–1) to store the incremental results for each step. This load-then-transform strategy also provides a natural checkpointing scheme to the entire transformation process, which enables to the process to be more easily monitored and restarted. However, a disadvantage to multistaging is that the space and time requirements increase.

It may also be possible to combine many simple logical transformations into a single SQL statement or single PL/SQL procedure. Doing so may provide better performance than performing each step independently, but it may also introduce difficulties in modifying, adding, or dropping individual transformations, as well as recovering from failed transformations.

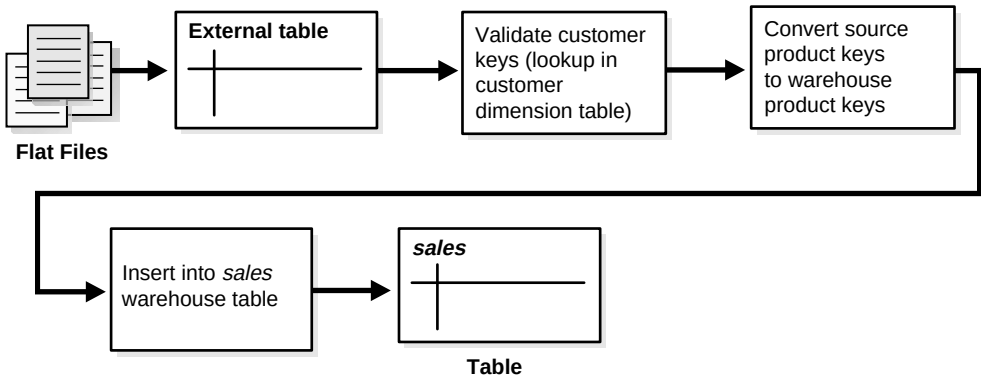
Pipelined Data Transformation

The ETL process flow can be changed dramatically and the database becomes an integral part of the ETL solution.

The new functionality renders some of the former necessary process steps obsolete while some others can be remodeled to enhance the data flow and the data transformation to become more scalable and non-interruptive. The task shifts from serial transform-then-load process (with most of the tasks done outside the database) or load-then-transform process, to an enhanced transform-while-loading.

Oracle offers a wide variety of new capabilities to address all the issues and tasks relevant in an ETL scenario. It is important to understand that the database offers toolkit functionality rather than trying to address a one-size-fits-all solution. The underlying database has to enable the most appropriate ETL process flow for a specific customer need, and not dictate or constrain it from a technical perspective. Figure 14–2 illustrates the new functionality, which is discussed throughout later sections.

Figure 14–2 Pipelined Data Transformation



Loading Mechanisms

You can use the following mechanisms for loading a data warehouse:

- Loading a Data Warehouse with SQL*Loader
- Loading a Data Warehouse with External Tables
- Loading a Data Warehouse with OCI and Direct-Path APIs
- Loading a Data Warehouse with Export/Import

Loading a Data Warehouse with SQL*Loader

Before any data transformations can occur within the database, the raw data must become accessible for the database. One approach is to load it into the database. Chapter 13, "Transportation in Data Warehouses", discusses several techniques for transporting data to an Oracle data warehouse. Perhaps the most common technique for transporting data is by way of flat files.

SQL*Loader is used to move data from flat files into an Oracle data warehouse. During this data load, SQL*Loader can also be used to implement basic data transformations. When using direct-path SQL*Loader, basic data manipulation, such as datatype conversion and simple `NULL` handling, can be automatically resolved during the data load. Most data warehouses use direct-path loading for performance reasons.

The conventional-path loader provides broader capabilities for data transformation than a direct-path loader: SQL functions can be applied to any column as those values are being loaded. This provides a rich capability for transformations during the data load. However, the conventional-path loader is slower than direct-path loader. For these reasons, the conventional-path loader should be considered primarily for loading and transforming smaller amounts of data.

The following is a simple example of a SQL*Loader controlfile to load data into the `sales` table of the `sh` sample schema from an external file `sh_sales.dat`. The external flat file `sh_sales.dat` consists of sales transaction data, aggregated on a daily level. Not all columns of this external file are loaded into `sales`. This external file will also be used as source for loading the second fact table of the `sh` sample schema, which is done using an external table:

The following shows the controlfile (`sh_sales.ctl`) to load the `sales` table:

```
LOAD DATA INFILE sh_sales.dat APPEND INTO TABLE sales
FIELDS TERMINATED BY "|"
(PROD_ID, CUST_ID, TIME_ID, CHANNEL_ID, PROMO_ID, QUANTITY_SOLD, AMOUNT_SOLD)
```

It can be loaded with the following command:

```
$  sqlldr sh/sh control=sh_sales.ctl direct=true
```

Loading a Data Warehouse with External Tables

Another approach for handling external data sources is using external tables. Oracle's external table feature enables you to use external data as a virtual table that can be queried and joined directly and in parallel without requiring the external

data to be first loaded in the database. You can then use SQL, PL/SQL, and Java to access the external data.

External tables enable the pipelining of the loading phase with the transformation phase. The transformation process can be merged with the loading process without any interruption of the data streaming. It is no longer necessary to stage the data inside the database for further processing inside the database, such as comparison or transformation. For example, the conversion functionality of a conventional load can be used for a direct-path `INSERT AS SELECT` statement in conjunction with the `SELECT` from an external table.

The main difference between external tables and regular tables is that externally organized tables are read-only. No DML operations (`UPDATE`/`INSERT`/`DELETE`) are possible and no indexes can be created on them.

External tables are a mostly compliant to the existing SQL*Loader functionality and provide superior functionality in most cases. External tables are especially useful for environments where the complete external source has to be joined with existing database objects or when the data has to be transformed in a complex manner. For example, unlike SQL*Loader, you can apply any arbitrary SQL transformation and use the direct path insert method.

You can create an external table named `sales_transactions_ext`, representing the structure of the complete sales transaction data, represented in the external file `sh_sales.dat`. The product department is especially interested in a cost analysis on product and time. We thus create a fact table named `cost` in the `sales history` schema. The operational source data is the same as for the `sales` fact table. However, because we are not investigating every dimensional information that is provided, the data in the cost fact table has a coarser granularity than in the sales fact table, for example, all different distribution channels are aggregated.

We cannot load the data into the cost fact table without applying the previously mentioned aggregation of the detailed information, due to the suppression of some of the dimensions.

The external table framework offers a solution to solve this. Unlike SQL*Loader, where you would have to load the data before applying the aggregation, you can combine the loading and transformation within a single SQL DML statement, as shown in the following. You do not have to stage the data temporarily before inserting into the target table.

The object directories must already exist, and point to the directory containing the `sh_sales.dat` file as well as the directory containing the bad and log files.

```
CREATE TABLE sales_transactions_ext
```

```
(PROD_ID NUMBER, CUST_ID NUMBER,
 TIME_ID DATE, CHANNEL_ID NUMBER,
 PROMO_ID NUMBER, QUANTITY_SOLD NUMBER,
 AMOUNT_SOLD NUMBER(10,2), UNIT_COST NUMBER(10,2),
 UNIT_PRICE NUMBER(10,2))
ORGANIZATION external (TYPE oracle_loader
  DEFAULT DIRECTORY data_file_dir ACCESS PARAMETERS
  (RECORDS DELIMITED BY NEWLINE CHARACTERSET US7ASCII
    BADFILE log_file_dir:'sh_sales.bad_xt'
    LOGFILE log_file_dir:'sh_sales.log_xt'
    FIELDS TERMINATED BY "|" LDRTRIM
    ( PROD_ID, CUST_ID,
      TIME_ID         DATE(10) "YYYY-MM-DD",
      CHANNEL_ID, PROMO_ID, QUANTITY_SOLD, AMOUNT_SOLD,
      UNIT_COST, UNIT_PRICE))
  location ('sh_sales.dat')
)REJECT LIMIT UNLIMITED;
```

The external table can now be used from within the database, accessing some columns of the external data only, grouping the data, and inserting it into the `costs` fact table:

```
INSERT /*+ APPEND */ INTO COSTS
(TIME_ID, PROD_ID, UNIT_COST, UNIT_PRICE)
SELECT TIME_ID, PROD_ID, AVG(UNIT_COST), AVG(amount_sold/quantity_sold)
FROM sales_transactions_ext GROUP BY time_id, prod_id;
```

> **See Also:** *Oracle Database SQL Reference* for a complete description of external table syntax and restrictions and *Oracle Database Utilities* for usage examples

Loading a Data Warehouse with OCI and Direct-Path APIs

OCI and direct-path APIs are frequently used when the transformation and computation are done outside the database and there is no need for flat file staging.

Loading a Data Warehouse with Export/Import

Export and import are used when the data is inserted as is into the target system. No complex extractions are possible. See Chapter 12, "Extraction in Data Warehouses" for further information.

Transformation Mechanisms

You have the following choices for transforming data inside the database:

- Transformation Using SQL
- Transformation Using PL/SQL
- Transformation Using Table Functions

Transformation Using SQL

Once data is loaded into the database, data transformations can be executed using SQL operations. There are four basic techniques for implementing SQL data transformations:

- CREATE TABLE ... AS SELECT And INSERT /*+APPEND*/ AS SELECT
- Transformation Using UPDATE
- Transformation Using MERGE
- Transformation Using Multitable INSERT

CREATE TABLE ... AS SELECT And INSERT /*+APPEND*/ AS SELECT

The `CREATE TABLE ... AS SELECT` statement (CTAS) is a powerful tool for manipulating large sets of data. As shown in the following example, many data transformations can be expressed in standard SQL, and CTAS provides a mechanism for efficiently executing a SQL query and storing the results of that query in a new database table. The `INSERT /*+APPEND*/ ... AS SELECT` statement offers the same capabilities with existing database tables.

In a data warehouse environment, CTAS is typically run in parallel using `NOLOGGING` mode for best performance.

A simple and common type of data transformation is data substitution. In a data substitution transformation, some or all of the values of a single column are modified. For example, our `sales` table has a `channel_id` column. This column indicates whether a given sales transaction was made by a company's own sales force (a direct sale) or by a distributor (an indirect sale).

You may receive data from multiple source systems for your data warehouse. Suppose that one of those source systems processes only direct sales, and thus the source system does not know indirect sales channels. When the data warehouse initially receives sales data from this system, all sales records have a `NULL` value for the `sales.channel_id` field. These `NULL` values must be set to the proper key

value. For example, you can do this efficiently using a SQL function as part of the insertion into the target sales table statement:

The structure of source table sales_activity_direct is as follows:

```
DESC sales_activity_direct
Name            Null?    Type
------------    -----    ----------------
SALES_DATE               DATE
PRODUCT_ID               NUMBER
CUSTOMER_ID              NUMBER
PROMOTION_ID             NUMBER
AMOUNT                   NUMBER
QUANTITY                 NUMBER

INSERT /*+ APPEND NOLOGGING PARALLEL */
INTO sales SELECT product_id, customer_id, TRUNC(sales_date), 3,
       promotion_id, quantity, amount
FROM sales_activity_direct;
```

Transformation Using UPDATE

Another technique for implementing a data substitution is to use an UPDATE statement to modify the sales.channel_id column. An UPDATE will provide the correct result. However, if the data substitution transformations require that a very large percentage of the rows (or all of the rows) be modified, then, it may be more efficient to use a CTAS statement than an UPDATE.

Transformation Using MERGE

Oracle Database's merge functionality extends SQL, by introducing the SQL keyword MERGE, in order to provide the ability to update or insert a row conditionally into a table or out of line single table views. Conditions are specified in the ON clause. This is, besides pure bulk loading, one of the most common operations in data warehouse synchronization.

Merge Examples The following discusses various implementations of a merge. The examples assume that new data for the dimension table products is propagated to the data warehouse and has to be either inserted or updated. The table products_delta has the same structure as products.

Example 14–1 Merge Operation Using SQL

```
MERGE INTO products t USING products_delta s
ON (t.prod_id=s.prod_id)
```

```
WHEN MATCHED THEN UPDATE SET
  t.prod_list_price=s.prod_list_price, t.prod_min_price=s.prod_min_price
WHEN NOT MATCHED THEN INSERT (prod_id, prod_name, prod_desc, prod_subcategory,
  prod_subcategory_desc, prod_category, prod_category_desc, prod_status,
  prod_list_price, prod_min_price)
VALUES (s.prod_id, s.prod_name, s.prod_desc, s.prod_subcategory,
  s.prod_subcategory_desc, s.prod_category, s.prod_category_desc,
  s.prod_status, s.prod_list_price, s.prod_min_price);
```

Transformation Using Multitable INSERT

Many times, external data sources have to be segregated based on logical attributes for insertion into different target objects. It's also frequent in data warehouse environments to fan out the same source data into several target objects. Multitable inserts provide a new SQL statement for these kinds of transformations, where data can either end up in several or exactly one target, depending on the business transformation rules. This insertion can be done conditionally based on business rules or unconditionally.

It offers the benefits of the INSERT ... SELECT statement when multiple tables are involved as targets. In doing so, it avoids the drawbacks of the two obvious alternatives. You either had to deal with *n* independent INSERT ... SELECT statements, thus processing the same source data *n* times and increasing the transformation workload *n* times. Alternatively, you had to choose a procedural approach with a per-row determination how to handle the insertion. This solution lacked direct access to high-speed access paths available in SQL.

As with the existing INSERT ... SELECT statement, the new statement can be parallelized and used with the direct-load mechanism for faster performance.

Example 14–2 Unconditional Insert

The following statement aggregates the transactional sales information, stored in sales_activity_direct, on a per daily base and inserts into both the sales and the costs fact table for the current day.

```
INSERT ALL
   INTO sales VALUES (product_id, customer_id, today, 3, promotion_id,
                      quantity_per_day, amount_per_day)
   INTO costs VALUES (product_id, today, promotion_id, 3,
                      product_cost, product_price)
SELECT TRUNC(s.sales_date) AS today, s.product_id, s.customer_id,
  s.promotion_id, SUM(s.amount) AS amount_per_day, SUM(s.quantity)
  quantity_per_day, p.prod_min_price*0.8 AS product_cost, p.prod_list_price
  AS product_price
```

```
FROM sales_activity_direct s, products p
WHERE s.product_id = p.prod_id AND TRUNC(sales_date) = TRUNC(SYSDATE)
GROUP BY TRUNC(sales_date), s.product_id, s.customer_id, s.promotion_id,
  p.prod_min_price*0.8, p.prod_list_price;
```

Example 14–3 Conditional ALL Insert

The following statement inserts a row into the `sales` and `costs` tables for all sales transactions with a valid promotion and stores the information about multiple identical orders of a customer in a separate table `cum_sales_activity`. It is possible two rows will be inserted for some sales transactions, and none for others.

```
INSERT ALL
WHEN promotion_id IN (SELECT promo_id FROM promotions) THEN
   INTO sales VALUES (product_id, customer_id, today, 3, promotion_id,
                      quantity_per_day, amount_per_day)
   INTO costs VALUES (product_id, today, promotion_id, 3,
                      product_cost, product_price)
WHEN num_of_orders > 1 THEN
   INTO cum_sales_activity VALUES (today, product_id, customer_id,
     promotion_id, quantity_per_day, amount_per_day, num_of_orders)
SELECT TRUNC(s.sales_date) AS today, s.product_id, s.customer_id,
   s.promotion_id, SUM(s.amount) AS amount_per_day, SUM(s.quantity)
   quantity_per_day, COUNT(*) num_of_orders, p.prod_min_price*0.8
   AS product_cost, p.prod_list_price as product_price
FROM sales_activity_direct s, products p
WHERE s.product_id = p.prod_id
AND TRUNC(sales_date) = TRUNC(SYSDATE)
GROUP BY TRUNC(sales_date), s.product_id, s.customer_id,
 s.promotion_id, p.prod_min_price*0.8, p.prod_list_price;
```

Example 14–4 Conditional FIRST Insert

The following statement inserts into an appropriate shipping manifest according to the total quantity and the weight of a product order. An exception is made for high value orders, which are also sent by express, unless their weight classification is not too high. It assumes the existence of appropriate tables `large_freight_shipping`, `express_shipping`, and `default_shipping`.

```
INSERT FIRST WHEN (sum_quantity_sold > 10 AND prod_weight_class < 5) OR
      (sum_quantity_sold > 5 AND prod_weight_class > 5) THEN
  INTO large_freight_shipping VALUES
      (time_id, cust_id, prod_id, prod_weight_class, sum_quantity_sold)
  WHEN sum_amount_sold > 1000 THEN
  INTO express_shipping VALUES
```

```
      (time_id, cust_id, prod_id, prod_weight_class,
       sum_amount_sold, sum_quantity_sold)
  ELSE INTO default_shipping VALUES
      (time_id, cust_id, prod_id, sum_quantity_sold)
SELECT s.time_id, s.cust_id, s.prod_id, p.prod_weight_class,
       SUM(amount_sold) AS sum_amount_sold,
       SUM(quantity_sold) AS sum_quantity_sold
FROM sales s, products p
WHERE s.prod_id = p.prod_id AND s.time_id = TRUNC(SYSDATE)
GROUP BY s.time_id, s.cust_id, s.prod_id, p.prod_weight_class;
```

Example 14–5 Mixed Conditional and Unconditional Insert

The following example inserts new customers into the `customers` table and stores all new customers with `cust_credit_limit` higher then 4500 in an additional, separate table for further promotions.

```
INSERT FIRST WHEN cust_credit_limit >= 4500 THEN INTO customers
  INTO customers_special VALUES (cust_id, cust_credit_limit)
  ELSE INTO customers
SELECT * FROM customers_new;
```

See Chapter 15, "Maintaining the Data Warehouse" for more information regarding `MERGE` operations.

Transformation Using PL/SQL

In a data warehouse environment, you can use procedural languages such as PL/SQL to implement complex transformations in the Oracle Database. Whereas CTAS operates on entire tables and emphasizes parallelism, PL/SQL provides a row-based approached and can accommodate very sophisticated transformation rules. For example, a PL/SQL procedure could open multiple cursors and read data from multiple source tables, combine this data using complex business rules, and finally insert the transformed data into one or more target table. It would be difficult or impossible to express the same sequence of operations using standard SQL statements.

Using a procedural language, a specific transformation (or number of transformation steps) within a complex ETL processing can be encapsulated, reading data from an intermediate staging area and generating a new table object as output. A previously generated transformation input table and a subsequent transformation will consume the table generated by this specific transformation. Alternatively, these encapsulated transformation steps within the complete ETL process can be integrated seamlessly, thus streaming sets of rows between each

other without the necessity of intermediate staging. You can use table functions to implement such behavior.

Transformation Using Table Functions

Table functions provide the support for pipelined and parallel execution of transformations implemented in PL/SQL, C, or Java. Scenarios as mentioned earlier can be done without requiring the use of intermediate staging tables, which interrupt the data flow through various transformations steps.

What is a Table Function?

A table function is defined as a function that can produce a set of rows as output. Additionally, table functions can take a set of rows as input. Prior to Oracle9*i*, PL/SQL functions:

- Could not take cursors as input.
- Could not be parallelized or pipelined.

Now, functions are not limited in these ways. Table functions extend database functionality by allowing:

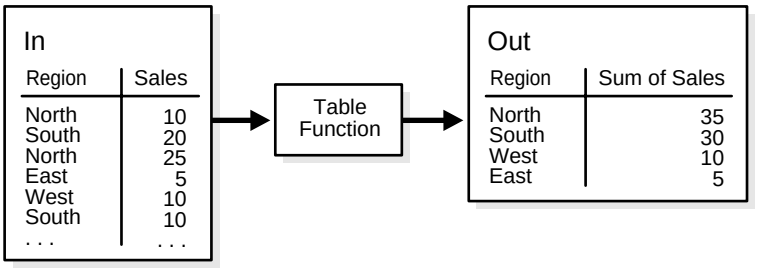
- Multiple rows to be returned from a function.
- Results of SQL subqueries (that select multiple rows) to be passed directly to functions.
- Functions take cursors as input.
- Functions can be parallelized.
- Returning result sets incrementally for further processing as soon as they are created. This is called incremental pipelining

Table functions can be defined in PL/SQL using a native PL/SQL interface, or in Java or C using the Oracle Data Cartridge Interface (ODCI).

> **See Also:** *PL/SQL User's Guide and Reference* for further information and *Oracle Data Cartridge Developer's Guide*

Figure 14–3 illustrates a typical aggregation where you input a set of rows and output a set of rows, in that case, after performing a SUM operation.

Figure 14–3 Table Function Example



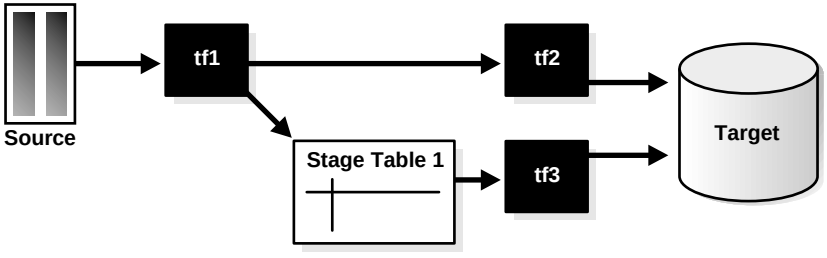
The pseudocode for this operation would be similar to:

```
INSERT INTO Out SELECT * FROM ("Table Function"(SELECT * FROM In));
```

The table function takes the result of the `SELECT` on `In` as input and delivers a set of records in a different format as output for a direct insertion into `Out`.

Additionally, a table function can fan out data within the scope of an atomic transaction. This can be used for many occasions like an efficient logging mechanism or a fan out for other independent transformations. In such a scenario, a single staging table will be needed.

Figure 14–4 Pipelined Parallel Transformation with Fanout



The pseudocode for this would be similar to:

```
INSERT INTO target SELECT * FROM (tf2(SELECT *
FROM (tf1(SELECT * FROM source))));
```

This will insert into `target` and, as part of `tf1`, into `Stage Table 1` within the scope of an atomic transaction.

```
INSERT INTO target SELECT * FROM tf3(SELT * FROM stage_table1);
```

Example 14–6 Table Functions Fundamentals

The following examples demonstrate the fundamentals of table functions, without the usage of complex business rules implemented inside those functions. They are chosen for demonstration purposes only, and are all implemented in PL/SQL.

Table functions return sets of records and can take cursors as input. Besides the sh sample schema, you have to set up the following database objects before using the examples:

```
CREATE TYPE product_t AS OBJECT (
      prod_id                    NUMBER(6)
    , prod_name                  VARCHAR2(50)
    , prod_desc                  VARCHAR2(4000)
    , prod_subcategory           VARCHAR2(50)
    , prod_subcategory_desc      VARCHAR2(2000)
    , prod_category              VARCHAR2(50)
    , prod_category_desc         VARCHAR2(2000)
    , prod_weight_class          NUMBER(2)
    , prod_unit_of_measure       VARCHAR2(20)
    , prod_pack_size             VARCHAR2(30)
    , supplier_id                NUMBER(6)
    , prod_status                VARCHAR2(20)
    , prod_list_price            NUMBER(8,2)
    , prod_min_price             NUMBER(8,2)
);
/
CREATE TYPE product_t_table AS TABLE OF product_t;
/
COMMIT;

CREATE OR REPLACE PACKAGE cursor_PKG AS
  TYPE product_t_rec IS RECORD (
      prod_id                     NUMBER(6)
    , prod_name                   VARCHAR2(50)
    , prod_desc                   VARCHAR2(4000)
    , prod_subcategory            VARCHAR2(50)
    , prod_subcategory_desc       VARCHAR2(2000)
    , prod_category               VARCHAR2(50)
    , prod_category_desc          VARCHAR2(2000)
    , prod_weight_class           NUMBER(2)
    , prod_unit_of_measure        VARCHAR2(20)
    , prod_pack_size              VARCHAR2(30)
    , supplier_id                 NUMBER(6)
    , prod_status                 VARCHAR2(20)
    , prod_list_price             NUMBER(8,2)
```

```
    , prod_min_price            NUMBER(8,2));
  TYPE product_t_rectab IS TABLE OF product_t_rec;
  TYPE strong_refcur_t IS REF CURSOR RETURN product_t_rec;
  TYPE refcur_t IS REF CURSOR;
END;
/

REM artificial help table, used later
CREATE TABLE obsolete_products_errors (prod_id NUMBER, msg VARCHAR2(2000));
```

The following example demonstrates a simple filtering; it shows all obsolete products except the prod_category Electronics. The table function returns the result set as a set of records and uses a weakly typed ref cursor as input.

```
CREATE OR REPLACE FUNCTION obsolete_products(cur cursor_pkg.refcur_t)
RETURN product_t_table
IS
    prod_id                    NUMBER(6);
    prod_name                  VARCHAR2(50);
    prod_desc                  VARCHAR2(4000);
    prod_subcategory           VARCHAR2(50);
    prod_subcategory_desc      VARCHAR2(2000);
    prod_category              VARCHAR2(50);
    prod_category_desc         VARCHAR2(2000);
    prod_weight_class          NUMBER(2);
    prod_unit_of_measure       VARCHAR2(20);
    prod_pack_size             VARCHAR2(30);
    supplier_id                NUMBER(6);
    prod_status                VARCHAR2(20);
    prod_list_price            NUMBER(8,2);
    prod_min_price             NUMBER(8,2);
    sales NUMBER:=0;
    objset product_t_table := product_t_table();
    i NUMBER := 0;
BEGIN
   LOOP
     -- Fetch from cursor variable
     FETCH cur INTO prod_id, prod_name, prod_desc, prod_subcategory,
    prod_subcategory_desc, prod_category, prod_category_desc, prod_weight_class,
    prod_unit_of_measure, prod_pack_size, supplier_id, prod_status,
    prod_list_price, prod_min_price;
     EXIT WHEN cur%NOTFOUND; -- exit when last row is fetched
     -- Category Electronics is not meant to be obsolete and will be suppressed
     IF prod_status='obsolete' AND prod_category != 'Electronics' THEN
     -- append to collection
```

```
     i:=i+1;
     objset.extend;
     objset(i):=product_t( prod_id, prod_name, prod_desc, prod_subcategory,
     prod_subcategory_desc, prod_category, prod_category_desc,
     prod_weight_class, prod_unit_of_measure, prod_pack_size, supplier_id,
     prod_status, prod_list_price, prod_min_price);
     END IF;
   END LOOP;
   CLOSE cur;
   RETURN objset;
END;
/
```

You can use the table function in a SQL statement to show the results. Here we use additional SQL functionality for the output:

```
SELECT DISTINCT UPPER(prod_category), prod_status
FROM TABLE(obsolete_products(
   CURSOR(SELECT prod_id, prod_name, prod_desc, prod_subcategory,
   prod_subcategory_desc, prod_category, prod_category_desc, prod_weight_class,
   prod_unit_of_measure, prod_pack_size,
   supplier_id, prod_status, prod_list_price, prod_min_price
          FROM products)));
```

The following example implements the same filtering than the first one. The main differences between those two are:

- This example uses a strong typed REF cursor as input and can be parallelized based on the objects of the strong typed cursor, as shown in one of the following examples.
- The table function returns the result set incrementally as soon as records are created.

```
CREATE OR REPLACE FUNCTION
  obsolete_products_pipe(cur cursor_pkg.strong_refcur_t) RETURN product_t_table
PIPELINED
PARALLEL_ENABLE (PARTITION cur BY ANY) IS
    prod_id                 NUMBER(6);
    prod_name               VARCHAR2(50);
    prod_desc               VARCHAR2(4000);
    prod_subcategory        VARCHAR2(50);
    prod_subcategory_desc   VARCHAR2(2000);
    prod_category           VARCHAR2(50);
    prod_category_desc      VARCHAR2(2000);
    prod_weight_class       NUMBER(2);
```

```
    prod_unit_of_measure   VARCHAR2(20);
    prod_pack_size         VARCHAR2(30);
    supplier_id            NUMBER(6);
    prod_status            VARCHAR2(20);
    prod_list_price        NUMBER(8,2);
    prod_min_price         NUMBER(8,2);
    sales NUMBER:=0;
BEGIN
 LOOP
     -- Fetch from cursor variable
     FETCH cur INTO prod_id, prod_name, prod_desc, prod_subcategory,
     prod_subcategory_desc, prod_category, prod_category_desc,
     prod_weight_class, prod_unit_of_measure, prod_pack_size, supplier_id,
     prod_status, prod_list_price, prod_min_price;
     EXIT WHEN cur%NOTFOUND; -- exit when last row is fetched
     IF prod_status='obsolete' AND prod_category !='Electronics' THEN
       PIPE ROW (product_t( prod_id, prod_name, prod_desc, prod_subcategory,
 prod_subcategory_desc, prod_category, prod_category_desc, prod_weight_class,
 prod_unit_of_measure, prod_pack_size, supplier_id, prod_status,
 prod_list_price, prod_min_price));
     END IF;
   END LOOP;
   CLOSE cur;
   RETURN;
END;
/
```

You can use the table function as follows:

```
SELECT DISTINCT prod_category,
                DECODE(prod_status,'obsolete','NO LONGER AVAILABLE','N/A')
FROM TABLE(obsolete_products_pipe(
  CURSOR(SELECT prod_id, prod_name, prod_desc, prod_subcategory,
         prod_subcategory_desc, prod_category, prod_category_desc,
         prod_weight_class, prod_unit_of_measure, prod_pack_size,
         supplier_id, prod_status, prod_list_price, prod_min_price
         FROM products)));
```

We now change the degree of parallelism for the input table products and issue the same statement again:

```
ALTER TABLE products PARALLEL 4;
```

The session statistics show that the statement has been parallelized:

```
SELECT * FROM V$PQ_SESSTAT WHERE statistic='Queries Parallelized';
```

```
STATISTIC              LAST_QUERY  SESSION_TOTAL
--------------------   ----------  -------------
Queries Parallelized            1              3

1 row selected.
```

Table functions are also capable to fanout results into persistent table structures. This is demonstrated in the next example. The function filters returns all obsolete products except a those of a specific `prod_category` (default Electronics), which was set to status `obsolete` by error. The result set of the table function consists of all other obsolete product categories. The detected wrong `prod_id`'s are stored in a separate table structure `obsolete_products_error`. Note that if a table function is part of an autonomous transaction, it must `COMMIT` or `ROLLBACK` before each `PIPE ROW` statement to avoid an error in the calling subprogram. Its result set consists of all other obsolete product categories. It furthermore demonstrates how normal variables can be used in conjunction with table functions:

```
CREATE OR REPLACE FUNCTION obsolete_products_dml(cur cursor_pkg.strong_refcur_t,
 prod_cat varchar2 DEFAULT 'Electronics') RETURN product_t_table
PIPELINED
PARALLEL_ENABLE (PARTITION cur BY ANY) IS
    PRAGMA AUTONOMOUS_TRANSACTION;
    prod_id                   NUMBER(6);
    prod_name                 VARCHAR2(50);
    prod_desc                 VARCHAR2(4000);
    prod_subcategory          VARCHAR2(50);
    prod_subcategory_desc     VARCHAR2(2000);
    prod_category             VARCHAR2(50);
    prod_category_desc        VARCHAR2(2000);
    prod_weight_class         NUMBER(2);
    prod_unit_of_measure      VARCHAR2(20);
    prod_pack_size            VARCHAR2(30);
    supplier_id               NUMBER(6);
    prod_status               VARCHAR2(20);
    prod_list_price      NUMBER(8,2);
    prod_min_price       NUMBER(8,2);
    sales                NUMBER:=0;
BEGIN
   LOOP
     -- Fetch from cursor variable
     FETCH cur INTO prod_id, prod_name, prod_desc, prod_subcategory,
  prod_subcategory_desc, prod_category, prod_category_desc, prod_weight_class,
  prod_unit_of_measure, prod_pack_size, supplier_id, prod_status,
```

```
    prod_list_price, prod_min_price;
    EXIT WHEN cur%NOTFOUND; -- exit when last row is fetched
    IF prod_status='obsolete' THEN
      IF prod_category=prod_cat THEN
         INSERT INTO obsolete_products_errors VALUES
         (prod_id, 'correction: category '||UPPER(prod_cat)||' still
  available');
         COMMIT;
      ELSE
      PIPE ROW (product_t( prod_id, prod_name, prod_desc, prod_subcategory,
 prod_subcategory_desc, prod_category, prod_category_desc, prod_weight_class,
 prod_unit_of_measure, prod_pack_size, supplier_id, prod_status,
 prod_list_price, prod_min_price));
      END IF;
    END IF;
  END LOOP;
  CLOSE cur;
  RETURN;
END;
/
```

The following query shows all obsolete product groups except the `prod_category` Electronics, which was wrongly set to status `obsolete`:

```
SELECT DISTINCT prod_category, prod_status FROM TABLE(obsolete_products_dml(
CURSOR(SELECT prod_id, prod_name, prod_desc, prod_subcategory,
  prod_subcategory_desc, prod_category, prod_category_desc, prod_weight_class,
  prod_unit_of_measure, prod_pack_size, supplier_id, prod_status,
  prod_list_price, prod_min_price
FROM products)));
```

As you can see, there are some products of the `prod_category` `Electronics` that were obsoleted by accident:

```
SELECT DISTINCT msg FROM obsolete_products_errors;
```

Taking advantage of the second input variable, you can specify a different product group than Electronics to be considered:

```
SELECT DISTINCT prod_category, prod_status
FROM TABLE(obsolete_products_dml(
CURSOR(SELECT prod_id, prod_name, prod_desc, prod_subcategory,
  prod_subcategory_desc, prod_category, prod_category_desc, prod_weight_class,
  prod_unit_of_measure, prod_pack_size, supplier_id, prod_status,
  prod_list_price, prod_min_price
FROM products),'Photo'));
```

Because table functions can be used like a normal table, they can be nested, as shown in the following:

```
SELECT DISTINCT prod_category, prod_status
FROM TABLE(obsolete_products_dml(CURSOR(SELECT *
FROM TABLE(obsolete_products_pipe(CURSOR(SELECT prod_id, prod_name, prod_desc,
 prod_subcategory, prod_subcategory_desc, prod_category, prod_category_desc,
 prod_weight_class, prod_unit_of_measure, prod_pack_size, supplier_id,
 prod_status, prod_list_price, prod_min_price
FROM products))))));
```

The biggest advantage of Oracle Database's ETL is its toolkit functionality, where you can combine any of the latter discussed functionality to improve and speed up your ETL processing. For example, you can take an external table as input, join it with an existing table and use it as input for a parallelized table function to process complex business logic. This table function can be used as input source for a `MERGE` operation, thus streaming the new information for the data warehouse, provided in a flat file within one single statement through the complete ETL process.

See *PL/SQL User's Guide and Reference* for details about table functions and the PL/SQL programming. For details about table functions implemented in other languages, see *Oracle Data Cartridge Developer's Guide*.

Loading and Transformation Scenarios

The following sections offer examples of typical loading and transformation tasks:

- Key Lookup Scenario
- Exception Handling Scenario
- Pivoting Scenarios

Key Lookup Scenario

A typical transformation is the key lookup. For example, suppose that sales transaction data has been loaded into a retail data warehouse. Although the data warehouse's `sales` table contains a `product_id` column, the sales transaction data extracted from the source system contains Uniform Price Codes (UPC) instead of product IDs. Therefore, it is necessary to transform the UPC codes into product IDs before the new sales transaction data can be inserted into the `sales` table.

In order to execute this transformation, a lookup table must relate the product_id values to the UPC codes. This table might be the product dimension table, or perhaps another table in the data warehouse that has been created specifically to support this transformation. For this example, we assume that there is a table named product, which has a product_id and an upc_code column.

This data substitution transformation can be implemented using the following CTAS statement:

```
CREATE TABLE temp_sales_step2 NOLOGGING PARALLEL AS SELECT sales_transaction_id,
  product.product_id sales_product_id, sales_customer_id, sales_time_id,
  sales_channel_id, sales_quantity_sold, sales_dollar_amount
FROM  temp_sales_step1, product
WHERE temp_sales_step1.upc_code = product.upc_code;
```

This CTAS statement will convert each valid UPC code to a valid product_id value. If the ETL process has guaranteed that each UPC code is valid, then this statement alone may be sufficient to implement the entire transformation.

Exception Handling Scenario

In the preceding example, if you must also handle new sales data that does not have valid UPC codes, you can use an additional CTAS statement to identify the invalid rows:

```
CREATE TABLE temp_sales_step1_invalid NOLOGGING PARALLEL AS
SELECT * FROM temp_sales_step1
WHERE temp_sales_step1.upc_code NOT IN (SELECT upc_code FROM product);
```

This invalid data is now stored in a separate table, temp_sales_step1_invalid, and can be handled separately by the ETL process.

Another way to handle invalid data is to modify the original CTAS to use an outer join:

```
CREATE TABLE temp_sales_step2 NOLOGGING PARALLEL AS
SELECT sales_transaction_id, product.product_id sales_product_id,
  sales_customer_id, sales_time_id, sales_channel_id, sales_quantity_sold,
   sales_dollar_amount
FROM  temp_sales_step1, product
WHERE temp_sales_step1.upc_code = product.upc_code (+);
```

Using this outer join, the sales transactions that originally contained invalidated UPC codes will be assigned a product_id of NULL. These transactions can be handled later.

Additional approaches to handling invalid UPC codes exist. Some data warehouses may choose to insert null-valued `product_id` values into their `sales` table, while other data warehouses may not allow any new data from the entire batch to be inserted into the `sales` table until all invalid UPC codes have been addressed. The correct approach is determined by the business requirements of the data warehouse. Regardless of the specific requirements, exception handling can be addressed by the same basic SQL techniques as transformations.

Pivoting Scenarios

A data warehouse can receive data from many different sources. Some of these source systems may not be relational databases and may store data in very different formats from the data warehouse. For example, suppose that you receive a set of sales records from a nonrelational database having the form:

```
product_id, customer_id, weekly_start_date, sales_sun, sales_mon, sales_tue,
  sales_wed, sales_thu, sales_fri, sales_sat
```

The input table looks like the following:

```
SELECT * FROM sales_input_table;

PRODUCT_ID CUSTOMER_ID WEEKLY_ST  SALES_SUN  SALES_MON  SALES_TUE  SALES_WED SALES_THU  SALES_FRI  SALES_SAT
---------- ----------- --------- ---------- ---------- ---------- ---------- ---------- ---------- ----------
       111         222 01-OCT-00        100        200        300        400        500        600        700
       222         333 08-OCT-00        200        300        400        500        600        700        800
       333         444 15-OCT-00        300        400        500        600        700        800        900
```

In your data warehouse, you would want to store the records in a more typical relational form in a fact table `sales` of the `sh` sample schema:

```
prod_id, cust_id, time_id, amount_sold
```

> **Note:** A number of constraints on the `sales` table have been disabled for purposes of this example, because the example ignores a number of table columns for the sake of brevity.

Thus, you need to build a transformation such that each record in the input stream must be converted into seven records for the data warehouse's `sales` table. This operation is commonly referred to as **pivoting**, and Oracle Database offers several ways to do this.

The result of the previous example will resemble the following:

```
SELECT prod_id, cust_id, time_id, amount_sold FROM sales;

   PROD_ID    CUST_ID   TIME_ID   AMOUNT_SOLD
---------- ----------   --------- -----------
       111        222   01-OCT-00         100
       111        222   02-OCT-00         200
       111        222   03-OCT-00         300
       111        222   04-OCT-00         400
       111        222   05-OCT-00         500
       111        222   06-OCT-00         600
       111        222   07-OCT-00         700
       222        333   08-OCT-00         200
       222        333   09-OCT-00         300
       222        333   10-OCT-00         400
       222        333   11-OCT-00         500
       222        333   12-OCT-00         600
       222        333   13-OCT-00         700
       222        333   14-OCT-00         800
       333        444   15-OCT-00         300
       333        444   16-OCT-00         400
       333        444   17-OCT-00         500
       333        444   18-OCT-00         600
       333        444   19-OCT-00         700
       333        444   20-OCT-00         800
       333        444   21-OCT-00         900
```

Example 14–7 Pivoting

The following example uses the multitable insert syntax to insert into the demo table `sh.sales` some data from an input table with a different structure. The multitable insert statement looks like the following:

```
INSERT ALL INTO sales (prod_id, cust_id, time_id, amount_sold)
      VALUES (product_id, customer_id, weekly_start_date, sales_sun)
      INTO sales (prod_id, cust_id, time_id, amount_sold)
      VALUES (product_id, customer_id, weekly_start_date+1, sales_mon)
      INTO sales (prod_id, cust_id, time_id, amount_sold)
      VALUES (product_id, customer_id, weekly_start_date+2, sales_tue)
      INTO sales (prod_id, cust_id, time_id, amount_sold)
      VALUES (product_id, customer_id, weekly_start_date+3, sales_wed)
      INTO sales (prod_id, cust_id, time_id, amount_sold)
      VALUES (product_id, customer_id, weekly_start_date+4, sales_thu)
      INTO sales (prod_id, cust_id, time_id, amount_sold)
      VALUES (product_id, customer_id, weekly_start_date+5, sales_fri)
      INTO sales (prod_id, cust_id, time_id, amount_sold)
```

```
      VALUES (product_id, customer_id, weekly_start_date+6, sales_sat)
SELECT product_id, customer_id, weekly_start_date, sales_sun,
      sales_mon, sales_tue, sales_wed, sales_thu, sales_fri, sales_sat
FROM sales_input_table;
```

This statement only scans the source table once and then inserts the appropriate data for each day.

15

Maintaining the Data Warehouse

This chapter discusses how to load and refresh a data warehouse, and discusses:

- Using Partitioning to Improve Data Warehouse Refresh
- Optimizing DML Operations During Refresh
- Refreshing Materialized Views
- Using Materialized Views with Partitioned Tables

Using Partitioning to Improve Data Warehouse Refresh

ETL (Extraction, Transformation and Loading) is done on a scheduled basis to reflect changes made to the original source system. During this step, you physically insert the new, clean data into the production data warehouse schema, and take all of the other steps necessary (such as building indexes, validating constraints, taking backups) to make this new data available to the end users. Once all of this data has been loaded into the data warehouse, the materialized views have to be updated to reflect the latest data.

The partitioning scheme of the data warehouse is often crucial in determining the efficiency of refresh operations in the data warehouse load process. In fact, the load process is often the primary consideration in choosing the partitioning scheme of data warehouse tables and indexes.

The partitioning scheme of the largest data warehouse tables (for example, the fact table in a star schema) should be based upon the loading paradigm of the data warehouse.

Most data warehouses are loaded with new data on a regular schedule. For example, every night, week, or month, new data is brought into the data warehouse. The data being loaded at the end of the week or month typically corresponds to the transactions for the week or month. In this very common scenario, the data warehouse is being loaded by time. This suggests that the data warehouse tables should be partitioned on a date column. In our data warehouse example, suppose the new data is loaded into the `sales` table every month. Furthermore, the `sales` table has been partitioned by month. These steps show how the load process will proceed to add the data for a new month (January 2001) to the table `sales`.

1. Place the new data into a separate table, `sales_01_2001`. This data can be directly loaded into `sales_01_2001` from outside the data warehouse, or this data can be the result of previous data transformation operations that have already occurred in the data warehouse. `sales_01_2001` has the exact same columns, datatypes, and so forth, as the `sales` table. Gather statistics on the `sales_01_2001` table.

2. Create indexes and add constraints on `sales_01_2001`. Again, the indexes and constraints on `sales_01_2001` should be identical to the indexes and constraints on `sales`. Indexes can be built in parallel and should use the `NOLOGGING` and the `COMPUTE STATISTICS` options. For example:

```
CREATE BITMAP INDEX sales_01_2001_customer_id_bix
  ON sales_01_2001(customer_id)
      TABLESPACE sales_idx NOLOGGING PARALLEL 8 COMPUTE STATISTICS;
```

Apply all constraints to the sales_01_2001 table that are present on the sales table. This includes referential integrity constraints. A typical constraint would be:

```
ALTER TABLE sales_01_2001 ADD CONSTRAINT sales_customer_id
      REFERENCES customer(customer_id) ENABLE NOVALIDATE;
```

If the partitioned table sales has a primary or unique key that is enforced with a global index structure, ensure that the constraint on sales_pk_jan01 is validated without the creation of an index structure, as in the following:

```
ALTER TABLE sales_01_2001 ADD CONSTRAINT sales_pk_jan01
PRIMARY KEY (sales_transaction_id) DISABLE VALIDATE;
```

The creation of the constraint with ENABLE clause would cause the creation of a unique index, which does not match a local index structure of the partitioned table. You must not have any index structure built on the nonpartitioned table to be exchanged for existing global indexes of the partitioned table. The exchange command would fail.

3. Add the sales_01_2001 table to the sales table.

In order to add this new data to the sales table, we need to do two things. First, we need to add a new partition to the sales table. We will use the ALTER TABLE ... ADD PARTITION statement. This will add an empty partition to the sales table:

```
ALTER TABLE sales ADD PARTITION sales_01_2001
VALUES LESS THAN (TO_DATE('01-FEB-2001', 'DD-MON-YYYY'));
```

Then, we can add our newly created table to this partition using the EXCHANGE PARTITION operation. This will exchange the new, empty partition with the newly loaded table.

```
ALTER TABLE sales EXCHANGE PARTITION sales_01_2001 WITH TABLE sales_01_2001
INCLUDING INDEXES WITHOUT VALIDATION UPDATE GLOBAL INDEXES;
```

The EXCHANGE operation will preserve the indexes and constraints that were already present on the sales_01_2001 table. For unique constraints (such as the unique constraint on sales_transaction_id), you can use the UPDATE GLOBAL INDEXES clause, as shown previously. This will automatically maintain your global index structures as part of the partition maintenance operation and keep them accessible throughout the whole process. If there were only foreign-key constraints, the exchange operation would be instantaneous.

The benefits of this partitioning technique are significant. First, the new data is loaded with minimal resource utilization. The new data is loaded into an entirely separate table, and the index processing and constraint processing are applied only to the new partition. If the `sales` table was 50 GB and had 12 partitions, then a new month's worth of data contains approximately 4 GB. Only the new month's worth of data needs to be indexed. None of the indexes on the remaining 46 GB of data needs to be modified at all. This partitioning scheme additionally ensures that the load processing time is directly proportional to the amount of new data being loaded, not to the total size of the `sales` table.

Second, the new data is loaded with minimal impact on concurrent queries. All of the operations associated with data loading are occurring on a separate `sales_01_2001` table. Therefore, none of the existing data or indexes of the `sales` table is affected during this data refresh process. The `sales` table and its indexes remain entirely untouched throughout this refresh process.

Third, in case of the existence of any global indexes, those are incrementally maintained as part of the exchange command. This maintenance does not affect the availability of the existing global index structures.

The exchange operation can be viewed as a publishing mechanism. Until the data warehouse administrator exchanges the `sales_01_2001` table into the `sales` table, end users cannot see the new data. Once the exchange has occurred, then any end user query accessing the `sales` table will immediately be able to see the `sales_01_2001` data.

Partitioning is useful not only for adding new data but also for removing and archiving data. Many data warehouses maintain a rolling window of data. For example, the data warehouse stores the most recent 36 months of `sales` data. Just as a new partition can be added to the `sales` table (as described earlier), an old partition can be quickly (and independently) removed from the `sales` table. These two benefits (reduced resources utilization and minimal end-user impact) are just as pertinent to removing a partition as they are to adding a partition.

Removing data from a partitioned table does not necessarily mean that the old data is physically deleted from the database. There are two alternatives for removing old data from a partitioned table. First, you can physically delete all data from the database by dropping the partition containing the old data, thus freeing the allocated space:

```
ALTER TABLE sales DROP PARTITION sales_01_1998;
```

Also, you can exchange the old partition with an empty table of the same structure; this empty table is created equivalent to step1 and 2 described in the load process.

Assuming the new empty table stub is named sales_archive_01_1998, the following SQL statement will 'empty' partition sales_01_1998:

```
ALTER TABLE sales EXCHANGE PARTITION sales_01_1998 WITH TABLE sales_archive_01_
1998 INCLUDING INDEXES WITHOUT VALIDATION UPDATE GLOBAL INDEXES;
```

Note that the old data is still existent as the exchanged, nonpartitioned table sales_archive_01_1998.

If the partitioned table was setup in a way that every partition is stored in a separate tablespace, you can archive (or transport) this table using Oracle Database's transportable tablespace framework before dropping the actual data (the tablespace). See "Transportation Using Transportable Tablespaces" on page 15-5 for further details regarding transportable tablespaces.

In some situations, you might not want to drop the old data immediately, but keep it as part of the partitioned table; although the data is no longer of main interest, there are still potential queries accessing this old, read-only data. You can use Oracle's data compression to minimize the space usage of the old data. We also assume that at least one compressed partition is already part of the partitioned table. See Chapter 3, "Physical Design in Data Warehouses" for a generic discussion of table compression and Chapter 5, "Parallelism and Partitioning in Data Warehouses" for partitioning and table compression.

Refresh Scenarios

A typical scenario might not only need to compress old data, but also to merge several old partitions to reflect the granularity for a later backup of several merged partitions. Let's assume that a backup (partition) granularity is on a quarterly base for any quarter, where the oldest month is more than 36 months behind the most recent month. In this case, we are therefore compressing and merging sales_01_1998, sales_02_1998, and sales_03_1998 into a new, compressed partition sales_q1_1998.

1. Create the new merged partition in parallel another tablespace. The partition will be compressed as part of the MERGE operation:

   ```
   ALTER TABLE sales MERGE PARTITION sales_01_1998, sales_02_1998, sales_03_
   1998 INTO PARTITION sales_q1_1998 TABLESPACE archive_q1_1998
   COMPRESS UPDATE GLOBAL INDEXES PARALLEL 4;
   ```

2. The partition MERGE operation invalidates the local indexes for the new merged partition. We therefore have to rebuild them:

   ```
   ALTER TABLE sales MODIFY PARTITION sales_1_1998
   ```

```
REBUILD UNUSABLE LOCAL INDEXES;
```

Alternatively, you can choose to create the new compressed table outside the partitioned table and exchange it back. The performance and the temporary space consumption is identical for both methods:

1. Create an intermediate table to hold the new merged information. The following statement inherits all NOT NULL constraints from the origin table by default:

   ```
   CREATE TABLE sales_q1_1998_out TABLESPACE archive_q1_1998 NOLOGGING COMPRESS
   PARALLEL 4 AS SELECT * FROM sales
   WHERE time_id >=  TO_DATE('01-JAN-1998','dd-mon-yyyy')
     AND time_id < TO_DATE('01-JUN-1998','dd-mon-yyyy');
   ```

2. Create the equivalent index structure for table sales_q1_1998_out than for the existing table sales.

3. Prepare the existing table sales for the exchange with the new compressed table sales_q1_1998_out. Because the table to be exchanged contains data actually covered in three partition, we have to 'create one matching partition, having the range boundaries we are looking for. You simply have to drop two of the existing partitions. Note that you have to drop the lower two partitions sales_01_1998 and sales_02_1998; the lower boundary of a range partition is always defined by the upper (exclusive) boundary of the previous partition:

   ```
   ALTER TABLE sales DROP PARTITION sales_01_1998;
   ALTER TABLE sales DROP PARTITION sales_02_1998;
   ```

4. You can now exchange table sales_q1_1998_out with partition sales_03_1998. Unlike what the name of the partition suggests, its boundaries cover Q1-1998.

   ```
   ALTER TABLE sales EXCHANGE PARTITION sales_03_1998
   WITH TABLE sales_q1_1998_out INCLUDING INDEXES WITHOUT VALIDATION
   UPDATE GLOBAL INDEXES;
   ```

Both methods apply to slightly different business scenarios: Using the MERGE PARTITION approach invalidates the local index structures for the affected partition, but it keeps all data accessible all the time. Any attempt to access the affected partition through one of the unusable index structures raises an error. The limited availability time is approximately the time for re-creating the local bitmap index structures. In most cases this can be neglected, since this part of the partitioned table shouldn't be touched too often.

The CTAS approach, however, minimizes unavailability of any index structures close to zero, but there is a specific time window, where the partitioned table does not have all the data, because we dropped two partitions. The limited availability time is approximately the time for exchanging the table. Depending on the existence and number of global indexes, this time window varies. Without any existing global indexes, this time window is a matter of a fraction to few seconds.

These examples are a simplification of the data warehouse rolling window load scenario. Real-world data warehouse refresh characteristics are always more complex. However, the advantages of this rolling window approach are not diminished in more complex scenarios.

Note that before you add single or multiple compressed partitions to a partitioned table for the first time, all local bitmap indexes must be either dropped or marked unusable. After the first compressed partition is added, no additional actions are necessary for all subsequent operations involving compressed partitions. It is irrelevant how the compressed partitions are added to the partitioned table. See Chapter 5, "Parallelism and Partitioning in Data Warehouses" for further details about partitioning and table compression.

Scenarios for Using Partitioning for Refreshing Data Warehouses

This section contains two typical scenarios where partitioning is used with refresh.

Refresh Scenario 1

Data is loaded daily. However, the data warehouse contains two years of data, so that partitioning by day might not be desired.

The solution is to partition by week or month (as appropriate). Use `INSERT` to add the new data to an existing partition. The `INSERT` operation only affects a single partition, so the benefits described previously remain intact. The `INSERT` operation could occur while the partition remains a part of the table. Inserts into a single partition can be parallelized:

```
INSERT /*+ APPEND*/ INTO sales PARTITION (sales_01_2001)
SELECT * FROM new_sales;
```

The indexes of this `sales` partition will be maintained in parallel as well. An alternative is to use the `EXCHANGE` operation. You can do this by exchanging the `sales_01_2001` partition of the `sales` table and then using an `INSERT` operation. You might prefer this technique when dropping and rebuilding indexes is more efficient than maintaining them.

Refresh Scenario 2

New data feeds, although consisting primarily of data for the most recent day, week, and month, also contain some data from previous time periods.

Solution 1 Use parallel SQL operations (such as CREATE TABLE ... AS SELECT) to separate the new data from the data in previous time periods. Process the old data separately using other techniques.

New data feeds are not solely time based. You can also feed new data into a data warehouse with data from multiple operational systems on a business need basis. For example, the sales data from direct channels may come into the data warehouse separately from the data from indirect channels. For business reasons, it may furthermore make sense to keep the direct and indirect data in separate partitions.

Solution 2 Oracle supports composite range list partitioning. The primary partitioning strategy of the sales table could be range partitioning based on `time_id` as shown in the example. However, the subpartitioning is a list based on the channel attribute. Each subpartition can now be loaded independently of each other (for each distinct channel) and added in a rolling window operation as discussed before. The partitioning strategy addresses the business needs in the most optimal manner.

Optimizing DML Operations During Refresh

You can optimize DML performance through the following techniques:

- Implementing an Efficient MERGE Operation
- Maintaining Referential Integrity
- Purging Data

Implementing an Efficient MERGE Operation

Commonly, the data that is extracted from a source system is not simply a list of new records that needs to be inserted into the data warehouse. Instead, this new data set is a combination of new records as well as modified records. For example, suppose that most of data extracted from the OLTP systems will be new sales transactions. These records will be inserted into the warehouse's `sales` table, but some records may reflect modifications of previous transactions, such as returned merchandise or transactions that were incomplete or incorrect when initially loaded into the data warehouse. These records require updates to the `sales` table.

As a typical scenario, suppose that there is a table called new_sales that contains both inserts and updates that will be applied to the sales table. When designing the entire data warehouse load process, it was determined that the new_sales table would contain records with the following semantics:

- If a given sales_transaction_id of a record in new_sales already exists in sales, then update the sales table by adding the sales_dollar_amount and sales_quantity_sold values from the new_sales table to the existing row in the sales table.
- Otherwise, insert the entire new record from the new_sales table into the sales table.

This UPDATE-ELSE-INSERT operation is often called a merge. A merge can be executed using one SQL statement.

Example 15–1 MERGE Operation

```
MERGE INTO sales s USING new_sales n
ON (s.sales_transaction_id = n.sales_transaction_id)
WHEN MATCHED THEN
UPDATE s_quantity = s_quantity + n_quantity, s_dollar = s_dollar + n_dollar
WHEN NOT MATCHED THEN INSERT (sales_quantity_sold, sales_dollar_amount)
VALUES (n.sales_quantity_sold, n.sales_dollar_amount);
```

In addition to using the MERGE statement for unconditional UPDATE ELSE INSERT functionality into a target table, you can also use it to:

- Perform an UPDATE only or INSERT only statement.
- Apply additional WHERE conditions for the UPDATE or INSERT portion of the MERGE statement.
- The UPDATE operation can even delete rows if a specific condition yields true.

Example 15–2 Omitting the INSERT Clause

In some data warehouse applications, it is not allowed to add new rows to historical information, but only to update them. It may also happen that you don't want to update but only insert new information. The following examples demonstrate the INSERT-only respective the UPDATE-only functionality:

```
MERGE USING Product_Changes S     -- Source/Delta table
INTO Products D1                  -- Destination table 1
ON (D1.PROD_ID = S.PROD_ID)       -- Search/Join condition
WHEN MATCHED THEN UPDATE          -- update if join
```

```
SET D1.PROD_STATUS = S.PROD_NEW_STATUS
```

Example 15–3 Omitting the UPDATE Clause

The following statement illustrates an example of omitting an UPDATE:

```
MERGE USING New_Product S           -- Source/Delta table
INTO Products D2                    -- Destination table 2
ON (D2.PROD_ID = S.PROD_ID)         -- Search/Join condition
WHEN NOT MATCHED THEN               -- insert if no join
INSERT (PROD_ID, PROD_STATUS) VALUES (S.PROD_ID, S.PROD_NEW_STATUS)
```

When the INSERT clause is omitted, Oracle performs a regular join of the source and the target tables. When the UPDATE clause is omitted, Oracle performs an antijoin of the source and the target tables. This makes the join between the source and target table more efficient.

Example 15–4 Skipping the UPDATE Clause

In some situations, you may want to skip the UPDATE operation when merging a given row into the table. In this case, you can use an optional WHERE clause in the UPDATE clause of the MERGE. As a result, the UPDATE operation only executes when a given condition is true. The following statement illustrates an example of skipping the UPDATE operation:

```
MERGE
USING Product_Changes S                      -- Source/Delta table
INTO Products P                              -- Destination table 1
ON (P.PROD_ID = S.PROD_ID)                   -- Search/Join condition
WHEN MATCHED THEN
UPDATE                                       -- update if join
SET P.PROD_LIST_PRICE = S.PROD_NEW_PRICE
WHERE P.PROD_STATUS <> "OBSOLETE"            -- Conditional UPDATE
```

This shows how the UPDATE operation would be skipped if the condition P.PROD_STATUS <> "OBSOLETE" is not true. The condition predicate can refer to both the target and the source table.

Example 15–5 Conditional Inserts with MERGE Statements

You may want to skip the INSERT operation when merging a given row into the table. So an optional WHERE clause is added to the INSERT clause of the MERGE. As a result, the INSERT operation only executes when a given condition is true. The following statement offers an example:

```
MERGE USING Product_Changes S        -- Source/Delta table
INTO Products P                      -- Destination table 1
ON (P.PROD_ID = S.PROD_ID)           -- Search/Join condition
WHEN MATCHED THEN UPDATE             -- update if join
SET P.PROD_LIST_PRICE = S.PROD_NEW_PRICE
WHERE P.PROD_STATUS <> "OBSOLETE"    -- Conditional
UPDATE WHEN NOT MATCHED THEN
INSERT                               -- insert if not join
SET P.PROD_LIST_PRICE = S.PROD_NEW_PRICE
WHERE S.PROD_STATUS <> "OBSOLETE"    -- Conditional INSERT
```

This example shows that the INSERT operation would be skipped if the condition S.PROD_STATUS <> "OBSOLETE" is not true, and INSERT will only occur if the condition is true. The condition predicate can refer to the source table only. This predicate would be most likely a column filter.

Example 15–6 Using the DELETE Clause with MERGE Statements

You may want to cleanse tables while populating or updating them. To do this, you may want to consider using the DELETE clause in a MERGE statement, as in the following example:

```
MERGE USING Product_Changes S
INTO Products D ON (D.PROD_ID = S.PROD_ID)
WHEN MATCHED THEN
UPDATE SET D.PROD_LIST_PRICE =S.PROD_NEW_PRICE, D.PROD_STATUS = S.PROD_
NEWSTATUS
DELETE WHERE (D.PROD_STATUS = "OBSOLETE")
WHEN NOT MATCHED THEN
INSERT (PROD_ID, PROD_LIST_PRICE, PROD_STATUS)
VALUES (S.PROD_ID, S.PROD_NEW_PRICE, S.PROD_NEW_STATUS);
```

Thus when a row is updated in products, Oracle checks the delete condition D.PROD_STATUS = "OBSOLETE", and deletes the row if the condition yields true.

The DELETE operation is not as same as that of a complete DELETE statement. Only the rows from the destination of the MERGE can be deleted. The only rows that will be affected by the DELETE are the ones that are updated by this MERGE statement. Thus, although a give row of the destination table meets the delete condition, if it does not join under the ON clause condition, it will not be deleted.

Example 15–7 Unconditional Inserts with MERGE Statements

You may want to insert all of the source rows into a table. In this case, the join between the source and target table can be avoided. By identifying special constant

join conditions that always result to FALSE, for example, 1=0, such MERGE statements will be optimized and the join condition will be suppressed.

```
MERGE USING New_Product S       -- Source/Delta table
INTO Products P                 -- Destination table 1
ON (1 = 0)                      -- Search/Join condition
WHEN NOT MATCHED THEN           -- insert if no join
INSERT (PROD_ID, PROD_STATUS) VALUES (S.PROD_ID, S.PROD_NEW_STATUS)
```

Maintaining Referential Integrity

In some data warehousing environments, you might want to insert new data into tables in order to guarantee referential integrity. For example, a data warehouse may derive `sales` from an operational system that retrieves data directly from cash registers. `sales` is refreshed nightly. However, the data for the `product` dimension table may be derived from a separate operational system. The `product` dimension table may only be refreshed once for each week, because the `product` table changes relatively slowly. If a new product was introduced on Monday, then it is possible for that product's `product_id` to appear in the `sales` data of the data warehouse before that `product_id` has been inserted into the data warehouses `product` table.

Although the sales transactions of the new product may be valid, this sales data will not satisfy the referential integrity constraint between the `product` dimension table and the `sales` fact table. Rather than disallow the new sales transactions, you might choose to insert the sales transactions into the `sales` table. However, you might also wish to maintain the referential integrity relationship between the `sales` and `product` tables. This can be accomplished by inserting new rows into the `product` table as placeholders for the unknown products.

As in previous examples, we assume that the new data for the `sales` table will be staged in a separate table, `new_sales`. Using a single `INSERT` statement (which can be parallelized), the `product` table can be altered to reflect the new products:

```
INSERT INTO PRODUCT_ID
  (SELECT sales_product_id, 'Unknown Product Name', NULL, NULL ...
   FROM new_sales WHERE sales_product_id NOT IN
  (SELECT product_id FROM product));
```

Purging Data

Occasionally, it is necessary to remove large amounts of data from a data warehouse. A very common scenario is the rolling window discussed previously, in which older data is rolled out of the data warehouse to make room for new data.

However, sometimes other data might need to be removed from a data warehouse. Suppose that a retail company has previously sold products from MS Software, and that MS Software has subsequently gone out of business. The business users of the warehouse may decide that they are no longer interested in seeing any data related to MS Software, so this data should be deleted.

One approach to removing a large volume of data is to use parallel delete as shown in the following statement:

```
DELETE FROM sales WHERE sales_product_id IN (SELECT product_id
   FROM product WHERE product_category = 'MS Software');
```

This SQL statement will spawn one parallel process for each partition. This approach will be much more efficient than a serial DELETE statement, and none of the data in the sales table will need to be moved. However, this approach also has some disadvantages. When removing a large percentage of rows, the DELETE statement will leave many empty row-slots in the existing partitions. If new data is being loaded using a rolling window technique (or is being loaded using direct-path INSERT or load), then this storage space will not be reclaimed. Moreover, even though the DELETE statement is parallelized, there might be more efficient methods. An alternative method is to re-create the entire sales table, keeping the data for all product categories except MS Software.

```
CREATE TABLE sales2 AS SELECT * FROM sales, product
WHERE sales.sales_product_id = product.product_id
AND product_category <> 'MS Software'
NOLOGGING PARALLEL (DEGREE 8)
#PARTITION ... ; #create indexes, constraints, and so on
DROP TABLE SALES;
RENAME SALES2 TO SALES;
```

This approach may be more efficient than a parallel delete. However, it is also costly in terms of the amount of disk space, because the sales table must effectively be instantiated twice.

An alternative method to utilize less space is to re-create the sales table one partition at a time:

```
CREATE TABLE sales_temp AS SELECT * FROM sales WHERE 1=0;
INSERT INTO sales_temp PARTITION (sales_99jan)
SELECT * FROM sales, product
WHERE sales.sales_product_id = product.product_id
AND product_category <> 'MS Software';
<create appropriate indexes and constraints on sales_temp>
ALTER TABLE sales EXCHANGE PARTITION sales_99jan WITH TABLE sales_temp;
```

Continue this process for each partition in the sales table.

Refreshing Materialized Views

When creating a materialized view, you have the option of specifying whether the refresh occurs ON DEMAND or ON COMMIT. In the case of ON COMMIT, the materialized view is changed every time a transaction commits, which changes data used by the materialized view, thus ensuring that the materialized view always contains the latest data. Alternatively, you can control the time when refresh of the materialized views occurs by specifying ON DEMAND. In this case, the materialized view can only be refreshed by calling one of the procedures in the DBMS_MVIEW package.

DBMS_MVIEW provides three different types of refresh operations.

- DBMS_MVIEW.REFRESH

  Refresh one or more materialized views.

- DBMS_MVIEW.REFRESH_ALL_MVIEWS

  Refresh all materialized views.

- DBMS_MVIEW.REFRESH_DEPENDENT

  Refresh all materialized views that depend on a specified master table or materialized view or list of master tables or materialized views.

  See Also: "Manual Refresh Using the DBMS_MVIEW Package" on page 15-16 for more information about this package

Performing a refresh operation requires temporary space to rebuild the indexes and can require additional space for performing the refresh operation itself. Some sites might prefer not to refresh all of their materialized views at the same time: as soon as some underlying detail data has been updated, all materialized views using this data will become stale. Therefore, if you defer refreshing your materialized views, you can either rely on your chosen rewrite integrity level to determine whether or not a stale materialized view can be used for query rewrite, or you can temporarily disable query rewrite with an ALTER SYSTEM SET QUERY_REWRITE_ENABLED = FALSE statement. After refreshing the materialized views, you can re-enable query rewrite as the default for all sessions in the current database instance by specifying ALTER SYSTEM SET QUERY_REWRITE_ENABLED as TRUE. Refreshing a materialized view automatically updates all of its indexes. In the case of full refresh,

this requires temporary sort space to rebuild all indexes during refresh. This is because the full refresh truncates or deletes the table before inserting the new full data volume. If insufficient temporary space is available to rebuild the indexes, then you must explicitly drop each index or mark it UNUSABLE prior to performing the refresh operation.

If you anticipate performing insert, update or delete operations on tables referenced by a materialized view concurrently with the refresh of that materialized view, and that materialized view includes joins and aggregation, Oracle recommends you use ON COMMIT fast refresh rather than ON DEMAND fast refresh.

Complete Refresh

A complete refresh occurs when the materialized view is initially defined as BUILD IMMEDIATE, unless the materialized view references a prebuilt table. For materialized views using BUILD DEFERRED, a complete refresh must be requested before it can be used for the first time. A complete refresh may be requested at any time during the life of any materialized view. The refresh involves reading the detail tables to compute the results for the materialized view. This can be a very time-consuming process, especially if there are huge amounts of data to be read and processed. Therefore, you should always consider the time required to process a complete refresh before requesting it.

There are, however, cases when the only refresh method available for an already built materialized view is complete refresh because the materialized view does not satisfy the conditions specified in the following section for a fast refresh.

Fast Refresh

Most data warehouses have periodic incremental updates to their detail data. As described in "Materialized View Schema Design" on page 8-8, you can use the SQL*Loader or any bulk load utility to perform incremental loads of detail data. Fast refresh of your materialized views is usually efficient, because instead of having to recompute the entire materialized view, the changes are applied to the existing data. Thus, processing only the changes can result in a very fast refresh time.

Partition Change Tracking (PCT) Refresh

When there have been some partition maintenance operations on the detail tables, this is the only method of fast refresh that can be used. PCT-based refresh on a

materialized view is enabled only if all the conditions described in "Partition Change Tracking" on page 9-2 are satisfied.

In the absence of partition maintenance operations on detail tables, when you request for a FAST method (`method => 'F'`) of refresh through procedures in DBMS_MVIEW package, Oracle will choose PCT refresh if it is enabled on the materialized view and is determined to be the better than log-based fast refresh. Similarly, when you request for a FORCE method (`method => '?'`), Oracle will choose PCT refresh if it is enabled on the materialized view and is calculated to be the better than log-based fast refresh and complete refresh. Alternatively, you can request the PCT method (`method => 'P'`), and Oracle will use the PCT method provided all PCT requirements are satisfied.

Oracle can use TRUNCATE PARTITION on a materialized view if it satisfies the conditions in "Benefits of Partitioning a Materialized View" on page 9-8 and hence, make the PCT refresh process more efficient.

ON COMMIT Refresh

A materialized view can be refreshed automatically using the ON COMMIT method. Therefore, whenever a transaction commits which has updated the tables on which a materialized view is defined, those changes will be automatically reflected in the materialized view. The advantage of using this approach is you never have to remember to refresh the materialized view. The only disadvantage is the time required to complete the commit will be slightly longer because of the extra processing involved. However, in a data warehouse, this should not be an issue because there is unlikely to be concurrent processes trying to update the same table.

Manual Refresh Using the DBMS_MVIEW Package

When a materialized view is refreshed ON DEMAND, one of four refresh methods can be specified as shown in the following table. You can define a default option during the creation of the materialized view. Table 15–1 details the refresh options.

Table 15–1 ON DEMAND Refresh Methods

Refresh Option	Parameter	Description
COMPLETE	C	Refreshes by recalculating the defining query of the materialized view

Table 15–1 (Cont.) ON DEMAND Refresh Methods

Refresh Option	Parameter	Description
FAST	F	Refreshes by incrementally applying changes to the materialized view. For local materialized views, it chooses the refresh method which is estimated by optimizer to be most efficient. The refresh methods considered are log-based FAST and FAST_PCT.
FAST_PCT	P	Refreshes by recomputing the rows in the materialized view affected by changed partitions in the detail tables.
FORCE	?	Attempts a fast refresh. If that is not possible, it does a complete refresh. For local materialized views, it chooses the refresh method which is estimated by optimizer to be most efficient. The refresh methods considered are log based FAST, FAST_PCT, and COMPLETE.

Three refresh procedures are available in the DBMS_MVIEW package for performing ON DEMAND refresh. Each has its own unique set of parameters.

> **See Also:** *PL/SQL Packages and Types Reference* for detailed information about the DBMS_MVIEW package and *Oracle Database Advanced Replication* for information showing how to use it in a replication environment

Refresh Specific Materialized Views with REFRESH

Use the DBMS_MVIEW.REFRESH procedure to refresh one or more materialized views. Some parameters are used only for replication, so they are not mentioned here. The required parameters to use this procedure are:

- The comma-delimited list of materialized views to refresh
- The refresh method: F-Fast, P-Fast_PCT, ?-Force, C-Complete
- The rollback segment to use
- Refresh after errors (TRUE or FALSE)

  A Boolean parameter. If set to TRUE, the number_of_failures output parameter will be set to the number of refreshes that failed, and a generic error message will indicate that failures occurred. The alert log for the instance will give details of refresh errors. If set to FALSE, the default, then refresh will stop after it encounters the first error, and any remaining materialized views in the list will not be refreshed.
- The following four parameters are used by the replication process. For warehouse refresh, set them to FALSE, 0,0,0.

- Atomic refresh (TRUE or FALSE)

  If set to TRUE, then all refreshes are done in one transaction. If set to FALSE, then the refresh of each specified materialized view is done in a separate transaction. If set to FALSE, Oracle can optimize refresh by using parallel DML and truncate DDL on a materialized views.

For example, to perform a fast refresh on the materialized view cal_month_sales_mv, the DBMS_MVIEW package would be called as follows:

```
DBMS_MVIEW.REFRESH('CAL_MONTH_SALES_MV', 'F', '', TRUE, FALSE, 0,0,0, FALSE);
```

Multiple materialized views can be refreshed at the same time, and they do not all have to use the same refresh method. To give them different refresh methods, specify multiple method codes in the same order as the list of materialized views (without commas). For example, the following specifies that cal_month_sales_mv be completely refreshed and fweek_pscat_sales_mv receive a fast refresh:

```
DBMS_MVIEW.REFRESH('CAL_MONTH_SALES_MV, FWEEK_PSCAT_SALES_MV', 'CF', '',
  TRUE, FALSE, 0,0,0, FALSE);
```

If the refresh method is not specified, the default refresh method as specified in the materialized view definition will be used.

Refresh All Materialized Views with REFRESH_ALL_MVIEWS

An alternative to specifying the materialized views to refresh is to use the procedure DBMS_MVIEW.REFRESH_ALL_MVIEWS. This procedure refreshes all materialized views. If any of the materialized views fails to refresh, then the number of failures is reported.

The parameters for this procedure are:

- The number of failures (this is an OUT variable)
- The refresh method: F-Fast, P-Fast_PCT, ?-Force, C-Complete
- Refresh after errors (TRUE or FALSE)

  A Boolean parameter. If set to TRUE, the number_of_failures output parameter will be set to the number of refreshes that failed, and a generic error message will indicate that failures occurred. The alert log for the instance will give details of refresh errors. If set to FALSE, the default, then refresh will stop after it encounters the first error, and any remaining materialized views in the list will not be refreshed.

- Atomic refresh (TRUE or FALSE)

If set to TRUE, then all refreshes are done in one transaction. If set to FALSE, then the refresh of each specified materialized view is done in a separate transaction. If set to FALSE, Oracle can optimize refresh by using parallel DML and truncate DDL on a materialized views.

An example of refreshing all materialized views is the following:

```
DBMS_MVIEW.REFRESH_ALL_MVIEWS(failures,'C','', TRUE, FALSE);
```

Refresh Dependent Materialized Views with REFRESH_DEPENDENT

The third procedure, DBMS_MVIEW.REFRESH_DEPENDENT, refreshes only those materialized views that depend on a specific table or list of tables. For example, suppose the changes have been received for the orders table but not for customer payments. The refresh dependent procedure can be called to refresh only those materialized views that reference the orders table.

The parameters for this procedure are:

- The number of failures (this is an OUT variable)
- The dependent table
- The refresh method: F-Fast, P-Fast_PCT, ?-Force, C-Complete
- The rollback segment to use
- Refresh after errors (TRUE or FALSE)

  A Boolean parameter. If set to TRUE, the number_of_failures output parameter will be set to the number of refreshes that failed, and a generic error message will indicate that failures occurred. The alert log for the instance will give details of refresh errors. If set to FALSE, the default, then refresh will stop after it encounters the first error, and any remaining materialized views in the list will not be refreshed.

- Atomic refresh (TRUE or FALSE)

  If set to TRUE, then all refreshes are done in one transaction. If set to FALSE, then the refresh of each specified materialized view is done in a separate transaction. If set to FALSE, Oracle can optimize refresh by using parallel DML and truncate DDL on a materialized views.

- Whether it is nested or not

  If set to TRUE, refresh all the dependent materialized views of the specified set of tables based on a dependency order to ensure the materialized views are truly fresh with respect to the underlying base tables.

To perform a full refresh on all materialized views that reference the customers table, specify:

```
DBMS_MVIEW.REFRESH_DEPENDENT(failures, 'CUSTOMERS', 'C', '', FALSE, FALSE );
```

To obtain the list of materialized views that are directly dependent on a given object (table or materialized view), use the procedure DBMS_MVIEW.GET_MV_DEPENDENCIES to determine the dependent materialized views for a given table, or for deciding the order to refresh nested materialized views.

```
DBMS_MVIEW.GET_MV_DEPENDENCIES(mvlist IN  VARCHAR2, deplist OUT  VARCHAR2)
```

The input to this function is the name or names of the materialized view. The output is a comma separated list of the materialized views that are defined on it. For example, the following statement:

```
DBMS_MVIEW.GET_MV_DEPENDENCIES("JOHN.SALES_REG, SCOTT.PROD_TIME", deplist)
```

This populates deplist with the list of materialized views defined on the input arguments. For example:

```
deplist <= "JOHN.SUM_SALES_WEST, JOHN.SUM_SALES_EAST, SCOTT.SUM_PROD_MONTH".
```

Using Job Queues for Refresh

Job queues can be used to refresh multiple materialized views in parallel. If queues are not available, fast refresh will sequentially refresh each view in the foreground process. To make queues available, you must set the JOB_QUEUE_PROCESSES parameter. This parameter defines the number of background job queue processes and determines how many materialized views can be refreshed concurrently. Oracle tries to balance the number of concurrent refreshes with the degree of parallelism of each refresh. The order in which the materialized views are refreshed is determined by dependencies imposed by nested materialized views and potential for efficient refresh by using query rewrite against other materialized views (See "Scheduling Refresh" on page 15-22 for details). This parameter is only effective when atomic_refresh is set to FALSE.

If the process that is executing DBMS_MVIEW.REFRESH is interrupted or the instance is shut down, any refresh jobs that were executing in job queue processes will be requeued and will continue running. To remove these jobs, use the DBMS_JOB.REMOVE procedure.

When Fast Refresh is Possible

Not all materialized views may be fast refreshable. Therefore, use the package DBMS_MVIEW.EXPLAIN_MVIEW to determine what refresh methods are available for a materialized view. See Chapter 8, "Basic Materialized Views" for further information about the DBMS_MVIEW package.

If you are not sure how to make a materialized view fast refreshable, you can use the DBMS_ADVISOR.TUNE_MVIEW procedure, which will provide a script containing the statements required to create a fast refreshable materialized view. See "Tuning Materialized Views for Fast Refresh and Query Rewrite" on page 17-47.

Recommended Initialization Parameters for Parallelism

The following initialization parameters need to be set properly for parallelism to be effective:

- PARALLEL_MAX_SERVERS should be set high enough to take care of parallelism. You need to consider the number of slaves needed for the refresh statement. For example, with a DOP of eight, you need 16 slave processes.
- PGA_AGGREGATE_TARGET should be set for the instance to manage the memory usage for sorts and joins automatically. If the memory parameters are set manually, SORT_AREA_SIZE should be less than HASH_AREA_SIZE.
- OPTIMIZER_MODE should equal all_rows.

Remember to analyze all tables and indexes for better optimization.

See Chapter 24, "Using Parallel Execution" for further information.

Monitoring a Refresh

While a job is running, you can query the V$SESSION_LONGOPS view to tell you the progress of each materialized view being refreshed.

```
SELECT * FROM V$SESSION_LONGOPS;
```

To look at the progress of which jobs are on which queue, use:

```
SELECT * FROM DBA_JOBS_RUNNING;
```

Checking the Status of a Materialized View

Three views are provided for checking the status of a materialized view: DBA_ MVEIWS, ALL_MVIEWS, and USER_MVIEWS. To check if a materialized view is fresh or stale, issue the following statement:

```
SELECT MVIEW_NAME, STALENESS, LAST_REFRESH_TYPE, COMPILE_STATE
FROM USER_MVIEWS ORDER BY MVIEW_NAME;

MVIEW_NAME            STALENESS      LAST_REF       COMPILE_STATE
----------            ---------      --------       -------------
CUST_MTH_SALES_MV     NEEDS_COMPILE  FAST           NEEDS_COMPILE
PROD_YR_SALES_MV      FRESH          FAST           VALID
```

If the compile_state column shows NEEDS COMPILE, the other displayed column values cannot be trusted as reflecting the true status. To revalidate the materialized view, issue the following statement:

```
ALTER MATERIALIZED VIEW [materialized_view_name] COMPILE;
```

Then reissue the SELECT statement.

Scheduling Refresh

Very often you will have multiple materialized views in the database. Some of these can be computed by rewriting against others. This is very common in data warehousing environment where you may have nested materialized views or materialized views at different levels of some hierarchy.

In such cases, you should create the materialized views as BUILD DEFERRED, and then issue one of the refresh procedures in DBMS_MVIEW package to refresh all the materialized views. Oracle Database will compute the dependencies and refresh the materialized views in the right order. Consider the example of a complete hierarchical cube described in "Examples of Hierarchical Cube Materialized Views" on page 20-32. Suppose all the materialized views have been created as BUILD DEFERRED. Creating the materialized views as BUILD DEFERRED will only create the metadata for all the materialized views. And, then, you can just call one of the refresh procedures in DBMS_MVIEW package to refresh all the materialized views in the right order:

```
EXECUTE DBMS_MVIEW.REFRESH_DEPENDENT(list=>'SALES', method => 'C');
```

The procedure will refresh the materialized views in the order of their dependencies (first sales_hierarchical_mon_cube_mv, followed by sales_ hierarchical_qtr_cube_mv, then, sales_hierarchical_yr_cube_mv and

finally, sales_hierarchical_all_cube_mv). Each of these materialized views will get rewritten against the one prior to it in the list).

The same kind of rewrite can also be used while doing PCT refresh. PCT refresh recomputes rows in a materialized view corresponding to changed rows in the detail tables. And, if there are other fresh materialized views available at the time of refresh, it can go directly against them as opposed to going against the detail tables.

Hence, it is always beneficial to pass a list of materialized views to any of the refresh procedures in DBMS_MVIEW package (irrespective of the method specified) and let the procedure figure out the order of doing refresh on materialized views.

Tips for Refreshing Materialized Views with Aggregates

Following are some guidelines for using the refresh mechanism for materialized views with aggregates.

- For fast refresh, create materialized view logs on all detail tables involved in a materialized view with the ROWID, SEQUENCE and INCLUDING NEW VALUES clauses.

  Include all columns from the table likely to be used in materialized views in the materialized view logs.

  Fast refresh may be possible even if the SEQUENCE option is omitted from the materialized view log. If it can be determined that only inserts or deletes will occur on all the detail tables, then the materialized view log does not require the SEQUENCE clause. However, if updates to multiple tables are likely or required or if the specific update scenarios are unknown, make sure the SEQUENCE clause is included.

- Use Oracle's bulk loader utility or direct-path INSERT (INSERT with the APPEND hint for loads).

  This is a lot more efficient than conventional insert. During loading, disable all constraints and re-enable when finished loading. Note that materialized view logs are required regardless of whether you use direct load or conventional DML.

  Try to optimize the sequence of conventional mixed DML operations, direct-path INSERT and the fast refresh of materialized views. You can use fast refresh with a mixture of conventional DML and direct loads. Fast refresh can perform significant optimizations if it finds that only direct loads have occurred, as illustrated in the following:

1. Direct-path INSERT (SQL*Loader or INSERT /*+ APPEND */) into the detail table
2. Refresh materialized view
3. Conventional mixed DML
4. Refresh materialized view

You can use fast refresh with conventional mixed DML (INSERT, UPDATE, and DELETE) to the detail tables. However, fast refresh will be able to perform significant optimizations in its processing if it detects that only inserts or deletes have been done to the tables, such as:

- DML INSERT or DELETE to the detail table
- Refresh materialized views
- DML update to the detail table
- Refresh materialized view

Even more optimal is the separation of INSERT and DELETE.

If possible, refresh should be performed after each type of data change (as shown earlier) rather than issuing only one refresh at the end. If that is not possible, restrict the conventional DML to the table to inserts only, to get much better refresh performance. Avoid mixing deletes and direct loads.

Furthermore, for refresh ON COMMIT, Oracle keeps track of the type of DML done in the committed transaction. Therefore, do not perform direct-path INSERT and DML to other tables in the same transaction, as Oracle may not be able to optimize the refresh phase.

For ON COMMIT materialized views, where refreshes automatically occur at the end of each transaction, it may not be possible to isolate the DML statements, in which case keeping the transactions short will help. However, if you plan to make numerous modifications to the detail table, it may be better to perform them in one transaction, so that refresh of the materialized view will be performed just once at commit time rather than after each update.

- Oracle recommends partitioning the tables because it enables you to use:
  - Parallel DML

    For large loads or refresh, enabling parallel DML will help shorten the length of time for the operation.
  - Partition Change Tracking (PCT) fast refresh

You can refresh your materialized views fast after partition maintenance operations on the detail tables. "Partition Change Tracking" on page 9-2 for details on enabling PCT for materialized views.

- Partitioning the materialized view will also help refresh performance as refresh can update the materialized view using parallel DML. For example, assume that the detail tables and materialized view are partitioned and have a parallel clause. The following sequence would enable Oracle to parallelize the refresh of the materialized view.

  1. Bulk load into the detail table.
  2. Enable parallel DML with an `ALTER SESSION ENABLE PARALLEL DML` statement.
  3. Refresh the materialized view.

- For refresh using `DBMS_MVIEW.REFRESH`, set the parameter `atomic_refresh` to `FALSE`.
  - For `COMPLETE` refresh, this will `TRUNCATE` to delete existing rows in the materialized view, which is faster than a delete.
  - For `PCT` refresh, if the materialized view is partitioned appropriately, this will use `TRUNCATE PARTITION` to delete rows in the affected partitions of the materialized view, which is faster than a delete.
  - For `FAST` or `FORCE` refresh, if `COMPLETE` or PCT refresh is chosen, this will be able to use the `TRUNCATE` optimizations described earlier.

- When using `DBMS_MVIEW.REFRESH` with `JOB_QUEUES`, remember to set `atomic` to `FALSE`. Otherwise, `JOB_QUEUES` will not get used. Set the number of job queue processes greater than the number of processors.

  If job queues are enabled and there are many materialized views to refresh, it is faster to refresh all of them in a single command than to call them individually.

- Use `REFRESH FORCE` to ensure refreshing a materialized view so that it can definitely be used for query rewrite. The best refresh method will be chosen. If a fast refresh cannot be done, a complete refresh will be performed.

- Refresh all the materialized views in a single procedure call. This gives Oracle an opportunity to schedule refresh of all the materialized views in the right order taking into account dependencies imposed by nested materialized views and potential for efficient refresh by using query rewrite against other materialized views.

Tips for Refreshing Materialized Views Without Aggregates

If a materialized view contains joins but no aggregates, then having an index on each of the join column rowids in the detail table will enhance refresh performance greatly, because this type of materialized view tends to be much larger than materialized views containing aggregates. For example, consider the following materialized view:

```
CREATE MATERIALIZED VIEW detail_fact_mv BUILD IMMEDIATE AS
SELECT s.rowid "sales_rid", t.rowid "times_rid", c.rowid "cust_rid",
   c.cust_state_province, t.week_ending_day, s.amount_sold
FROM sales s, times t, customers c
WHERE s.time_id = t.time_id AND s.cust_id = c.cust_id;
```

Indexes should be created on columns `sales_rid`, `times_rid` and `cust_rid`. Partitioning is highly recommended, as is enabling parallel DML in the session before invoking refresh, because it will greatly enhance refresh performance.

This type of materialized view can also be fast refreshed if DML is performed on the detail table. It is recommended that the same procedure be applied to this type of materialized view as for a single table aggregate. That is, perform one type of change (direct-path `INSERT` or DML) and then refresh the materialized view. This is because Oracle Database can perform significant optimizations if it detects that only one type of change has been done.

Also, Oracle recommends that the refresh be invoked after each table is loaded, rather than load all the tables and then perform the refresh.

For refresh `ON COMMIT`, Oracle keeps track of the type of DML done in the committed transaction. Oracle therefore recommends that you do not perform direct-path and conventional DML to other tables in the same transaction because Oracle may not be able to optimize the refresh phase. For example, the following is not recommended:

1. Direct load new data into the fact table
2. DML into the store table
3. Commit

Also, try not to mix different types of conventional DML statements if possible. This would again prevent using various optimizations during fast refresh. For example, try to avoid the following:

1. Insert into the fact table
2. Delete from the fact table

3. Commit

If many updates are needed, try to group them all into one transaction because refresh will be performed just once at commit time, rather than after each update.

When you use the DBMS_MVIEW package to refresh a number of materialized views containing only joins with the ATOMIC parameter set to TRUE, if you disable parallel DML, refresh performance may degrade.

In a data warehousing environment, assuming that the materialized view has a parallel clause, the following sequence of steps is recommended:

1. Bulk load into the fact table
2. Enable parallel DML
3. An ALTER SESSION ENABLE PARALLEL DML statement
4. Refresh the materialized view

Tips for Refreshing Nested Materialized Views

All underlying objects are treated as ordinary tables when refreshing materialized views. If the ON COMMIT refresh option is specified, then all the materialized views are refreshed in the appropriate order at commit time. In other words, Oracle builds a partially ordered set of materialized views and refreshes them such that, after the successful completion of the refresh, all the materialized views are fresh. The status of the materialized views can be checked by querying the appropriate USER_, DBA_, or ALL_MVIEWS view.

If any of the materialized views are defined as ON DEMAND refresh (irrespective of whether the refresh method is FAST, FORCE, or COMPLETE), you will need to refresh them in the correct order (taking into account the dependencies between the materialized views) because the nested materialized view will be refreshed with respect to the current contents of the other materialized views (whether fresh or not). This can be achieved by invoking the refresh procedure against the materialized view at the top of the nested hierarchy and specifying the nested parameter as TRUE.

If a refresh fails during commit time, the list of materialized views that has not been refreshed is written to the alert log, and you must manually refresh them along with all their dependent materialized views.

Use the same DBMS_MVIEW procedures on nested materialized views that you use on regular materialized views.

These procedures have the following behavior when used with nested materialized views:

- If REFRESH is applied to a materialized view my_mv that is built on other materialized views, then my_mv will be refreshed with respect to the current contents of the other materialized views (that is, the other materialized views will not be made fresh first) unless you specify nested => TRUE.
- If REFRESH_DEPENDENT is applied to materialized view my_mv, then only materialized views that directly depend on my_mv will be refreshed (that is, a materialized view that depends on a materialized view that depends on my_mv will not be refreshed) unless you specify nested => TRUE.
- If REFRESH_ALL_MVIEWS is used, the order in which the materialized views will be refreshed is guaranteed to respect the dependencies between nested materialized views.
- GET_MV_DEPENDENCIES provides a list of the immediate (or direct) materialized view dependencies for an object.

Tips for Fast Refresh with UNION ALL

You can use fast refresh for materialized views that use the UNION ALL operator by providing a maintenance column in the definition of the materialized view. For example, a materialized view with a UNION ALL operator can be made fast refreshable as follows:

```
CREATE MATERIALIZED VIEW fast_rf_union_all_mv AS
SELECT x.rowid AS r1, y.rowid AS r2, a, b, c, 1 AS marker
FROM x, y WHERE x.a = y.b
UNION ALL
SELECT p.rowid, r.rowid, a, c, d, 2 AS marker
FROM p, r WHERE p.a = r.y;
```

The form of a maintenance marker column, column MARKER in the example, must be numeric_or_string_literal AS column_alias, where each UNION ALL member has a distinct value for numeric_or_string_literal.

Tips After Refreshing Materialized Views

After you have performed a load or incremental load and rebuilt the detail table indexes, you need to re-enable integrity constraints (if any) and refresh the materialized views and materialized view indexes that are derived from that detail data. In a data warehouse environment, referential integrity constraints are

normally enabled with the NOVALIDATE or RELY options. An important decision to make before performing a refresh operation is whether the refresh needs to be recoverable. Because materialized view data is redundant and can always be reconstructed from the detail tables, it might be preferable to disable logging on the materialized view. To disable logging and run incremental refresh non-recoverably, use the ALTER MATERIALIZED VIEW ... NOLOGGING statement prior to refreshing.

If the materialized view is being refreshed using the ON COMMIT method, then, following refresh operations, consult the alert log alert_*SID*.log and the trace file ora_*SID*_number.trc to check that no errors have occurred.

Using Materialized Views with Partitioned Tables

A major maintenance component of a data warehouse is synchronizing (refreshing) the materialized views when the detail data changes. Partitioning the underlying detail tables can reduce the amount of time taken to perform the refresh task. This is possible because partitioning enables refresh to use parallel DML to update the materialized view. Also, it enables the use of Partition Change Tracking.

Fast Refresh with Partition Change Tracking

In a data warehouse, changes to the detail tables can often entail partition maintenance operations, such as DROP, EXCHANGE, MERGE, and ADD PARTITION. To maintain the materialized view after such operations in used to require manual maintenance (see also CONSIDER FRESH) or complete refresh. You now have the option of using an addition to fast refresh known as Partition Change Tracking (PCT) refresh.

For PCT to be available, the detail tables must be partitioned. The partitioning of the materialized view itself has no bearing on this feature. If PCT refresh is possible, it will occur automatically and no user intervention is required in order for it to occur. See "Partition Change Tracking" on page 9-2 for PCT requirements.

The following examples illustrate the use of this feature. In "PCT Fast Refresh Scenario 1", assume sales is a partitioned table using the time_id column and products is partitioned by the prod_category column. The table times is not a partitioned table.

PCT Fast Refresh Scenario 1

1. All detail tables must have materialized view logs. To avoid redundancy, only the materialized view log for the sales table is provided in the following:

```
CREATE MATERIALIZED VIEW LOG ON SALES WITH ROWID, SEQUENCE
  (prod_id, time_id, quantity_sold, amount_sold)
INCLUDING NEW VALUES;
```

2. The following materialized view satisfies requirements for PCT.

```
CREATE MATERIALIZED VIEW cust_mth_sales_mv
BUILD IMMEDIATE
REFRESH FAST ON DEMAND
ENABLE QUERY REWRITE AS
SELECT s.time_id, s.prod_id, SUM(s.quantity_sold), SUM(s.amount_sold),
       p.prod_name, t.calendar_month_name, COUNT(*),
       COUNT(s.quantity_sold), COUNT(s.amount_sold)
FROM sales s, products p, times t
WHERE  s.time_id = t.time_id AND s.prod_id = p.prod_id
GROUP BY t.calendar_month_name, s.prod_id, p.prod_name, s.time_id;
```

3. You can use the DBMS_MVIEW.EXPLAIN_MVIEW procedure to determine which tables will allow PCT refresh. See "Analyzing Materialized View Capabilities" on page 8-37 for how to use this procedure.

```
MVNAME               CAPABILITY_NAME   POSSIBLE  RELATED_TEXT  MSGTXT
-----------------    --------------    --------  ------------  ----------------
CUST_MTH_SALES_MV    PCT               Y         SALES
CUST_MTH_SALES_MV    PCT_TABLE         Y         SALES
CUST_MTH_SALES_MV    PCT_TABLE         N         PRODUCTS      no partition key or PMARKER
                                                                in SELECT list
CUST_MTH_SALES_MV    PCT_TABLE         N         TIMES         relation is not partitioned table
```

As can be seen from the partial sample output from EXPLAIN_MVIEW, any partition maintenance operation performed on the sales table will allow PCT fast refresh. However, PCT is not possible after partition maintenance operations or updates to the products table as there is insufficient information contained in cust_mth_sales_mv for PCT refresh to be possible. Note that the times table is not partitioned and hence can never allow for PCT refresh. Oracle will apply PCT refresh if it can determine that the materialized view has sufficient information to support PCT for all the updated tables.

1. Suppose at some later point, a SPLIT operation of one partition in the sales table becomes necessary.

```
ALTER TABLE SALES
SPLIT PARTITION month3 AT (TO_DATE('05-02-1998', 'DD-MM-YYYY'))
INTO (PARTITION month3_1 TABLESPACE summ,
      PARTITION month3 TABLESPACE summ);
```

2. Insert some data into the sales table.
3. Fast refresh cust_mth_sales_mv using the DBMS_MVIEW.REFRESH procedure.

```
EXECUTE DBMS_MVIEW.REFRESH('CUST_MTH_SALES_MV', 'F',
   '',TRUE,FALSE,0,0,0,FALSE);
```

Fast refresh will automatically do a PCT refresh as it is the only fast refresh possible in this scenario. However, fast refresh will not occur if a partition maintenance operation occurs when any update has taken place to a table on which PCT is not enabled. This is shown in "PCT Fast Refresh Scenario 2".

"PCT Fast Refresh Scenario 1" would also be appropriate if the materialized view was created using the PMARKER clause as illustrated in the following:

```
CREATE MATERIALIZED VIEW cust_sales_marker_mv
BUILD IMMEDIATE
REFRESH FAST ON DEMAND
ENABLE QUERY REWRITE AS
SELECT DBMS_MVIEW.PMARKER(s.rowid) s_marker, SUM(s.quantity_sold),
  SUM(s.amount_sold), p.prod_name, t.calendar_month_name, COUNT(*),
  COUNT(s.quantity_sold), COUNT(s.amount_sold)
FROM sales s, products p, times t
WHERE  s.time_id = t.time_id AND s.prod_id = p.prod_id
GROUP BY DBMS_MVIEW.PMARKER(s.rowid),
         p.prod_name, t.calendar_month_name;
```

PCT Fast Refresh Scenario 2

In "PCT Fast Refresh Scenario 2", the first four steps are the same as in "PCT Fast Refresh Scenario 1" on page 15-29. Then, the SPLIT partition operation to the sales table is performed, but before the materialized view refresh occurs, records are inserted into the times table.

1. The same as in "PCT Fast Refresh Scenario 1".
2. The same as in "PCT Fast Refresh Scenario 1".
3. The same as in "PCT Fast Refresh Scenario 1".
4. The same as in "PCT Fast Refresh Scenario 1".
5. After issuing the same SPLIT operation, as shown in "PCT Fast Refresh Scenario 1", some data will be inserted into the times table.

```
ALTER TABLE SALES
SPLIT PARTITION month3 AT (TO_DATE('05-02-1998', 'DD-MM-YYYY'))
```

```
INTO (PARTIITION month3_1 TABLESPACE summ,
      PARTITION month3 TABLESPACE summ);
```

6. Refresh `cust_mth_sales_mv`.

```
EXECUTE DBMS_MVIEW.REFRESH('CUST_MTH_SALES_MV', 'F',
    '',TRUE,FALSE,0,0,0,FALSE);
ORA-12052: cannot fast refresh materialized view SH.CUST_MTH_SALES_MV
```

The materialized view is not fast refreshable because DML has occurred to a table on which PCT fast refresh is not possible. To avoid this occurring, Oracle recommends performing a fast refresh immediately after any partition maintenance operation on detail tables for which partition tracking fast refresh is available.

If the situation in "PCT Fast Refresh Scenario 2" occurs, there are two possibilities; perform a complete refresh or switch to the `CONSIDER FRESH` option outlined in the following, if suitable. However, it should be noted that `CONSIDER FRESH` and partition change tracking fast refresh are not compatible. Once the `ALTER MATERIALIZED VIEW cust_mth_sales_mv CONSIDER FRESH` statement has been issued, PCT refresh will not longer be applied to this materialized view, until a complete refresh is done. Moreover, you should not use `CONSIDER FRESH` unless you have taken manual action to ensure that the materialized view is indeed fresh.

A common situation in a data warehouse is the use of rolling windows of data. In this case, the detail table and the materialized view may contain say the last 12 months of data. Every month, new data for a month is added to the table and the oldest month is deleted (or maybe archived). PCT refresh provides a very efficient mechanism to maintain the materialized view in this case.

PCT Fast Refresh Scenario 3

1. The new data is usually added to the detail table by adding a new partition and exchanging it with a table containing the new data.

```
ALTER TABLE sales ADD PARTITION month_new ...
ALTER TABLE sales EXCHANGE PARTITION month_new month_new_table
```

2. Next, the oldest partition is dropped or truncated.

```
ALTER TABLE sales DROP PARTITION month_oldest;
```

3. Now, if the materialized view satisfies all conditions for PCT refresh.

```
EXECUTE DBMS_MVIEW.REFRESH('CUST_MTH_SALES_MV', 'F', '', TRUE,
FALSE,0,0,0,FALSE);
```

Fast refresh will automatically detect that PCT is available and perform a PCT refresh.

Fast Refresh with CONSIDER FRESH

In a data warehouse, you may often wish to accumulate historical information in the materialized view even though this information is no longer in the detailed tables. In this case, you could maintain the materialized view using the ALTER MATERIALIZED VIEW *materialized_view_name* CONSIDER FRESH statement.

Note that CONSIDER FRESH declares that the contents of the materialized view are FRESH (in sync with the detail tables). Care must be taken when using this option in this scenario in conjunction with query rewrite because you may see unexpected results.

After using CONSIDER FRESH in an historical scenario, you will be able to apply traditional fast refresh after DML and direct loads to the materialized view, but not PCT fast refresh. This is because if the detail table partition at one time contained data that is currently kept in aggregated form in the materialized view, PCT refresh in attempting to resynchronize the materialized view with that partition could delete historical data which cannot be recomputed.

Assume the sales table stores the prior year's data and the cust_mth_sales_mv keeps the prior 10 years of data in aggregated form.

1. Remove old data from a partition in the sales table:

   ```
   ALTER TABLE sales TRUNCATE PARTITION month1;
   ```

   The materialized view is now considered stale and requires a refresh because of the partition operation. However, as the detail table no longer contains all the data associated with the partition fast refresh cannot be attempted.

2. Therefore, alter the materialized view to tell Oracle to consider it fresh.

   ```
   ALTER MATERIALIZED VIEW cust_mth_sales_mv CONSIDER FRESH;
   ```

   This statement informs Oracle that cust_mth_sales_mv is fresh for your purposes. However, the materialized view now has a status that is neither known fresh nor known stale. Instead, it is UNKNOWN. If the materialized view has query rewrite enabled in QUERY_REWRITE_INTEGRITY=stale_tolerated mode, it will be used for rewrite.

3. Insert data into sales.

4. Refresh the materialized view.

```
EXECUTE DBMS_MVIEW.REFRESH('CUST_MTH_SALES_MV', 'F', '', TRUE,
FALSE,0,0,0,FALSE);
```

Because the fast refresh detects that only INSERT statements occurred against the sales table it will update the materialized view with the new data. However, the status of the materialized view will remain UNKNOWN. The only way to return the materialized view to FRESH status is with a complete refresh which, also will remove the historical data from the materialized view.

16

Change Data Capture

Change Data Capture efficiently identifies and captures data that has been added to, updated in, or removed from, Oracle relational tables and makes this change data available for use by applications or individuals. Change Data Capture is provided as a database component beginning with Oracle9*i*.

This chapter describes Change Data Capture in the following sections:

See *PL/SQL Packages and Types Reference* for reference information about the Change Data Capture publish and subscribe PL/SQL packages.

Overview of Change Data Capture

Often, data warehousing involves the extraction and transportation of relational data from one or more production databases into a data warehouse for analysis. Change Data Capture quickly identifies and processes only the data that has changed and makes the change data available for further use.

Capturing Change Data Without Change Data Capture

Prior to the introduction of Change Data Capture, there were a number of ways that users could capture change data, including table differencing and change-value selection.

Table differencing involves transporting a copy of an entire table from the source (production) database to the staging database (where the change data is captured), where an older version of the table already exists. Using the SQL `MINUS` operator, you can obtain the inserted and new versions of updated rows with the following query:

```
SELECT * FROM new_version
MINUS SELECT * FROM old_version;
```

Moreover, you can obtain the deleted rows and old versions of updated rows with the following query:

```
SELECT * FROM old_version
MINUS SELECT * FROM new_version;
```

However, there are several problems with this method:

- It requires that the new version of the entire table be transported to the staging database, not just the change data, thereby greatly increasing transport costs.
- The computational cost of performing the two `MINUS` operations on the staging database can be very high.
- Table differencing cannot capture data changes that have reverted to their old values. For example, suppose the price of a product changes several times between the old version and the new version of the product's table. If the price in the new version ends up being the same as the old, table differencing cannot detect that the price has fluctuated. Moreover, any intermediate price values between the old and new versions of the product's table cannot be captured using table differencing.

- There is no way to determine which changes were made as part of the same transaction. For example, suppose a sales manager creates a special discount to close a deal. The fact that the creation of the discount and the creation of the sale occurred as part of the same transaction cannot be captured, unless the source database is specifically designed to do so.

Change-value selection involves capturing the data on the source database by selecting the new and changed data from the source tables based on the value of a specific column. For example, suppose the source table has a LAST_UPDATE_DATE column. To capture changes, you base your selection from the source table on the LAST_UPDATE_DATE column value.

However, there are also several problems with this method:

- The overhead of capturing the change data must be borne on the source database, and you must run potentially expensive queries against the source table on the source database. The need for these queries may force you to add indexes that would otherwise be unneeded. There is no way to offload this overhead to the staging database.
- This method is no better at capturing intermediate values than the table differencing method. If the price in the product's table fluctuates, you will not be able to capture all the intermediate values, or even tell if the price had changed, if the ending value is the same as it was the last time that you captured change data.
- This method is also no better than the table differencing method at capturing which data changes were made together in the same transaction. If you need to capture information concerning which changes occurred together in the same transaction, you must include specific designs for this purpose in your source database.
- The granularity of the change-value column may not be fine enough to uniquely identify the new and changed rows. For example, suppose the following:
  - You capture data changes using change-value selection on a date column such as LAST_UPDATE_DATE.
  - The capture happens at a particular instant in time, 14-FEB-2003 17:10:00.
  - Additional updates occur to the table during the same second that you performed your capture.

When you next capture data changes, you will select rows with a LAST_UPDATE_DATE strictly after 14-FEB-2003 17:10:00, and thereby miss the changes that occurred during the remainder of that second.

To use change-value selection, you either have to accept that anomaly, add an artificial change-value column with the granularity you need, or lock out changes to the source table during the capture process, thereby further burdening the performance of the source database.

- You have to design your source database in advance with this capture mechanism in mind – all tables from which you wish to capture change data must have a change-value column. If you want to build a data warehouse with data sources from legacy systems, those legacy systems may not supply the necessary change-value columns you need.

Change Data Capture does not depend on expensive and cumbersome table differencing or change-value selection mechanisms. Instead, it captures the change data resulting from INSERT, UPDATE, and DELETE operations made to user tables. The change data is then stored in a relational table called a change table, and the change data is made available to applications or individuals in a controlled way.

Capturing Change Data with Change Data Capture

Change Data Capture can capture and publish committed change data in either of the following modes:

- **Synchronous**

  Change data is captured immediately, as each SQL statement that performs a data manipulation language (DML) operation (INSERT, UPDATE, or DELETE) is made, by using triggers on the source database. In this mode, change data is captured as part of the transaction modifying the source table. Synchronous Change Data Capture is available with Oracle Standard Edition and Enterprise Edition.

- **Asynchronous**

  Change data is captured after a SQL statement that performs a DML operation is committed, by taking advantage of the data sent to the redo log files. In this mode, change data is not captured as part of the transaction that is modifying the source table, and therefore has no effect on that transaction. Asynchronous Change Data Capture is available with Oracle Enterprise Edition only.

Asynchronous Change Data Capture is built on, and provides a relational interface to, Oracle Streams. See *Oracle Streams Concepts and Administration* for information on Oracle Streams.

The following list describes the advantages of capturing change data with Change Data Capture:

- Completeness

  Change Data Capture can capture all effects of `INSERT`, `UPDATE`, and `DELETE` operations, including data values before and after `UPDATE` operations.

- Performance

  Asynchronous Change Data Capture can be configured to have minimal performance impact on the source database.

- Interface

  Change Data Capture includes the PL/SQL `DBMS_CDC_PUBLISH` and `DBMS_CDC_SUBSCRIBE` packages, which provide easy-to-use publish and subscribe interfaces.

- Cost

  Change Data Capture reduces overhead cost because it simplifies the extraction of change data from the database and is part of Oracle9*i* and later databases.

A Change Data Capture system is based on the interaction of a publisher and subscribers to capture and distribute change data, as described in the next section.

Publish and Subscribe Model

Most Change Data Capture systems have one person who captures and publishes change data; this person is the **publisher**. There can be multiple applications or individuals that access the change data; these applications and individuals are the **subscribers**. Change Data Capture provides PL/SQL packages to accomplish the publish and subscribe tasks.

The following sections describe the roles of the publisher and subscriber in detail. Subsequent sections describe change sources, more about modes of Change Data Capture, and change tables.

Publisher

The publisher is usually a database administrator (DBA) who creates and maintains the schema objects that make up the Change Data Capture system. Typically, a publisher deals with two databases:

- **Source database**

  This is the production database that contains the data of interest. Its associated tables are referred to as the **source tables**.

- **Staging database**

  This is the database where the change data capture takes place. Depending on the capture mode that the publisher uses, the staging database can be the same as, or different from, the source database. The following Change Data Capture objects reside on the staging database:

  – Change table

    A **change table** is a relational table that contains change data for a single source table. To subscribers, a change table is known as a **publication**.

  – Change set

    A **change set** is a set of change data that is guaranteed to be transactionally consistent. It contains one or more change tables.

  – Change source

    The **change source** is a logical representation of the source database. It contains one or more change sets.

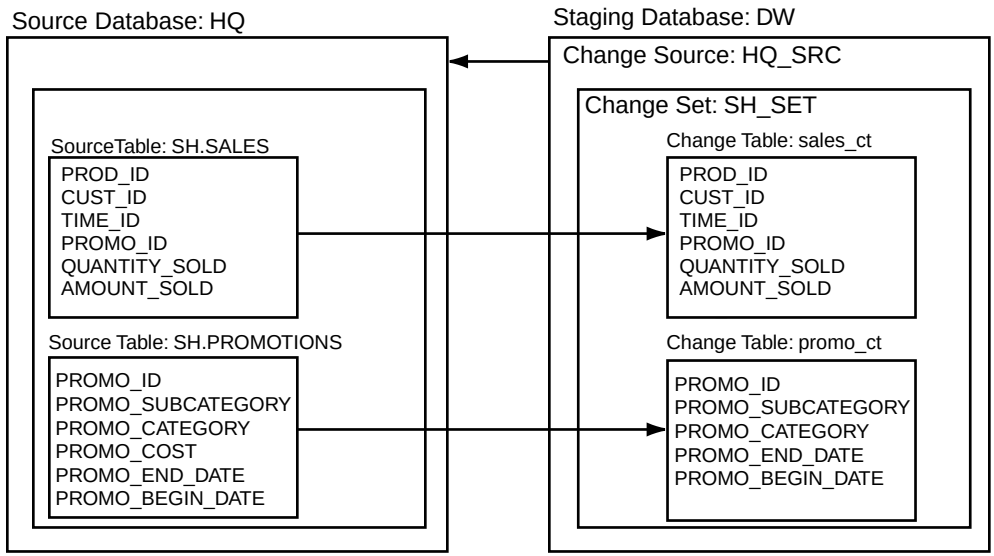
The publisher performs these tasks:

- Determines the source databases and tables from which the subscribers are interested in viewing change data, and the mode (synchronous or asynchronous) in which to capture the change data.
- Uses the Oracle-supplied package, `DBMS_CDC_PUBLISH`, to set up the system to capture change data from the source tables of interest.
- Allows subscribers to have controlled access to the change data in the change tables by using the SQL `GRANT` and `REVOKE` statements to grant and revoke the `SELECT` privilege on change tables for users and roles. (Keep in mind, however, that subscribers use views, not change tables directly, to access change data.)

In Figure 16–1, the publisher determines that subscribers are interested in viewing change data from the HQ source database. In particular, subscribers are interested in change data from the SH.SALES and SH.PROMOTIONS source tables.

The publisher creates a change source HQ_SRC on the DW staging database, a change set, SH_SET, and two change tables: sales_ct and promo_ct. The sales_ct change table contains all the columns from the source table, SH.SALES. For the promo_ct change table, however, the publisher has decided to exclude the PROMO_COST column.

Figure 16–1 Publisher Components in a Change Data Capture System



Subscribers

The subscribers are consumers of the published change data. A subscriber performs the following tasks:

- Uses the Oracle supplied package, DBMS_CDC_SUBSCRIBE, to:
  - Create subscriptions

    A **subscription** controls access to the change data from one or more source tables of interest within a single change set. A subscription contains one or more subscriber views.

A **subscriber view** is a view that specifies the change data from a specific publication in a subscription. The subscriber is restricted to seeing change data that the publisher has published and has granted the subscriber access to use. See "Subscribing to Change Data" on page 16-42 for more information on choosing a method for specifying a subscriber view.

  - Notify Change Data Capture when ready to receive a set of change data

    A **subscription window** defines the time range of rows in a publication that the subscriber can currently see in subscriber views. The oldest row in the window is called the **low boundary**; the newest row in the window is called the **high boundary**. Each subscription has its own subscription window that applies to all of its subscriber views.

  - Notify Change Data Capture when finished with a set of change data

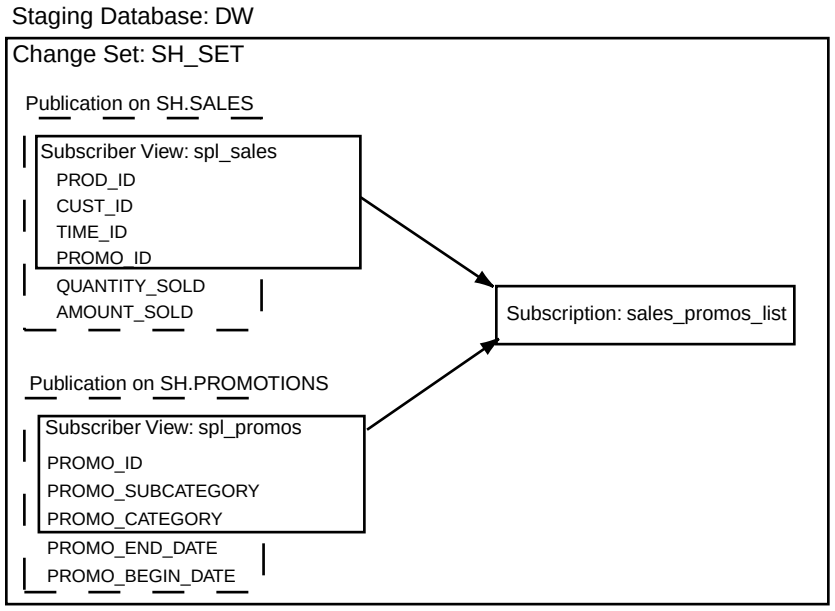
- Uses `SELECT` statements to retrieve change data from the subscriber views.

A subscriber has the privileges of the user account under which the subscriber is running, plus any additional privileges that have been granted to the subscriber.

In Figure 16–2, the subscriber is interested in a subset of columns that the publisher (in Figure 16–1) has published. Note that the *publications* shown in Figure 16–2, are represented as *change tables* in Figure 16–1; this reflects the different terminology used by subscribers and publishers, respectively.

The subscriber creates a subscription, `sales_promos_list` and two subscriber views (`spl_sales` and `spl_promos`) on the `SH_SET` change set on the DW staging database. Within each subscriber view, the subscriber includes a subset of the columns that were made available by the publisher. Note that because the publisher did not create a change table that includes the `PROMO_COST` column, there is no way for the subscriber to view change data for that column.

Figure 16–2 Subscriber Components in a Change Data Capture System



Change Data Capture provides the following benefits for subscribers:

- Guarantees that each subscriber sees all the changes
- Keeps track of multiple subscribers and gives each subscriber shared access to change data
- Handles all the storage management by automatically removing data from change tables when it is no longer required by any of the subscribers

> **Note:** Oracle provides the previously listed benefits only when the subscriber accesses change data through a subscriber view.

Change Sources and Modes of Data Capture

Change Data Capture provides synchronous and asynchronous modes for capturing change data. The following sections summarize how each mode of Change Data Capture is performed, and the change source associated with each mode of Change Data Capture.

Synchronous

This mode uses triggers on the source database to capture change data. It has no latency because the change data is captured continuously and in real time on the source database. The change tables are populated when DML operations on the source table are committed.

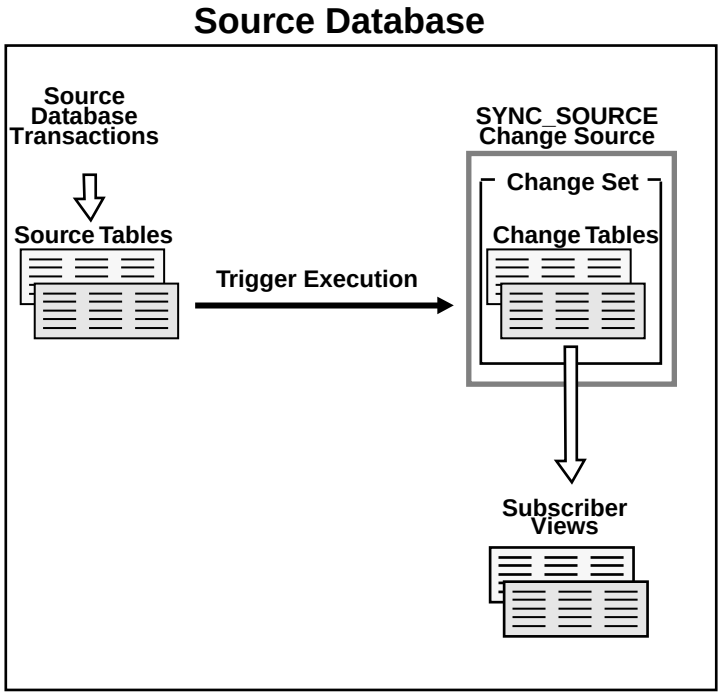
While the synchronous mode of Change Data Capture adds overhead to the source database at capture time, this mode can reduce costs (as compared to attempting to extract change data using table differencing or change-value section) by simplifying the extraction of change data.

There is a single, predefined synchronous change source, SYNC_SOURCE, that represents the source database. This is the only synchronous change source. It cannot be altered or dropped.

Change tables for this mode of Change Data Capture must reside locally in the source database.

Figure 16–3 illustrates the synchronous configuration. Triggers executed after DML operations occur on the source tables populate the change tables in the change sets within the SYNC_SOURCE change source.

Figure 16–3 Synchronous Change Data Capture Configuration



Asynchronous

This mode captures change data after the changes have been committed to the source database by using the database redo log files.

The asynchronous mode of Change Data Capture is dependent on the level of supplemental logging enabled at the source database. Supplemental logging adds redo logging overhead at the source database, so it must be carefully balanced with the needs of the applications or individuals using Change Data Capture. See "Asynchronous Change Data Capture and Supplemental Logging" on page 16-50 for information on supplemental logging.

There are two methods of capturing change data asynchronously, HotLog and AutoLog, as described in the following sections:

HotLog

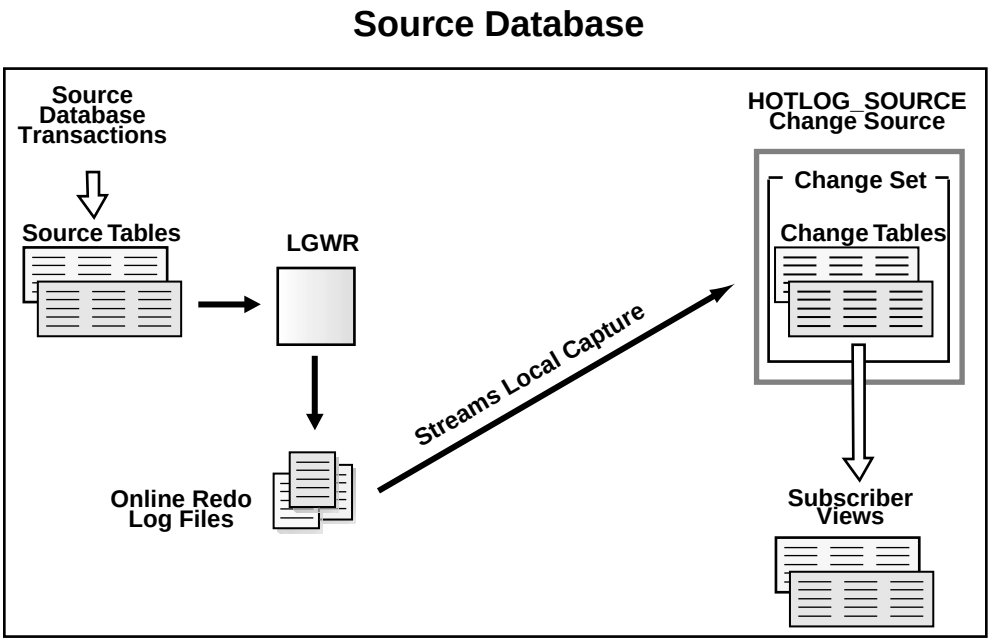
Change data is captured from the online redo log file on the source database. There is a brief latency between the act of committing source table transactions and the arrival of change data.

There is a single, predefined HotLog change source, HOTLOG_SOURCE, that represents the current redo log files of the source database. This is the only HotLog change source. It cannot be altered or dropped.

Change tables for this mode of Change Data Capture must reside locally in the source database.

Figure 16–4, illustrates the asynchronous HotLog configuration. The Logwriter Process (LGWR) records committed transactions in the online redo log files on the source database. Change Data Capture uses Oracle Streams processes to automatically populate the change tables in the change sets within the HOTLOG_SOURCE change source as newly committed transactions arrive.

Figure 16–4 Asynchronous HotLog Configuration



AutoLog

Change data is captured from a set of redo log files managed by log transport services. **Log transport services** control the automated transfer of redo log files from the source database to the staging database. Using database initialization parameters (described in "Initialization Parameters for Asynchronous AutoLog Publishing" on page 16-22), the publisher configures log transport services to copy the redo log files from the source database system to the staging database system and to automatically register the redo log files. Change sets are populated automatically as new redo log files arrive. The degree of latency depends on frequency of redo log switches on the source database.

There is no predefined AutoLog change source. The publisher provides information about the source database to create an AutoLog change source. See "Performing Asynchronous AutoLog Publishing" on page 16-35 for details.

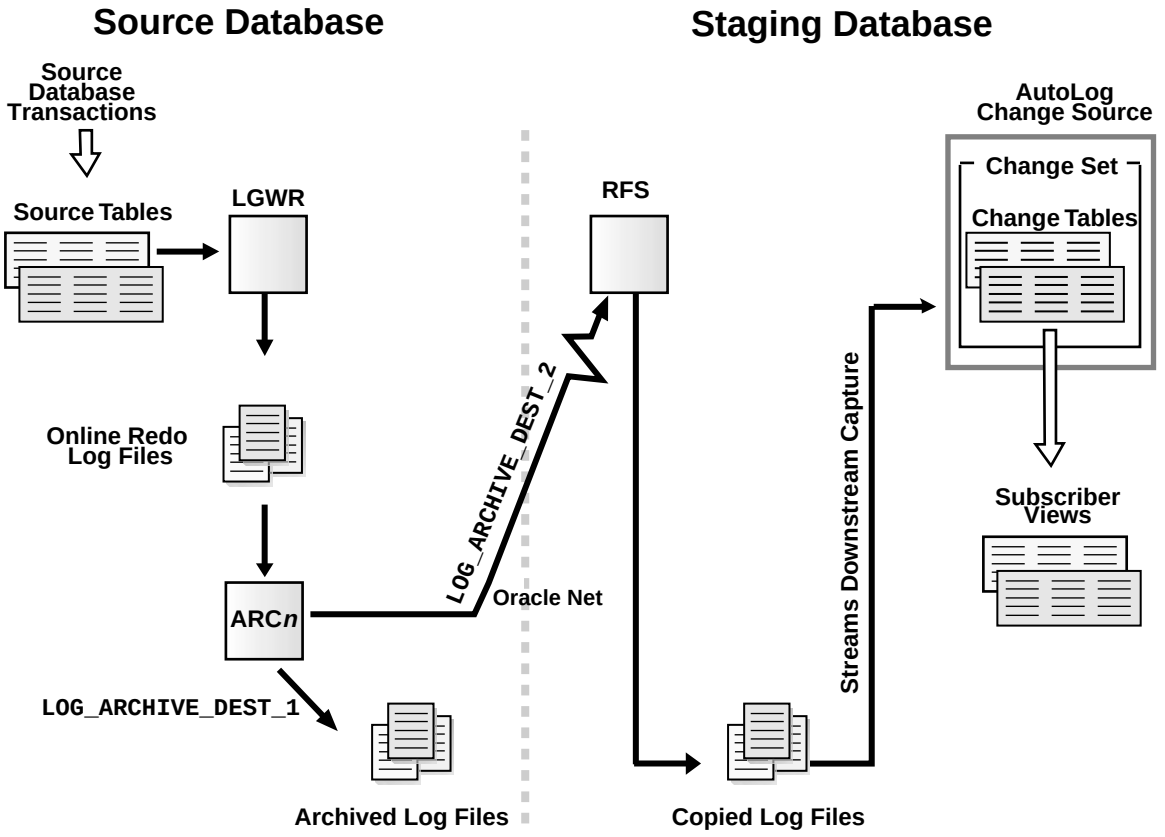
Change sets for this mode of Change Data Capture can be remote from or local to the source database. Typically, they are remote.

Figure 16–5 shows a typical Change Data Capture asynchronous AutoLog configuration in which, when the log switches on the source database, archiver processes archive the redo log file on the source database to the destination specified by the `LOG_ARCHIVE_DEST_1` parameter and copy the redo log file to the staging database as specified by the `LOG_ARCHIVE_DEST_2` parameter. (Although the image presents these parameters as `LOG_ARCHIVE_DEST_1` and `LOG_ARCHIVE_DEST_2`, the integer value in these parameter strings can be any value between 1 and 10.)

Note that the archiver processes use Oracle Net to send redo data over the network to the remote file server (RFS) process. Transmitting redo log files to a remote destination requires uninterrupted connectivity through Oracle Net.

On the staging database, the RFS process writes the redo data to the copied log files in the location specified by the value of the `TEMPLATE` attribute in the `LOG_ARCHIVE_DEST_2` parameter (specified in the source database initialization parameter file). Then, Change Data Capture uses Oracle Streams downstream capture to populate the change tables in the change sets within the AutoLog change source.

Figure 16–5 Asynchronous AutoLog Change Data Capture Configuration



Change Sets

A **change set** is a logical grouping of change data that is guaranteed to be transactionally consistent and that can be managed as a unit. A change set is a member of one (and only one) change source. A change source can contain one or more change sets. Conceptually, a change set shares the same mode as its change source. For example, an AutoLog change set is a change set contained in an AutoLog change source.

When a publisher includes two or more change tables in the same change set, subscribers can perform join operations across the tables represented within the change set and be assured of transactional consistency.

There are three modes of change sets, as follow:

- Synchronous

  New change data arrives automatically as DML operations on the source tables are committed. Publishers can define new change sets in the predefined SYNC_SOURCE change source or use the predefined change set, SYNC_SET. The SYNC_SET change set cannot be altered or dropped.

- Asynchronous HotLog

  New change data arrives automatically, on a transaction-by-transaction basis from the current online redo log file. Publishers define change sets in the predefined HOTLOG_SOURCE change source.

- Asynchronous AutoLog

  New change data arrives automatically, on a log-by-log basis, as log transport services makes redo log files available. Publishers define change sets in publisher-defined AutoLog change sources.

Publishers can purge unneeded change data from change tables at the change set level to keep the change tables in the change set from growing larger indefinitely.

See "Purging Change Tables of Unneeded Data" on page 16-65 for more information on purging change data.

Valid Combinations of Change Sources and Change Sets

Table 16–1 summarizes the valid combinations of change sources and change sets and indicates whether each is predefined or publisher-defined. In addition, Table 16–1 indicates whether the change source represents a local or remote source database, and whether the change source is used for synchronous or asynchronous Change Data Capture.

Table 16–1 Summary of Change Sources and Change Sets

Mode	Change Source	Source Database Represented	Associated Change Sets
Synchronous	Predefined SYNC_SOURCE	Local	Predefined SYNC_SET and publisher-defined
Asynchronous HotLog	Predefined HOTLOG_SOURCE	Local	Publisher-defined
Asynchronous AutoLog	Publisher-defined	Remote or local	Publisher-defined

Note: Although the AutoLog source database and AutoLog staging database can be the same, this arrangement is rarely used. AutoLog examples in this chapter assume that the source and staging databases are different.

Change Tables

A given change table contains the change data resulting from DML operations performed on a given source table. A change table consists of two things: the change data itself, which is stored in a database table; and the system metadata necessary to maintain the change table, which includes control columns.

The publisher specifies the source columns that are to be included in the change table. Typically, for a change table to contain useful data, the publisher needs to include the primary key column in the change table along with any other columns of interest to subscribers. For example, suppose subscribers are interested in changes that occur to the `UNIT_COST` and the `UNIT_PRICE` columns in the `SH.COSTS` table. If the publisher does not include the `PROD_ID` column in the change table, subscribers will know only that the unit cost and unit price of some products have changed, but will be unable to determine for which products these changes have occurred.

There are optional and required control columns. The required control columns are always included in a change table; the optional ones are included if specified by the publisher when creating the change table. Control columns are managed by Change Data Capture. See "Understanding Change Table Control Columns" on page 16-60 and "Understanding TARGET_COLMAP$ and SOURCE_COLMAP$ Values" on page 16-62 for detailed information on control columns.

Getting Information About the Change Data Capture Environment

Information about the Change Data Capture environment is provided in the static data dictionary views described in Table 16–2 and Table 16–3. Table 16–2 lists the views that are intended for use by publishers; the user must have the `SELECT_CATALOG_ROLE` privilege to access the views listed in this table. Table 16–3 lists the views that are intended for use by subscribers. Table 16–3 includes views with the prefixes `ALL` and `USER`. These prefixes have the following general meanings:

- A view with the `ALL` prefix allows the user to display all the information accessible to the user, including information from the current user's schema as

well as information from objects in other schemas, if the current user has access to those objects by way of grants of privileges or roles.

- A view with the USER prefix allows the user to display all the information from the schema of the user issuing the query without the use of additional special privileges or roles.

> **Note:** To look at all the views (those intended for both the publisher and the subscriber), a user must have the SELECT_CATALOG_ROLE privilege.

Table 16–2 Views Intended for Use by Change Data Capture Publishers

View Name	Description
CHANGE_SOURCES	Describes existing change sources.
CHANGE_SETS	Describes existing change sets.
CHANGE_TABLES	Describes existing change tables.
DBA_SOURCE_TABLES	Describes all existing source tables in the database.
DBA_PUBLISHED_COLUMNS	Describes all published columns of source tables in the database.
DBA_SUBSCRIPTIONS	Describes all subscriptions.
DBA_SUBSCRIBED_TABLES	Describes all source tables to which any subscriber has subscribed.
DBA_SUBSCRIBED_COLUMNS	Describes the columns of source tables to which any subscriber has subscribed.

Table 16–3 Views Intended for Use by Change Data Capture Subscribers

View Name	Description
ALL_SOURCE_TABLES	Describes all existing source tables accessible to the current user.
USER_SOURCE_TABLES	Describes all existing source tables owned by the current user.
ALL_PUBLISHED_COLUMNS	Describes all published columns of source tables accessible to the current user.
USER_PUBLISHED_COLUMNS	Describes all published columns of source tables owned by the current user.
ALL_SUBSCRIPTIONS	Describes all subscriptions accessible to the current user.

Table 16–3 (Cont.) Views Intended for Use by Change Data Capture Subscribers

View Name	Description
USER_SUBSCRIPTIONS	Describes all the subscriptions owned by the current user.
ALL_SUBSCRIBED_TABLES	Describes the source tables to which any subscription accessible to the current user has subscribed.
USER_SUBSCRIBED_TABLES	Describes the source tables to which the current user has subscribed.
ALL_SUBSCRIBED_COLUMNS	Describes the columns of source tables to which any subscription accessible to the current user has subscribed.
USER_SUBSCRIBED_COLUMNS	Describes the columns of source tables to which the current user has subscribed.

See *Oracle Database Reference* for complete information about these views.

Preparing to Publish Change Data

This section describes the tasks the publisher should perform before starting to publish, information on creating the publisher, information on selecting a mode in which to capture change data, and instructions on setting database initialization parameters required by Change Data Capture.

The publisher should do the following before performing the actual steps for publishing:

- Gather requirements from the subscribers.
- Determine which source database contains the relevant source tables.
- Choose the capture mode: synchronous, asynchronous HotLog, or asynchronous AutoLog, as described in "Determining the Mode in Which to Capture Data" on page 16-20.
- Ensure that the source and staging database DBAs have set database initialization parameters, as described in"Setting Initialization Parameters for Change Data Capture Publishing" on page 16-21 and "Publishing Change Data" on page 16-27.

Creating a User to Serve As a Publisher

Typically, a DBA creates a user to serve as a publisher for Change Data Capture. The following sections describe the tasks involved.

Granting Privileges and Roles to the Publisher

On the staging database, the publisher must be granted the privileges and roles in the following list:

- EXECUTE_CATALOG_ROLE privilege
- SELECT_CATALOG_ROLE privilege
- CREATE TABLE and CREATE SESSION privileges
- CONNECT and RESOURCE roles

In addition, for asynchronous HotLog and AutoLog publishing, the publisher must:

- Be granted the CREATE SEQUENCE privilege
- Be granted the DBA role
- Be the GRANTEE specified in a DBMS_STREAMS_AUTH.GRANT_ADMIN_PRIVILEGE() subprogram issued by the staging database DBA

In other words, for asynchronous publishing, the publisher should be configured as an Oracle Streams administrator. See Oracle Streams Concepts and Administration for information on configuring an Oracle Streams administrator.

Creating a Default Tablespace for the Publisher

When creating the publisher account, the DBA should also consider specifying a default tablespace for the publisher in which he or she can create change tables. Otherwise, Oracle recommends that the publisher always specify a tablespace that he or she owns when creating a change table. To create a tablespace, the publisher will need the following privileges:

- CREATE TABLESPACE
- UNLIMITED TABLESPACE

Password Files and Setting the REMOTE_LOGIN_PASSWORDFILE Parameter

For asynchronous AutoLog publishing only, the REMOTE_LOGIN_PASSWORDFILE database initialization parameter must be set to SHARED on both the source and staging databases, and a password file must exist on both the machines hosting the source and staging databases. The DBA uses the orapwd utility to create password files. For example:

```
ORAPWD FILE=orapw PASSWORD=mypassword ENTRIES=10
```

This example creates a password file with 10 entries, where the password for SYS is **mypassword.** For redo log file transmission to succeed, the password for the SYS user account *must be identical* for the source and staging databases.

Determining the Mode in Which to Capture Data

Three factors influence the decision on the mode in which to capture change data:

- Whether or not the staging database is remote from the source database
- Tolerance for latency between changes made on the source database and changes captured by Change Data Capture
- Performance impact on the source database transactions and overall database performance

Table 16–4 summarizes the factors that might influence the mode decision.

Table 16–4 Factors Influencing Choice of Change Data Capture Mode

Mode	Location of Staging Database	Latency	Source Database Performance Impact
Synchronous	Must be the same as the source database.	None - change data is automatically committed as part of the same transaction it reflects.	Adds overhead to source database transactions to perform change data capture.
Asynchronous HotLog	Must be the same as the source database.	Change data is captured from the current online redo log file. Change sets are populated automatically as new committed transactions arrive.	Minimal impact on source database transactions to perform supplemental logging. Additional source database overhead to perform change data capture.
Asynchronous AutoLog	Typically remote from the source database.	Depends on the frequency of redo log switches on the source database. Change sets are populated automatically as new log files arrive.	Minimal impact on source database transactions to perform supplemental logging. Minimal source database overhead for log transport services. When the source database is remote from the staging database, then this mode has the least impact on the source database.

Setting Initialization Parameters for Change Data Capture Publishing

Initialization parameters must be set on the source or staging database, or both, for Change Data Capture to succeed. Which parameters to set depend on the mode in which Change Data Capture is publishing change data, and on whether the parameters are being set on the source or staging database.

The following sections describe the database initialization parameter settings for each mode of Change Data Capture. Sometimes you are directed to add a value to a current setting. See "Determining the Current Setting of an Initialization Parameter" on page 16-25 to determine the current setting for a parameter if that parameter is not explicitly specified in the initialization parameter file. See *Oracle Database Reference* and *Oracle Data Guard Concepts and Administration* for more information about these database initialization parameters.

Initialization Parameters for Synchronous Publishing

Set the JAVA_POOL_SIZE parameter as follows:

```
JAVA_POOL_SIZE = 50000000
```

Initialization Parameters for Asynchronous HotLog Publishing

Table 16–5 lists the source database initialization parameters and their recommended settings for Asychronous HotLog publishing.

Table 16–5 Source Database Initialization Parameters for Asynchronous HotLog Publishing

Parameter	Recommended Value
COMPATIBLE	10.1.0
JAVA_POOL_SIZE	50000000
JOB_QUEUE_PROCESSES	2
PARALLEL_MAX_SERVERS	(current value) + (5 * (the number of change sets planned))
PROCESSES	(current value) + (7 * (the number of change sets planned))

Table 16–5 (Cont.) Source Database Initialization Parameters for Asynchronous HotLog Publishing

Parameter	Recommended Value
SESSIONS	(current value) + (2 * (the number of change sets planned))
STREAMS_POOL_SIZE	■ If the current value of the STREAMS_POOL_SIZE parameter is 50 MB or greater, then set this parameter to: (current value) + ((the number of change sets planned) * (21 MB)) ■ If the current value of the STREAMS_POOL_SIZE parameter is less than 50 MB, then set the value of this parameter to: 50 MB + ((the number of change sets planned) * (21 MB)) See *Oracle Streams Concepts and Administration* for information on how the STREAMS_POOL_SIZE parameter is applied when changed dynamically.
UNDO_RETENTION	3600

Initialization Parameters for Asynchronous AutoLog Publishing

Table 16–6 lists the database initialization parameters and their recommended settings for the asynchronous AutoLog publishing source database and Table 16–7 lists the database initialization parameters and their recommended settings for the asynchronous AutoLog publishing staging database.

Table 16–6 Source Database Initialization Parameters for Asynchronous AutoLog Publishing

Parameter	Recommended Value
COMPATIBLE	10.1.0
JAVA_POOL_SIZE	50000000
LOG_ARCHIVE_DEST_1[1]	The directory specification on the system hosting the source database where the archived redo log files are to be kept.

Table 16–6 (Cont.) Source Database Initialization Parameters for Asynchronous AutoLog Publishing

Parameter	Recommended Value
LOG_ARCHIVE_DEST_2[1]	This parameter must include the SERVICE, ARCH or LGWR ASYNC, OPTIONAL, NOREGISTER, and REOPEN attributes so that log transport services are configured to copy the redo log files from the source database to the staging database. These attributes are set as follows: ■ SERVICE specifies the network name of the staging database. ■ ARCH or LGWR ASYNC ARCH specifies that the archiver process (ARCn) copy the redo log files to the staging database after a source database log switch occurs. LGWR ASYNC specifies that the log writer process (LGWR) copy redo data to the staging database as the redo is generated on the source database. Note that, the copied redo data becomes available to Change Data Capture only after a source database log switch occurs. ■ OPTIONAL specifies that the copying of a redo log file to the staging database need not succeed before the corresponding online redo log at the source database can be overwritten. This is needed to avoid stalling operations on the source database due to a transmission failure to the staging database. The original redo log file remains available to the source database in either archived or backed up form, if it is needed. ■ NOREGISTER specifies that the staging database location is not recorded in the staging database control file. ■ REOPEN specifies the minimum number of seconds the archiver process (ARCn) should wait before trying to access the staging database if a previous attempt to access this location failed. ■ TEMPLATE defines a directory specification and a format template for the file name used for the redo log files that are copied to the staging database.[2]
LOG_ARCHIVE_DEST_STATE_1[1]	ENABLE Indicates that log transport services can transmit archived redo log files to this destination.
LOG_ARCHIVE_DEST_STATE_2[1]	ENABLE Indicates that log transport services can transmit redo log files to this destination.
LOG_ARCHIVE_FORMAT	"arch1_%s_%t_%r.dbf" Specifies a format template for the default file name when archiving redo log files.[2] The string value (arch1) and the file name extension (.dbf) do not have to be exactly as specified here.
REMOTE_LOGIN_PASSWORDFILE	SHARED

[1] The integer value in this parameter can be any value between 1 and 10. In this manual, the values 1 and 2 are used. For each LOG_ARCHIVE_DEST_*n* parameter, there must be a corresponding LOG_ARCHIVE_DEST_STATE_*n* parameter that specifies the same value for **n**.

[2] In the format template, %t corresponds to the thread number, %s corresponds to the sequence number, and %r corresponds to the resetlogs ID. Together, these ensure that unique names are constructed for the copied redo log files.

Table 16–7 Staging Database Initialization Parameters for Asynchronous AutoLog Publishing

Parameter	Recommended Value
COMPATIBLE	10.1.0
GLOBAL_NAMES	TRUE
JAVA_POOL_SIZE	50000000
JOB_QUEUE_PROCESSES	2
PARALLEL_MAX_SERVERS	(current value) + (5 * (the number of change sets planned))
PROCESSES	(current value) + (7 * (the number of change sets planned))
REMOTE_LOGIN_PASSWORDFILE	SHARED
SESSIONS	(current value)+ (2 * (the number of change sets planned))
STREAMS_POOL_SIZE	▪ If the current value of the STREAMS_POOL_SIZE parameter is 50 MB or greater, then set this parameter to: (current value) + ((the number of change sets planned) * (21 MB)) ▪ If the current value of the STREAMS_POOL_SIZE parameter is less than 50 MB, then set the value of this parameter to: 50 MB + ((the number of change sets planned) * (21 MB)) See *Oracle Streams Concepts and Administration* for information on how the STREAMS_POOL_SIZE parameter is applied when changed dynamically.
UNDO_RETENTION	3600

Note: If Oracle Streams capture parallelism and apply parallelism values are increased after the change sets are created, the source or staging database DBA (depending on the mode of capture) must adjust initialization parameter values as described in "Adjusting Initialization Parameter Values When Oracle Streams Values Change" on page 16-25.

Determining the Current Setting of an Initialization Parameter

Sometimes the DBA needs to make adjustments to the initialization parameters by adding a value to the current setting. If a specific parameter is not in the initialization parameter file, the default value (if one exists) is the current value. The current settings of all initialization parameters for the database are displayed when the following SQL statement is issued:

```
SQL> SHOW PARAMETERS
```

The current setting of a particular initialization parameter can be displayed, in this case, STREAMS_POOL_SIZE, using a SQL statement such as the following:

```
SQL> SHOW PARAMETER STREAMS_POOL_SIZE
```

Retaining Initialization Parameter Values When a Database Is Restarted

To ensure that changed initialization parameter values are retained when the database is restarted:

- If the database is using a pfile, manually update the pfile with any parameter values you change with the SQL ALTER SYSTEM statement.
- If the database is using an spfile, then parameter values you change with the SQL ALTER SYSTEM statement are automatically changed in the parameter file.
- If the database is using an spfile and a given initialization parameter is not or cannot be changed dynamically (such as the PROCESS, LOG_ARCHIVE_FORMAT, and REMOTE_LOGIN_ENABLE parameters), you must change the value with the SQL ALTER SYSTEM statement, and then restart the database.
- If the ORA-04031 error is returned when the DBA attempts to set the JAVA_POOL_SIZE dynamically, he or she should place the parameter in the database initialization parameter file and restart the database.

See *Oracle Database Administrator's Guide* for information on managing initialization parameters using a pfile or an spfile.

In any case, if a parameter is reset dynamically, the new value should also be placed in the initialization parameter file, so that the new value is retained if the database is restarted.

Adjusting Initialization Parameter Values When Oracle Streams Values Change

Asynchronous Change Data Capture uses an Oracle Streams configuration for each HotLog and AutoLog change set. This Streams configuration consists of a Streams

capture process and a Streams apply process, with an accompanying queue and queue table. Each Streams configuration uses additional processes, parallel execution servers, and memory.

Oracle Streams capture and apply processes each have a parallelism parameter that is used to improve performance. When a publisher first creates a change set, its capture parallelism value and apply parallelism value are each 1. If desired, a publisher can increase one or both of these values using Streams interfaces.

If Oracle Streams capture parallelism and apply parallelism values are increased after the change sets are created, the staging database DBA must adjust initialization parameter values accordingly. Example 16–1 and Example 16–2 demonstrate how to obtain the capture parallelism and apply parallelism values for change set CHICAGO_DAILY. By default, each parallelism value is 1, so the amount by which a given parallelism value has been increased is the returned value minus 1.

Example 16–1 Obtaining the Oracle Streams Capture Parallelism Value

```
SELECT cp.value FROM DBA_CAPTURE_PARAMETERS cp, CHANGE_SETS cset
   WHERE cset.SET_NAME = 'CHICAGO_DAILY' and
   cset.CAPTURE_NAME = cp.CAPTURE_NAME and
   cp.PARAMETER = 'PARALLELISM';
```

Example 16–2 Obtaining the Oracle Streams Apply Parallelism Value

```
SELECT ap.value FROM DBA_APPLY_PARAMETERS ap, CHANGE_SETS cset
   WHERE cset.SET_NAME = 'CHICAGO_DAILY' and
   cset.APPLY_NAME = ap.APPLY_NAME and
   ap.parameter = 'PARALLELISM';
```

The staging database DBA must adjust the staging database initialization parameters as described in the following list to accommodate the parallel execution servers and other processes and memory required for asynchronous Change Data Capture:

- PARALLEL_MAX_SERVERS

  For each change set for which Oracle Streams capture or apply parallelism values were increased, increase the value of this parameter by the sum of increased Streams parallelism values.

  For example, if the statement in Example 16–1 returns a value of 2, and the statement in Example 16–2 returns a value of 3, then the staging database DBA should increase the value of the PARALLEL_MAX_SERVERS parameter by (2-1)

+ (3-1), or 3 for the CHICAGO_DAILY change set. If the Streams capture or apply parallelism values have increased for other change sets, increases for those change sets must also be made.

- PROCESSES

  For each change set for which Oracle Streams capture or apply parallelism values were changed, increase the value of this parameter by the sum of increased Streams parallelism values. See the previous list item, PARALLEL_MAX_SERVERS, for an example.

- STREAMS_POOL_SIZE

  For each change set for which Oracle Streams capture or apply parallelism values were changed, increase the value of this parameter by (10MB * (the increased capture parallelism value)) + (1MB * increased apply parallelism value).

  For example, if the statement in Example 16–1 returns a value of 2, and the statement in Example 16–2 returns a value of 3, then the staging database DBA should increase the value of the STREAMS_POOL_SIZE parameter by (10 MB * (2-1) + 1MB * (3-1)), or 12MB for the CHICAGO_DAILY change set. If the Oracle Streams capture or apply parallelism values have increased for other change sets, increases for those change sets must also be made.

  See *Oracle Streams Concepts and Administration* for more information on Streams capture parallelism and apply parallelism values. See *Oracle Database Reference* for more information about database initialization parameters.

Publishing Change Data

The following sections provide step-by-step instructions on performing the various types of publishing:

- Performing Synchronous Publishing
- Performing Asynchronous HotLog Publishing
- Performing Asynchronous AutoLog Publishing

Performing Synchronous Publishing

For synchronous Change Data Capture, the publisher must use the predefined change source, SYNC_SOURCE. The publisher can define new change sets or can use the predefined change set, SYNC_SET. The publisher should not create change

tables on source tables owned by SYS or SYSTEM because triggers will not fire and therefore changes will not be captured.

This example shows how to create a change set. If the publisher wants to use the predefined SYNC_SET, he or she should skip Step 3 and specify SYNC_SET as the change set name in the remaining steps.

This example assumes that the publisher and the source database DBA are two different people.

Step 1 Source Database DBA: Set the JAVA_POOL_SIZE parameter.

The source database DBA sets the database initialization parameters, as described in "Setting Initialization Parameters for Change Data Capture Publishing" on page 16-21.

```
java_pool_size = 50000000
```

Step 2 Source Database DBA: Create and grant privileges to the publisher.

The source database DBA creates a user (for example, cdcpub), to serve as the Change Data Capture publisher and grants the necessary privileges to the publisher so that he or she can perform the operations needed to create Change Data Capture change sets and change tables on the source database, as described in"Creating a User to Serve As a Publisher" on page 16-18. This example assumes that the tablespace ts_cdcpub has already been created.

```
CREATE USER cdcpub IDENTIFIED BY cdcpub DEFAULT TABLESPACE ts_cdcpub
QUOTA UNLIMITED ON SYSTEM
QUOTA UNLIMITED ON SYSAUX;
GRANT CREATE SESSION TO cdcpub;
GRANT CREATE TABLE TO cdcpub;
GRANT CREATE TABLESPACE TO cdcpub;
GRANT UNLIMITED TABLESPACE TO cdcpub;
GRANT SELECT_CATALOG_ROLE TO cdcpub;
GRANT EXECUTE_CATALOG_ROLE TO cdcpub;
GRANT CONNECT, RESOURCE TO cdcpub;
```

Step 3 Staging Database Publisher: Create a change set.

The publisher uses the DBMS_CDC_PUBLISH.CREATE_CHANGE_SET procedure on the staging database to create change sets.

The following example shows how to create a change set called CHICAGO_DAILY:

```
BEGIN
```

```
    DBMS_CDC_PUBLISH.CREATE_CHANGE_SET(
    change_set_name    => 'CHICAGO_DAILY',
    description        => 'Change set for job history info',
    change_source_name => 'SYNC_SOURCE');
END;
/
```

The change set captures changes from the predefined change source SYNC_SOURCE. Because begin_date and end_date parameters cannot be specified for synchronous change sets, capture begins at the earliest available change data and continues capturing change data indefinitely.

Step 4 Staging Database Publisher: Create a change table.

The publisher uses the DBMS_CDC_PUBLISH.CREATE_CHANGE_TABLE procedure to create change tables.

The publisher can set the options_string field of the DBMS_CDC_PUBLISH.CREATE_CHANGE_TABLE procedure to have more control over the physical properties and tablespace properties of the change table. The options_string field can contain any option, except partitioning, that is available in the CREATE TABLE statement.

The following example creates a change table that captures changes that occur on a source table. The example uses the sample table HR.JOB_HISTORY as the source table. It assumes that the publisher has already created the TS_CHICAGO_DAILY tablespace.

```
BEGIN
   DBMS_CDC_PUBLISH.CREATE_CHANGE_TABLE(
   owner => 'cdcpub',
   change_table_name => 'jobhist_ct',
   change_set_name => 'CHICAGO_DAILY',
   source_schema => 'HR',
   source_table => 'JOB_HISTORY',
   column_type_list => 'EMPLOYEE_ID NUMBER(6),START_DATE DATE,
                        END_DATE DATE,JOB_ID VARCHAR2(10),
                        DEPARTMENT_ID NUMBER(4)',
   capture_values => 'both',
   rs_id => 'y',
   row_id => 'n',
   user_id => 'n',
   timestamp => 'n',
   object_id => 'n',
   source_colmap => 'y',
```

```
   target_colmap => 'y',
   options_string => 'TABLESPACE TS_CHICAGO_DAILY');
END;
 /
```

This statement creates a change table named `jobhist_ct` within the change set `CHICAGO_DAILY`. The `column_type_list` parameter identifies the columns captured by the change table. The `source_schema` and `source_table` parameters identify the schema and source table that reside in the source database.

The `capture_values` setting in the example indicates that for update operations, the change data will contain two separate rows for each row that changed: one row will contain the row values before the update occurred, and the other row will contain the row values after the update occurred.

Step 5 Staging Database Publisher: Grant access to subscribers.

The publisher controls subscriber access to change data by granting and revoking the `SELECT` privilege on change tables for users and roles. The publisher grants access to specific change tables. Without this step, a subscriber cannot access any change data. This example assumes that user `subscriber1` already exists.

```
GRANT SELECT ON cdcpub.jobhist_ct TO subscriber1;
```

The Change Data Capture synchronous system is now ready for subscriber1 to create subscriptions.

Performing Asynchronous HotLog Publishing

Change Data Capture uses Oracle Streams local capture to perform asynchronous HotLog publishing. (See *Oracle Streams Concepts and Administration* for information on Streams local capture.)

For asynchronous HotLog Change Data Capture, the publisher must use the predefined change source, `HOTLOG_SOURCE`, and must create the change sets and the change tables that will contain the changes. The staging database is always the source database. This example assumes that the publisher and the source database DBA are two different people.

The following steps set up redo logging, Oracle Streams, and Change Data Capture for asynchronous HotLog publishing:

Step 1 Source Database DBA: Set the database initialization parameters.

The source database DBA sets the database initialization parameters, as described in "Setting Initialization Parameters for Change Data Capture Publishing" on page 16-21. In this example, one change set will be defined and the current value of the STREAMS_POOL_SIZE parameter is 50 MB or greater.

```
compatible = 10.1.0
java_pool_size = 50000000;
job_queue_processes = 2
parallel_max_servers = <current value> + 5
processes = <current value> + 7
sessions = <current value> + 2
streams_pool_size = <current value> + 21 MB
undo_retention = 3600
```

Step 2 Source Database DBA: Alter the source database.

The source database DBA performs the following three tasks. The second is required. The first and third are optional, but recommended. It is assumed that the database is currently running in ARCHIVELOG mode.

1. Place the database into FORCE LOGGING logging mode to protect against unlogged direct write operations in the source database that cannot be captured by asynchronous Change Data Capture:

   ```
   ALTER DATABASE FORCE LOGGING;
   ```

2. Enable supplemental logging. Supplemental logging places additional column data into a redo log file whenever an UPDATE operation is performed. Minimally, database-level minimal supplemental logging must be enabled for any Change Data Capture source database:

   ```
   ALTER DATABASE ADD SUPPLEMENTAL LOG DATA;
   ```

3. Create an unconditional log group on all columns to be captured in the source table. Source table columns that are unchanged and are not in an unconditional log group, will be null in the change table, instead of reflecting their actual source table values. (This example captures rows in the HR.JOB_HISTORY table only. The source database DBA would repeat this step for each source table for which change tables will be created.)

   ```
   ALTER TABLE HR.JOB_HISTORY
   ADD SUPPLEMENTAL LOG GROUP log_group_jobhist
     (EMPLOYEE_ID, START_DATE, END_DATE, JOB_ID, DEPARTMENT_ID) ALWAYS;
   ```

If you intend to capture all the column values in a row whenever a column in that row is updated, you can use the following statement instead of listing each column one-by-one in the ALTER TABLE statement. However, do not use this form of the ALTER TABLE statement if all columns are not needed. Logging all columns incurs more overhead than logging selected columns.

```
ALTER TABLE HR.JOB_HISTORY ADD SUPPLEMENTAL LOG DATA (ALL) COLUMNS;
```

See *Oracle Database Administrator's Guide* for information about running a database in ARCHIVELOG mode. See "Asynchronous Change Data Capture and Supplemental Logging" on page 16-50 for more information on enabling supplemental logging.

Step 3 Source Database DBA: Create and grant privileges to the publisher.

The source database DBA creates a user, (for example, cdcpub), to serve as the Change Data Capture publisher and grants the necessary privileges to the publisher so that he or she can perform the underlying Oracle Streams operations needed to create Change Data Capture change sets, and change tables on the source database, as described in "Creating a User to Serve As a Publisher" on page 16-18. This example assumes that the ts_cdcpub tablespace has already been created. For example:

```
CREATE USER cdcpub IDENTIFIED BY cdcpub DEFAULT TABLESPACE ts_cdcpub
QUOTA UNLIMITED ON SYSTEM
QUOTA UNLIMITED ON SYSAUX;
GRANT CREATE SESSION TO cdcpub;
GRANT CREATE TABLE TO cdcpub;
GRANT CREATE TABLESPACE TO cdcpub;
GRANT UNLIMITED TABLESPACE TO cdcpub;
GRANT SELECT_CATALOG_ROLE TO cdcpub;
GRANT EXECUTE_CATALOG_ROLE TO cdcpub;
GRANT CREATE SEQUENCE TO cdcpub;
GRANT CONNECT, RESOURCE, DBA TO cdcpub;

EXECUTE DBMS_STREAMS_AUTH.GRANT_ADMIN_PRIVILEGE(GRANTEE => 'cdcpub');
```

Note that for HotLog Change Data Capture, the source database and the staging database are the same database.

Step 4 Source Database DBA: Prepare the source tables.

The source database DBA must prepare the source tables on the source database for asynchronous Change Data Capture by instantiating each source table so that the underlying Oracle Streams environment records the information it needs to capture

each source table's changes. The source table structure and the column datatypes must be supported by Change Data Capture. See "Datatypes and Table Structures Supported for Asynchronous Change Data Capture" on page 16-51 for more information.

```
BEGIN
DBMS_CAPTURE_ADM.PREPARE_TABLE_INSTANTIATION(TABLE_NAME => 'hr.job_history');
END;
/
```

Step 5 Staging Database Publisher: Create change sets.

The publisher uses the DBMS_CDC_PUBLISH.CREATE_CHANGE_SET procedure on the staging database to create change sets. Note that when Change Data Capture creates a change set, its associated Oracle Streams capture and apply processes are also created (but not started).

The following example creates a change set called CHICAGO_DAILY that captures changes starting today, and stops capturing change data 5 days from now.

```
BEGIN
       DBMS_CDC_PUBLISH.CREATE_CHANGE_SET(
       change_set_name => 'CHICAGO_DAILY',
       description => 'Change set for job history info',
       change_source_name => 'HOTLOG_SOURCE',
       stop_on_ddl => 'y',
       begin_date => sysdate,
       end_date => sysdate+5);
END;
/
```

The change set captures changes from the predefined HOTLOG_SOURCE change source.

Step 6 Staging Database Publisher: Create the change tables that will contain the changes to the source tables.

The publisher uses the DBMS_CDC_PUBLISH.CREATE_CHANGE_TABLE procedure on the staging database to create change tables.

The publisher creates one or more change tables for each source table to be published, specifies which columns should be included, and specifies the combination of before and after images of the change data to capture.

The following example creates a change table on the staging database that captures changes made to a source table on the source database. The example uses the sample table HR.JOB_HISTORY as the source table.

```
BEGIN
       DBMS_CDC_PUBLISH.CREATE_CHANGE_TABLE(
       owner => 'cdcpub',
       change_table_name => 'job_hist_ct',
       change_set_name => 'CHICAGO_DAILY',
       source_schema => 'HR',
       source_table => 'JOB_HISTORY',
       column_type_list => 'EMPLOYEE_ID NUMBER(6),START_DATE DATE,END_DATE DATE,
       JOB_ID VARCHAR(10), DEPARTMENT_ID NUMBER(4)',
       capture_values => 'both',
       rs_id => 'y',
       row_id => 'n',
       user_id => 'n',
       timestamp => 'n',
       object_id => 'n',
       source_colmap => 'n',
       target_colmap => 'y',
       options_string => 'TABLESPACE TS_CHICAGO_DAILY');
END;
/
```

This statement creates a change table named job_history_ct within change set CHICAGO_DAILY. The column_type_list parameter identifies the columns to be captured by the change table. The source_schema and source_table parameters identify the schema and source table that reside on the source database.

The capture_values setting in this statement indicates that for update operations, the change data will contain two separate rows for each row that changed: one row will contain the row values before the update occurred and the other row will contain the row values after the update occurred.

The options_string parameter in this statement specifies a tablespace for the change table. (This example assumes that the publisher previously created the TS_CHICAGO_DAILY tablespace.)

Step 7 Staging Database Publisher: Enable the change set.

Because asynchronous change sets are always disabled when they are created, the publisher must alter the change set to enable it. The Oracle Streams capture and apply processes are started when the change set is enabled.

```
BEGIN
```

```
   DBMS_CDC_PUBLISH.ALTER_CHANGE_SET(
      change_set_name => 'CHICAGO_DAILY',
      enable_capture => 'y');
END;
/
```

Step 8 Staging Database Publisher: Grant access to subscribers.

The publisher controls subscriber access to change data by granting and revoking the SELECT privilege on change tables for users and roles. The publisher grants access to specific change tables. Without this step, a subscriber cannot access change data. This example assumes that user `subscriber1` already exists.

```
GRANT SELECT ON cdcpub.job_hist_ct TO subscriber1;
```

The Change Data Capture Asynchronous HotLog system is now ready for subscriber1 to create subscriptions.

Performing Asynchronous AutoLog Publishing

Change Data Capture uses Oracle Streams downstream capture to perform asynchronous AutoLog publishing. The Change Data Capture staging database is considered a downstream database in the Streams environment. See *Oracle Streams Concepts and Administration* for information on Streams downstream capture.

For asynchronous AutoLog Change Data Capture, the publisher creates new change sources, as well as the change sets and the change tables that will contain the changes that are made to individual source tables. For AutoLog Change Data Capture, the staging database is usually remote from the source database.

Steps must be performed on both the source database and the staging database to set up redo logging, Streams, and Change Data Capture for asynchronous AutoLog publishing. Because the source database and staging database are usually on separate systems, this example assumes that the source database DBA, the staging database DBA, and the publisher are different people.

Step 1 Source Database DBA: Prepare to copy redo log files from the source database.

The source database DBA and the staging database DBA must set up log transport services to copy redo log files from the source database to the staging database and to prepare the staging database to receive these redo log files, as follows:

1. The source database DBA configures Oracle Net so that the source database can communicate with the staging database. (See *Oracle Net Services Administrator's Guide* for information about Oracle Net).
2. The source database DBA sets the database initialization parameters on the source database as described in "Setting Initialization Parameters for Change Data Capture Publishing" on page 16-21. In the following code example `stagingdb` is the network name of the staging database:

```
compatible = 10.1.0
java_pool_size = 50000000
log_archive_dest_1="location=/oracle/dbs mandatory reopen=5"
log_archive_dest_2 = "service=stagingdb arch optional noregister reopen=5
                      template = /usr/oracle/dbs/arch1_%s_%t_%r.dbf"
log_archive_dest_state_1 = enable
log_archive_dest_state_2 = enable
log_archive_format="arch1_%s_%t_%r.dbf"
remote_login_passwordfile=shared
```

   See *Oracle Data Guard Concepts and Administration* for information on log transport services.

Step 2 Staging Database DBA: Set the database initialization parameters.

The staging database DBA sets the database initialization parameters on the staging database, as described in "Setting Initialization Parameters for Change Data Capture Publishing" on page 16-21. In this example, one change set will be defined and the current value for the `STREAMS_POOL_SIZE` parameter is 50 MB or greater:

```
compatible = 10.1.0
global_names = true
java_pool_size = 50000000
job_queue_processes = 2
parallel_max_servers = <current_value> + 5
processes = <current_value> + 7
remote_login_passwordfile = shared
sessions = <current value> + 2
streams_pool_size = <current_value> + 21 MB
undo_retention = 3600
```

Step 3 Source Database DBA: Alter the source database.

The source database DBA performs the following three tasks. The second is required. The first and third are optional, but recommended. It is assumed that the database is currently running in `ARCHIVELOG` mode.

1. Place the database into FORCE LOGGING logging mode to protect against unlogged direct writes in the source database that cannot be captured by asynchronous Change Data Capture:

   ```
   ALTER DATABASE FORCE LOGGING;
   ```

2. Enable supplemental logging. Supplemental logging places additional column data into a redo log file whenever an update operation is performed.

   ```
   ALTER DATABASE ADD SUPPLEMENTAL LOG DATA;
   ```

3. Create an unconditional log group on all columns to be captured in the source table. Source table columns that are unchanged and are not in an unconditional log group, will be null in the change table, instead of reflecting their actual source table values. (This example captures rows in the HR.JOB_HISTORY table only. The source database DBA would repeat this step for each source table for which change tables will be created).

   ```
   ALTER TABLE HR.JOB_HISTORY
   ADD SUPPLEMENTAL LOG GROUP log_group_job_hist
      (EMPLOYEE_ID, START_DATE, END_DATE, JOB_ID, DEPARTMENT_ID) ALWAYS;
   ```

   If you intend to capture all the column values in a row whenever a column in that row is updated, you can use the following statement instead of listing each column one-by-one in the ALTER TABLE statement. However, do not use this form of the ALTER TABLE statement if all columns are not needed. Logging all columns incurs more overhead than logging selected columns.

   ```
   ALTER TABLE HR.JOB_HISTORY ADD SUPPLEMENTAL LOG DATA (ALL) COLUMNS;
   ```

   See *Oracle Database Administrator's Guide* for information about running a database in ARCHIVELOG mode. See "Asynchronous Change Data Capture and Supplemental Logging" on page 16-50 for more information on enabling supplemental logging.

Step 4 Staging Database DBA: Create and grant privileges to the publisher.

The staging database DBA creates a user, (for example, cdcpub), to serve as the Change Data Capture publisher and grants the necessary privileges to the publisher so that he or she can perform the underlying Oracle Streams operations needed to create Change Data Capture change sources, change sets, and change tables on the staging database, as described in "Creating a User to Serve As a Publisher" on page 16-18. For example:.

```
CREATE USER cdcpub IDENTIFIED BY cdcpub DEFAULT TABLESPACE ts_cdcpub
```

```
   QUOTA UNLIMITED ON SYSTEM
   QUOTA UNLIMITED ON SYSAUX;
GRANT CREATE SESSION TO cdcpub;
GRANT CREATE TABLE TO cdcpub;
GRANT CREATE TABLESPACE TO cdcpub;
GRANT UNLIMITED TABLESPACE TO cdcpub;
GRANT SELECT_CATALOG_ROLE TO cdcpub;
GRANT EXECUTE_CATALOG_ROLE TO cdcpub;
GRANT CONNECT, RESOURCE, DBA TO cdcpub;
GRANT CREATE SEQUENCE TO cdcpub;
EXECUTE DBMS_STREAMS_AUTH.GRANT_ADMIN_PRIVILEGE(grantee => 'cdcpub');
```

Step 5 Source Database DBA: Build the LogMiner data dictionary.

The source database DBA builds a LogMiner data dictionary at the source database so that log transport services can transport this data dictionary to the staging database. This LogMiner data dictionary build provides the table definitions as they were just prior to beginning to capture change data. Change Data Capture automatically updates the data dictionary with any source table data definition language (DDL) operations that are made during the course of change data capture to ensure that the dictionary is always synchronized with the source database tables.

When building the LogMiner data dictionary, the source database DBA should get the SCN value of the data dictionary build. In Step 8, when the publisher creates a change source, he or she will need to provide this value as the `first_scn` parameter.

```
SET SERVEROUTPUT ON
VARIABLE f_scn NUMBER;
BEGIN
    :f_scn := 0;
    DBMS_CAPTURE_ADM.BUILD(:f_scn);
    DBMS_OUTPUT.PUT_LINE('The first_scn value is ' || :f_scn);
END;
/
The first_scn value is 207722
```

For asynchronous AutoLog publishing to work, it is critical that the source database DBA build the data dictionary before the source tables are prepared. The source database DBA must be careful to follow Step 5 and Step 6 in the order they are presented here.

See *Oracle Streams Concepts and Administration* for more information on the LogMiner data dictionary.

Step 6 Source Database DBA: Prepare the source tables.

The source database DBA must prepare the source tables on the source database for asynchronous Change Data Capture by instantiating each source table so that the underlying Oracle Streams environment records the information it needs to capture each source table's changes. The source table structure and the column datatypes must be supported by Change Data Capture. See "Datatypes and Table Structures Supported for Asynchronous Change Data Capture" on page 16-51 for more information.

```
BEGIN
   DBMS_CAPTURE_ADM.PREPARE_TABLE_INSTANTIATION(
       TABLE_NAME => 'hr.job_history');
END;
/
```

Step 7 Source Database DBA: Get the global name of the source database.

In Step 8, the publisher will need to reference the global name of the source database. The source database DBA can query the `GLOBAL_NAME` column in the `GLOBAL_NAME` view on the source database to retrieve this information for the publisher:

```
SELECT GLOBAL_NAME FROM GLOBAL_NAME;
GLOBAL_NAME
--------------------------------------------------------------------------------
HQDB
```

Step 8 Staging Database Publisher: Identify each change source database and create the change sources.

The publisher uses the `DBMS_CDC_PUBLISH.CREATE_AUTOLOG_CHANGE_SOURCE` procedure on the staging database to create change sources.

The process of managing the capture system begins with the creation of a change source. A change source describes the source database from which the data will be captured, and manages the relationship between the source database and the staging database. A change source always specifies the SCN of a data dictionary build from the source database as its `first_scn` parameter.

The publisher gets the SCN of the data dictionary build and the global database name from the source database DBA (as shown in Step 5 and Step 7, respectively). If the publisher cannot get the value to use for the `first_scn` parameter value from the source database DBA, then, with the appropriate privileges, he or she can query the `V$ARCHIVED_LOG` view on the source database to determine the value. This is

described in the DBMS_CDC_PUBLISH chapter of the *PL/SQL Packages and Types Reference*.

On the staging database, the publisher creates the AutoLog change source and specifies the global name as the source_database parameter value and the SCN of the data dictionary build as the first_scn parameter value:

```
BEGIN
   DBMS_CDC_PUBLISH.CREATE_AUTOLOG_CHANGE_SOURCE(
       change_source_name => 'CHICAGO',
       description => 'test source',
       source_database => 'HQDB',
       first_scn => 207722);
END;
/
```

Step 9 Staging Database Publisher: Create change sets.

The publisher uses the DBMS_CDC_PUBLISH.CREATE_CHANGE_SET procedure on the staging database to create change sets. The publisher can optionally provide beginning and ending dates to indicate where to begin and end the data capture.

Note that when Change Data Capture creates a change set, its associated Oracle Streams capture and apply processes are also created (but not started).

The following example shows how to create a change set called CHICAGO_DAILY that captures changes starting today, and continues capturing change data indefinitely. (If, at some time in the future, the publisher decides that he or she wants to stop capturing change data for this change set, he or she should disable the change set and then drop it.)

```
BEGIN
   DBMS_CDC_PUBLISH.CREATE_CHANGE_SET(
       change_set_name => 'CHICAGO_DAILY',
       description => 'change set for job history info',
       change_source_name => 'CHICAGO',
       stop_on_ddl => 'y');
END;
/
```

Step 10 Staging Database Publisher: Create the change tables.

The publisher uses the DBMS_CDC_PUBLISH.CREATE_CHANGE_TABLE procedure on the staging database to create change tables.

The publisher creates one or more change tables for each source table to be published, specifies which columns should be included, and specifies the combination of before and after images of the change data to capture.

The publisher can set the options_string field of the DBMS_CDC_PUBLISH.CREATE_CHANGE_TABLE procedure to have more control over the physical properties and tablespace properties of the change tables. The options_string field can contain any option available (except partitioning) on the CREATE TABLE statement. In this example, it specifies a tablespace for the change set. (This example assumes that the publisher previously created the TS_CHICAGO_DAILY tablespace.)

The following example creates a change table on the staging database that captures changes made to a source table in the source database. The example uses the sample table HR.JOB_HISTORY.

```
BEGIN
   DBMS_CDC_PUBLISH.CREATE_CHANGE_TABLE(
       owner => 'cdcpub',
       change_table_name => 'JOB_HIST_CT',
       change_set_name => 'CHICAGO_DAILY',
       source_schema => 'HR',
       source_table => 'JOB_HISTORY',
       column_type_list => 'EMPLOYEE_ID NUMBER(6),START_DATE DATE,END_DATE DATE,
       JOB_ID VARCHAR2(10), DEPARTMENT_ID NUMBER(4)',
       capture_values => 'both',
       rs_id => 'y',
       row_id => 'n',
       user_id => 'n',
       timestamp => 'n',
       object_id => 'n',
       source_colmap => 'n',
       target_colmap => 'y',
       options_string => 'TABLESPACE TS_CHICAGO_DAILY');
END;
/
```

This example creates a change table named job_hist_ct within change set CHICAGO_DAILY. The column_type_list parameter identifies the columns captured by the change table. The source_schema and source_table parameters identify the schema and source table that reside in the source database, not the staging database.

The capture_values setting in the example indicates that for update operations, the change data will contain two separate rows for each row that changed: one row

will contain the row values before the update occurred and the other row will contain the row values after the update occurred.

Step 11 Staging Database Publisher: Enable the change set.

Because asynchronous change sets are always disabled when they are created, the publisher must alter the change set to enable it. The Oracle Streams capture and apply processes are started when the change set is enabled.

```
BEGIN
   DBMS_CDC_PUBLISH.ALTER_CHANGE_SET(
       change_set_name => 'CHICAGO_DAILY',
       enable_capture => 'y');
END;
/
```

Step 12 Source Database DBA: Switch the redo log files at the source database.

To begin capturing data, a log file must be archived. The source database DBA can initiate the process by switching the current redo log file:

```
ALTER SYSTEM ARCHIVE LOGFILE;
```

Step 13 Staging Database Publisher: Grant access to subscribers.

The publisher controls subscriber access to change data by granting and revoking the SQL `SELECT` privilege on change tables for users and roles on the staging database. The publisher grants access to specific change tables. Without this step, a subscriber cannot access any change data. This example assumes that user `subscriber1` already exists.

```
GRANT SELECT ON cdcpub.job_hist_ct TO subscriber1;
```

The Change Data Capture asynchronous AutoLog system is now ready for subscriber1 to create subscriptions.

Subscribing to Change Data

When a publisher creates a change table, Change Data Capture assigns it a publication ID and maintains a list of all the publication IDs in the `ALL_PUBLISHED_COLUMNS` view. A **publication ID** is a numeric value that Change Data Capture assigns to each change table defined by the publisher.

The subscribers register their interest in one or more source tables, and obtain subscriptions to these tables. Assuming sufficient access privileges, the subscribers

may subscribe to any source tables for which the publisher has created one or more change tables by doing one of the following:

- Specifying the source tables and columns of interest.

  When there are multiple publications that contain the columns of interest, then Change Data Capture selects one on behalf of the user.

- Specifying the publication IDs and columns of interest.

  When there are multiple publications on a single source table and these publications share some columns, the subscriber should specify publication IDs (rather than source tables) if any of the shared columns will be used in a single subscription.

The following steps provide an example to demonstrate the second scenario:

Step 1 Find the source tables for which the subscriber has access privileges.

The subscriber queries the ALL_SOURCE_TABLES view to see all the published source tables for which the subscriber has access privileges:

```
SELECT * FROM ALL_SOURCE_TABLES;

SOURCE_SCHEMA_NAME             SOURCE_TABLE_NAME
------------------------------ ------------------------------
HR                             JOB_HISTORY
```

Step 2 Find the change set names and columns for which the subscriber has access privileges.

The subscriber queries the ALL_PUBLISHED_COLUMNS view to see all the change sets, columns, and publication IDs for the HR.JOB_HISTORY table for which the subscriber has access privileges:

```
SELECT UNIQUE CHANGE_SET_NAME, COLUMN_NAME, PUB_ID
FROM ALL_PUBLISHED_COLUMNS
WHERE SOURCE_SCHEMA_NAME ='HR' AND SOURCE_TABLE_NAME = 'JOB_HISTORY';

CHANGE_SET_NAME     COLUMN_NAME        PUB_ID
---------------     ------------------ ------------
CHICAGO_DAILY       DEPARTMENT_ID      34883
CHICAGO_DAILY       EMPLOYEE_ID        34883
CHICAGO_DAILY       END_DATE           34883
CHICAGO_DAILY       JOB_ID             34883
CHICAGO_DAILY       START_DATE         34883
```

Step 3 Create a subscription.

The subscriber calls the DBMS_CDC_SUBSCRIBE.CREATE_SUBSCRIPTION procedure to create a subscription.

The following example shows how the subscriber identifies the change set of interest (CHICAGO_DAILY), and then specifies a unique subscription name that will be used throughout the life of the subscription:

```
BEGIN
       DBMS_CDC_SUBSCRIBE.CREATE_SUBSCRIPTION(
       change_set_name => 'CHICAGO_DAILY',
       description => 'Change data for JOB_HISTORY',
       subscription_name => 'JOBHIST_SUB');
END;
/
```

Step 4 Subscribe to a source table and the columns in the source table.

The subscriber calls the DBMS_CDC_SUBSCRIBE.SUBSCRIBE procedure to specify which columns of the source tables are of interest to the subscriber.

A subscription can contain one or more source tables referenced by the same change set.

In the following example, the subscriber wants to see the EMPLOYEE_ID, START_DATE, and END_DATE columns from the JOB_HISTORY table. Because all these columns are contained in the same publication (and the subscriber has privileges to access that publication) as shown in the query in Step 2, the following call can be used:

```
BEGIN
       DBMS_CDC_SUBSCRIBE.SUBSCRIBE(
       subscription_name => 'JOBHIST_SUB',
       source_schema => 'HR',
       source_table => 'JOB_HISTORY',
       column_list => 'EMPLOYEE_ID, START_DATE, END_DATE, JOB_ID',
       subscriber_view => 'JOBHIST_VIEW');
END;
/
```

However, assume that for security reasons the publisher has not created a single change table that includes all these columns. Suppose that instead of the results shown in Step 2, the query of the ALL_PUBLISHED_COLUMNS view shows that the columns of interest are included in multiple publications as shown in the following example:

```
SELECT UNIQUE CHANGE_SET_NAME, SOURCE_TABLE_NAME, COLUMN_NAME, PUB_ID
FROM ALL_PUBLISHED_COLUMNS
WHERE SOURCE_SCHEMA_NAME ='HR' AND SOURCE_TABLE_NAME = 'JOB_HISTORY';

CHANGE_SET_NAME     COLUMN_NAME        PUB_ID
---------------     ------------------ ------------
CHICAGO_DAILY       DEPARTMENT_ID      34883
CHICAGO_DAILY       EMPLOYEE_ID        34883
CHICAGO_DAILY       END_DATE           34885
CHICAGO_DAILY       JOB_ID             34883
CHICAGO_DAILY       START_DATE         34885
CHICAGO_DAILY       EMPLOYEE_ID        34885
```

This returned data shows that the EMPLOYEE_ID column is included in both publication 34883 and publication 34885. A single subscribe call must specify columns available in a single publication. Therefore, if the subscriber wants to subscribe to columns in both publications, using EMPLOYEE_ID to join across the subscriber views, then the subscriber must use two calls, each specifying a different publication ID:

```
BEGIN
       DBMS_CDC_SUBSCRIBE.SUBSCRIBE(
       subscription_name => 'MULTI_PUB',
       publication_id => 34885,
       column_list => 'EMPLOYEE_ID, START_DATE, END_DATE',
       subscriber_view => 'job_dates');

       DBMS_CDC_SUBSCRIBE.SUBSCRIBE(
       subscription_name => 'MULTI_PUB',
       publication_id => 34883,
       column_list => 'EMPLOYEE_ID, JOB_ID',
       subscriber_view => 'job_type');
END;
/
```

Note that each DBMS_CDC_SUBSCRIBE.SUBSCRIBE call specifies a unique subscriber view.

Step 5 Activate the subscription.

The subscriber calls the DBMS_CDC_SUBSCRIBE.ACTIVATE_SUBSCRIPTION procedure to activate the subscription.

A subscriber calls this procedure when finished subscribing to source tables (or publications), and ready to receive change data. Whether subscribing to one or

multiple source tables, the subscriber needs to call the ACTIVATE_SUBSCRIPTION procedure only once.

The ACTIVATE_SUBSCRIPTION procedure creates empty subscriber views. At this point, no additional source tables can be added to the subscription.

```
BEGIN
   DBMS_CDC_SUBSCRIBE.ACTIVATE_SUBSCRIPTION(
       subscription_name => 'JOBHIST_SUB');
END;
/
```

Step 6 Get the next set of change data.

The subscriber calls the DBMS_CDC_SUBSCRIBE.EXTEND_WINDOW procedure to get the next available set of change data. This sets the high boundary of the subscription window.

For example:

```
BEGIN
   DBMS_CDC_SUBSCRIBE.EXTEND_WINDOW(
       subscription_name => 'JOBHIST_SUB');
END;
/
```

If this is the subscriber's first call to the EXTEND_WINDOW procedure, then the subscription window contains all the change data in the publication. Otherwise, the subscription window contains all the new change data that was created since the last call to the EXTEND_WINDOW procedure.

If no new change data has been added, then the subscription window remains unchanged.

Step 7 Read and query the contents of the subscriber views.

The subscriber uses the SQL SELECT statement on the subscriber view to query the change data (within the current boundaries of the subscription window). The subscriber can do this for each subscriber view in the subscription. For example:

```
SELECT EMPLOYEE_ID, START_DATE, END_DATE FROM JOBHIST_VIEW;

EMPLOYEE_ID START_DAT  END_DATE
----------- ---------  ---------
176         24-MAR-98  31-DEC-98
180         24-MAR-98  31-DEC-98
190         01-JAN-99  31-DEC-99
```

```
200         01-JAN-99  31-DEC-99
```

The subscriber view name, JOBHIST_VIEW, was specified when the subscriber called the DBMS_CDC_SUBSCRIBE.SUBSCRIBE procedure in Step 4.

Step 8 Indicate that the current set of change data is no longer needed.

The subscriber uses the DBMS_CDC_SUBSCRIBE.PURGE_WINDOW procedure to let Change Data Capture know that the subscriber no longer needs the current set of change data. This helps Change Data Capture to manage the amount of data in the change table and sets the low boundary of the subscription window. Calling the DBMS_CDC_SUBSCRIBE.PURGE_WINDOW procedure causes the subscription window to be empty.

For example:

```
BEGIN
   DBMS_CDC_SUBSCRIBE.PURGE_WINDOW(
       subscription_name => 'JOBHIST_SUB');
END;
/
```

Step 9 Repeat Steps 6 through 8.

The subscriber repeats Steps 6 through 8 as long as the subscriber is interested in additional change data.

Step 10 End the subscription.

The subscriber uses the DBMS_CDC_SUBSCRIBE.DROP_SUBSCRIPTION procedure to end the subscription. This is necessary to prevent the publications that underlie the subscription from holding change data indefinitely.

```
BEGIN
   DBMS_CDC_SUBSCRIBE.DROP_SUBSCRIPTION(
       subscription_name => 'JOBHIST_SUB');
END;
/
```

Considerations for Asynchronous Change Data Capture

The following sections provide information that the publisher and the source and staging database DBAs should be aware of when using the asynchronous mode of Change Data Capture.

Asynchronous Change Data Capture and Redo Log Files

The asynchronous mode of Change Data Capture uses redo log files, as follows:

- HotLog

  Asynchronous HotLog Change Data Capture reads online redo log files whenever possible and archived redo log files otherwise.

- AutoLog

  Asynchronous AutoLog Change Data Capture reads redo log files that have been copied from the source database to the staging database by log transport services.

  In ARCH mode, log transport services copies archived redo log files to the staging database after a log switch occurs on the source database. In LGWR mode, log transport services copies redo data to the staging database while it is being written to the online redo log file on the source database, and then makes it available to Change Data Capture when a log switch occurs on the source database.

For log files to be archived, the source databases for asynchronous Change Data Capture must run in ARCHIVELOG mode, as specified with the following SQL statement:

```
ALTER DATABASE ARCHIVELOG;
```

See *Oracle Database Administrator's Guide* for information about running a database in ARCHIVELOG mode.

A redo log file used by Change Data Capture must remain available on the staging database until Change Data Capture has captured it. However, it is not necessary that the redo log file remain available until the Change Data Capture subscriber is done with the change data.

To determine which redo log files are no longer needed by Change Data Capture for a given change set, the publisher alters the change set's Streams capture process, which causes Streams to perform some internal cleanup and populates the DBA_LOGMNR_PURGED_LOG view. The publisher follows these steps:

1. Uses the following query on the staging database to get the three SCN values needed to determine an appropriate new first_scn value for the change set, CHICAGO_DAILY:

   ```
   SELECT cap.CAPTURE_NAME, cap.FIRST_SCN, cap.APPLIED_SCN,
          cap.SAFE_PURGE_SCN
   FROM DBA_CAPTURE cap, CHANGE_SETS cset
   ```

```
WHERE cset.SET_NAME = 'CHICAGO_DAILY' AND
      cap.CAPTURE_NAME = cset.CAPTURE_NAME;

CAPTURE_NAME                   FIRST_SCN APPLIED_SCN SAFE_PURGE_SCN
------------------------------ ---------- ----------- --------------
CDC$C_CHICAGO_DAILY                778059      778293         778293
```

2. Determines a new `first_scn` value that is greater than the original `first_scn` value and less than or equal to the `applied_scn` and `safe_purge_scn` values returned by the query in step 1. In this example, this value is 778293, and the capture process name is `CDC$C_CHICAGO_DAILY`, therefore the publisher can alter the `first_scn` value for the capture process as follows:

   ```
   BEGIN
   DBMS_CAPTURE_ADM.ALTER_CAPTURE(
     capture_name => 'CDC$C_CHICAGO_DAILY',
     first_scn    => 778293);
   END;
   /
   ```

   If there if not an SCN value that meets these criteria, then the change set needs all of its redo log files.

3. Queries the `DBA_LOGMNR_PURGED_LOG` view to see any log files that are no longer needed by Change Data Capture:

   ```
   SELECT FILE_NAME
   FROM DBA_LOGMNR_PURGED_LOG;
   ```

> **Note:** Redo log files may be required on the staging database for purposes other than Change Data Capture. Before deleting a redo log file, the publisher should be sure that no other users need it.

See the information on setting the first SCN for an existing capture process and on capture process checkpoints in *Oracle Streams Concepts and Administration* for more information.

The `first_scn` value can be updated for all change sets in an AutoLog change source by using the `DBMS_CDC_PUBLISH.ALTER_AUTOLOG_CHANGE_SOURCE` `first_scn` parameter. Note that the new `first_scn` value must meet the criteria stated in step 2 of the preceding list for all change sets in the AutoLog change source.

Both the size of the redo log files and the frequency with which a log switch occurs can affect the generation of the archived log files at the source database. For Change Data Capture, the most important factor in deciding what size to make a redo log file is the tolerance for latency between when a change is made and when that change data is available to subscribers. However, because the Oracle Database software attempts a check point at each log switch, if the redo log file is too small, frequent log switches will lead to frequent checkpointing and negatively impact the performance of the source database.

See *Oracle Data Guard Concepts and Administration* for step-by-step instructions on monitoring log file archival information. Substitute the terms source and staging database for the Oracle Data Guard terms primary database and archiving destinations, respectively.

When using log transport services to supply redo log files to an AutoLog change source, gaps in the sequence of redo log files are automatically detected and resolved. If a situation arises where it is necessary to manually add a log file to an AutoLog change set, the publisher can use instructions on explicitly assigning log files to a downstream capture process described in the *Oracle Streams Concepts and Administration*. These instructions require the name of the capture process for the AutoLog change set. The publisher can obtain the name of the capture process for an AutoLog change set from the `CHANGE_SETS` data dictionary view.

Asynchronous Change Data Capture and Supplemental Logging

The asynchronous mode of Change Data Capture works best when supplemental logging is enabled on the source database. (Supplemental logging is not used by synchronous Change Data Capture.)

Oracle recommends that the source database DBA enable minimal supplemental logging at the database level:

```
ALTER DATABASE ADD SUPPLEMENTAL LOG DATA;
```

In addition, Oracle recommends that the source database DBA:

- Supplementally log all source table columns that are part of a primary key or function to uniquely identify a row. This can be done using database-level or table-level identification key logging, or through a table-level unconditional log group.
- Create an unconditional log group for all source table columns that are captured by any asynchronous change table. This should be done before any change tables are created on a source table.

```
ALTER TABLE SH.PROMOTIONS
ADD SUPPLEMENTAL LOG GROUP log_group_cust
(PROMO_NAME, PROMO_SUBCATEGORY, PROMO_CATEGORY) ALWAYS;
```

If an unconditional log group is not created for all source table columns to be captured, then when an update DML operation occurs, some unchanged user column values in change tables will be null instead of reflecting the actual source table value.

For example, suppose a source table contains two columns, X and Y, and that the source database DBA has defined an unconditional log group for that table that includes only column Y. Furthermore, assume that a user updates only column Y in that table row. When the subscriber views the change data for that row, the value of the unchanged column X will be null. However, because the actual column value for X is excluded from the redo log file and therefore cannot be included in the change table, the subscriber cannot assume that the actual source table value for column X is null. The subscriber must rely on the contents of the TARGET_COLMAP$ control column to determine whether the actual source table value for column X is null or it is unchanged.

See *Oracle Database Utilities* for more information on the various types of supplemental logging.

Datatypes and Table Structures Supported for Asynchronous Change Data Capture

Asynchronous Change Data Capture supports columns of all built-in Oracle datatypes except the following:

- BFILE
- BLOB
- CLOB
- LONG
- NCLOB
- ROWID
- UROWID
- object types (for example, XMLType)

Asynchronous Change Data Capture does not support the following table structures:

- Source tables that are temporary tables
- Source tables that are object tables
- Index-organized tables with columns of unsupported datatypes (including LOB columns) or with overflow segments

Managing Published Data

This section provides information about the management tasks involved in managing change sets and change tables. For the most part, these tasks are the responsibility of the publisher. However, to purge unneeded data from the change tables, both the publisher and the subscribers have responsibilities as described in "Purging Change Tables of Unneeded Data" on page 16-65.

The following topics are covered in this section:

- Managing Asynchronous Change Sets
- Managing Change Tables
- Considerations for Exporting and Importing Change Data Capture Objects
- Impact on Subscriptions When the Publisher Makes Changes

Managing Asynchronous Change Sets

This section provides information about tasks that the publisher can perform to manage asynchronous change sets. The following topics are covered:

- Creating Asynchronous Change Sets with Starting and Ending Dates
- Enabling and Disabling Asynchronous Change Sets
- Stopping Capture on DDL for Asynchronous Change Sets
- Recovering from Errors Returned on Asynchronous Change Sets

Creating Asynchronous Change Sets with Starting and Ending Dates

Change sets associated with asynchronous change sources (AutoLog and HotLog) can optionally specify starting and ending dates to limit the change data that they capture. A change set with no starting date begins capture with the earliest available change data. A change set with no ending date continues capturing change data indefinitely.

The following example creates a change set, JOBHIST_SET, in the AutoLog change source, HQ_SOURCE, that starts capture two days from now and continues indefinitely:

```
BEGIN
DBMS_CDC_PUBLISH.CREATE_CHANGE_SET(
change_set_name => 'JOBHIST_SET',
description => 'Job History Application Change Set',
change_source_name => 'HQ_SOURCE',
stop_on_ddl => 'Y',
begin_date => sysdate+2);
END;
/
```

Enabling and Disabling Asynchronous Change Sets

The publisher can enable and disable asynchronous change sets. When a change set is disabled, it cannot process new change data until the change set is enabled.

Synchronous change sets are always created enabled and cannot be disabled. Asynchronous change sets are always created disabled.

The publisher can enable the JOBHIST_SET asynchronous change set with the following call:

```
BEGIN
DBMS_CDC_PUBLISH.ALTER_CHANGE_SET(
change_set_name => 'JOBHIST_SET',
enable_capture  => 'y');
END;
/
```

The Oracle Streams capture and apply processes for the change set are started when the change set is enabled.

The publisher can disable the JOBHIST_SET asynchronous change set with the following call:

```
BEGIN
DBMS_CDC_PUBLISH.ALTER_CHANGE_SET(
change_set_name => 'JOBHIST_SET',
enable_capture  => 'n');
END;
/
```

The Oracle Streams capture and apply processes for the change set are stopped when the change set is disabled.

Although a disabled change set cannot process new change data, it does not lose any change data provided that the necessary archived redo log files remain available until the change set is enabled and processes them. Oracle recommends that change sets be enabled as much as possible to avoid accumulating archived redo log files. See "Asynchronous Change Data Capture and Redo Log Files" on page 16-48 for more information.

Change Data Capture can automatically disable an asynchronous change set if DDL is encountered during capture and the `stop_on_ddl parameter` is set to 'Y', or if there is an internal capture error. The publisher must check the alert log for more information, take any necessary actions to adjust to the DDL or recover from the internal error, and explicitly enable the change set. See "Recovering from Errors Returned on Asynchronous Change Sets" on page 16-55 for more information.

Stopping Capture on DDL for Asynchronous Change Sets

The publisher can specify that a change set be automatically disabled by Change Data Capture if DDL is encountered. Some DDL commands can adversely affect capture, such as dropping a source table column that is being captured. If the change set stops on DDL, the publisher has a chance to analyze and fix the problem before capture proceeds. If the change set does not stop on DDL, internal capture errors are possible after DDL occurs.

The publisher can specify whether a change set stops on DDL when creating or altering the change set. The publisher can alter the `JOBHIST_SET` change set to stop on DDL with the following call:

```
BEGIN
DBMS_CDC_PUBLISH.ALTER_CHANGE_SET(
change_set_name  => 'JOBHIST_SET',
stop_on_ddl      => 'y');
END;
/
```

The publisher can alter the `JOBHIST_SET` change set so that it does not stop on DDL by using the following call:

```
BEGIN
DBMS_CDC_PUBLISH.ALTER_CHANGE_SET(
 change_set_name => 'JOBHIST_SET',
 stop_on_ddl     => 'n');
END;
```

```
/
```

If a DDL statement causes processing to stop, a message is written to the alert log indicating the DDL statement and change set involved. For example, if a TRUNCATE TABLE DDL statement causes the JOB_HIST change set to stop processing, the alert log contains lines such as the following:

```
Change Data Capture received DDL for change set JOB_HIST
Change Data Capture received DDL and stopping: truncate table job_history
```

Because they do not affect the column data itself, the following DDL statements do not cause Change Data Capture to stop capturing change data when the stop_on_ddl parameter is set to 'Y':

- ANALYZE TABLE
- LOCK TABLE
- GRANT privileges to access a table
- REVOKE privileges to access a table
- COMMENT on a table
- COMMENT on a column

These statements can be issued on the source database without concern for their impact on Change Data Capture processing. For example, when an ANALYZE TABLE command is issued on the JOB_HISTORY source table, the alert log on the staging database will contain a line similar to the following when the stop_on_ddl parameter is set to 'Y':

```
Change Data Capture received DDL and ignoring: analyze table job_history compute
statistics
```

Recovering from Errors Returned on Asynchronous Change Sets

Errors during asynchronous Change Data Capture are possible due to a variety of circumstances. If a change set stops on DDL, that DDL must be removed before capture can continue. If a change set does not stop on DDL, but a DDL change occurs that affects capture, it can result in an internal error. There are also system conditions that can cause internal errors, such as being out of disk space.

In all these cases, the change set is disabled and marked as having an error. Subscriber procedures detect when a change set has an error and return the following message:

```
ORA-31514: change set disabled due to capture error
```

The publisher must check the alert log for more information and attempt to fix the underlying problem. The publisher can then attempt to recover from the error by calling ALTER_CHANGE_SET with the recover_after_error and remove_ddl parameters set appropriately. The publisher can retry this procedure as many times as necessary to resolve the problem. When recovery succeeds, the error is removed from the change set and the publisher can enable the asynchronous change set (as described in "Enabling and Disabling Asynchronous Change Sets" on page 16-53).

If more information is needed to resolve capture errors, the publisher can query the DBA_APPLY_ERROR view to see information about Streams apply errors; capture errors correspond to Streams apply errors. The publisher must always use the DBMS_CDC_PUBLISH.ALTER_CHANGE_SET procedure to recover from capture errors because both Streams and Change Data Capture actions are needed for recovery and only the DBMS_CDC_PUBLISH.ALTER_CHANGE_SET procedure performs both sets of actions. See *Oracle Streams Concepts and Administration* for information about the error queue and apply errors.

The following two scenarios demonstrate how a publisher might investigate and then recover from two different types of errors returned to Change Data Capture:

An Error Due to Running Out of Disk Space The publisher can view the contents of the alert log to determine which error is being returned for a given change set and which SCN is not being processed. For example, the alert log may contain lines such as the following (where LCR refers to a logical change record):

```
Change Data Capture has encountered error number: 1688 for change set: CHICAGO_
DAILY
Change Data Capture did not process LCR with scn 219337
```

The publisher can determine the message associated with the error number specified in the alert log by querying the DBA_APPLY_ERROR view for the error message text, where the APPLY_NAME in the DBA_APPLY_ERROR view equals the APPLY_NAME of the change set specified in the alert log. For example:

```
SQL> SELECT ERROR_MESSAGE FROM DBA_APPLY_ERROR
       WHERE APPLY_NAME =
      (SELECT APPLY_NAME FROM CHANGE_SETS WHERE SET_NAME ='CHICAGO_DAILY');

ERROR_MESSAGE
--------------------------------------------------------------------------------
ORA-01688: unable to extend table LOGADMIN.CT1 partition P1 by 32 in tablespace
```

```
    TS_CHICAGO_DAILY
```

After taking action to fix the problem that is causing the error, the publisher can attempt to recover from the error. For example, the publisher can attempt to recover the CHICAGO_DAILY change set after an error with the following call:

```
BEGIN
DBMS_CDC_PUBLISH.ALTER_CHANGE_SET(
  change_set_name     => 'CHICAGO_DAILY',
  recover_after_error => 'y');
END;
/
```

If the recovery does not succeed, then an error is returned and the publisher can take further action to attempt to resolve the problem. The publisher can retry the recovery procedure as many times as necessary to resolve the problem.

Note: When recovery succeeds, the publisher must remember to enable the change set.

An Error Due to Stopping on DDL Suppose a SQL TRUNCATE TABLE statement is issued against the JOB_HISTORY source table and the stop_on_ddl parameter is set to 'Y', then an error such as the following is returned from an attempt to enable the change set:

```
ERROR at line 1:
ORA-31468: cannot process DDL change record
ORA-06512: at "SYS.DBMS_CDC_PUBLISH", line 79
ORA-06512: at line 2
```

The alert log will contain lines similar to the following:

```
Mon Jun  9 16:13:44 2003
Change Data Capture received DDL for change set JOB_HIST
Change Data Capture received DDL and stopping: truncate table job_history
Mon Jun  9 16:13:50 2003
Change Data Capture did not process LCR with scn 219777
Streams Apply Server P001 pid=19 OS id=11730 stopped
Streams Apply Reader P000 pid=17 OS id=11726 stopped
Streams Apply Server P000 pid=17 OS id=11726 stopped
Streams Apply Server P001 pid=19 OS id=11730 stopped
Streams AP01 with pid=15, OS id=11722 stopped
```

Because the TRUNCATE TABLE statement removes all rows from a table, the publisher will want to notify subscribers before taking action to reenable Change Data Capture processing. He or she might suggest to subscribers that they purge and extend their subscription windows. The publisher can then attempt to restore Change Data Capture processing by altering the change set and specifying the `remove_ddl => 'Y'` parameter along with the `recover_after_error => 'Y'` parameter, as follows:

```
BEGIN
DBMS_CDC_PUBLISH.ALTER_CHANGE_SET(
  change_set_name     => 'JOB_HIST',
  recover_after_error => 'y',
  remove_ddl          => 'y');
END;
/
```

After this procedure completes, the alert log will contain lines similar to the following:

```
Mon Jun  9 16:20:17 2003
Change Data Capture received DDL and ignoring: truncate table JOB_HISTORY
The scn for the truncate statement is 202998
```

Now, the publisher must enable the change set.

Managing Change Tables

All change table management tasks are the responsibility of the publisher with one exception: purging change tables of unneeded data. This task requires action from both the publisher and the subscriber to work most effectively.

The following topics are discussed in this section:

- Creating Change Tables
- Understanding Change Table Control Columns
- Understanding TARGET_COLMAP$ and SOURCE_COLMAP$ Values
- Controlling Subscriber Access to Change Tables
- Purging Change Tables of Unneeded Data
- Dropping Change Tables

Creating Change Tables

When creating change tables, the publisher should be aware that Oracle recommends the following:

- For all modes of Change Data Capture, publishers should not create change tables in system tablespaces.

  Either of the following methods can be used to ensure that change tables are created in tablespaces managed by the publisher. The first method creates all the change tables created by the publisher in a single tablespace, while the second method allows the publisher to specify a different tablespace for each change table.

  – When the database administrator creates the account for the publisher, he or she can specify a default tablespace. For example:

    ```
    CREATE USER cdcpub DEFAULT TABLESPACE ts_cdcpub;
    ```

  – When the publisher creates a change table, he or she can use the `options_string` parameter to specify a tablespace for the change table being created. See Step 4 in "Performing Synchronous Publishing" on page 16-27 for an example.

  If both methods are used, the tablespace specified by the publisher in the `options_string` parameter takes precedence over the default tablespace specified in the SQL `CREATE USER` statement.

- For asynchronous Change Data Capture, the publisher should be certain that the source table that will be referenced in a `DBMS_CDC_PUBLISH.CREATE_CHANGE_TABLE` procedure has been created prior to calling this procedure, particularly if the change set that will be specified in the procedure has the `stop_on_ddl` parameter set to 'Y'.

  Suppose the publisher created a change set with the `stop_on_ddl` parameter set to 'Y', then created the change table, and then the source table was created. In this scenario, the DDL that creates the source table would trigger the `stop_on_ddl` condition and cause Change Data Capture processing to stop.

  Note: The publisher must not attempt to control a change table's partitioning properties. Change Data Capture automatically manages the change table partitioning as part of its change table management.

- For asynchronous Change Data Capture, the source database DBA should create an unconditional log group for all source table columns that will be captured in a change table. This should be done before any change tables are created on a source table. If an unconditional log group is not created for source table columns to be captured, then when an update DML operation occurs, some unchanged user column values in change tables will be null instead of reflecting the actual source table value. This will require the publisher to evaluate the TARGET_COLMAP$ control column to distinguish unchanged column values from column values that are actually null. See "Asynchronous Change Data Capture and Supplemental Logging" on page 16-50 for information on creating unconditional log groups and see "Understanding Change Table Control Columns" on page 16-60 for information on control columns.

Understanding Change Table Control Columns

A change table consists of two things: the change data itself, which is stored in a database table, and the system metadata necessary to maintain the change table, which includes control columns.

Table 16–8 describes the control columns for a change table, including the column name, datatype, mode, whether the column is optional or not, and a description.

The mode indicates the type of Change Data Capture associated with the column. A value of *All* indicates that the column is associated with the synchronous mode and both modes of asynchronous Change Data Capture. Note that for both synchronous and asynchronous Change Data Capture, if the subscriber wants a query of a subscriber view to return DML changes in the order in which they occurred, the query should order data by CSCN$ and then RSID$.

A column is considered optional if the publisher can choose to exclude it from a change table using the operation parameter and the parameters that indicate specific control columns in the DBMS_CDC_PUBLISH.CREATE_CHANGE_TABLE procedure. The syntax for the DBMS_CDC_PUBLISH.CREATE_CHANGE_TABLE procedure is documented in *PL/SQL Packages and Types Reference*.

Table 16–8 Control Columns for a Change Table

Column	Datatype	Mode	Optional Column	Description
OPERATION$	CHAR(2)	All	No	The value in this column can be any one of the following[1]: I: Indicates this row represents an insert operation UO: Indicates this row represents the before-image of an updated source table row for the following cases: ■ Asynchronous Change Data Capture ■ Synchronous Change Data Capture when the change table includes a primary key-based object ID and a captured column that is not a primary key has changed. UU: Indicates this row represents the before-image of an updated source table row for synchronous Change Data Capture, in cases other than those represented by UO. UN: Indicates this row represents the after-image of an updated source table row. D: Indicates this row represents a delete operation.
CSCN$	NUMBER	All	No	Commit SCN of this transaction.
RSID$	NUMBER	All	Yes	Unique row sequence ID within this transaction.[2] The RSID$ column reflects an operation's capture order within a transaction, but not across transactions. The publisher cannot use the RSID$ column value by itself to order committed operations across transactions; it must be used in conjunction with the CSCN$ column value.
SOURCE_ COLMAP$	RAW(128)	Synchronous	Yes	Bit mask[3] of updated columns in the source table.
TARGET_ COLMAP$	RAW(128)	All	Yes	Bit mask[3] of updated columns in the change table.
COMMIT_ TIMESTAMP$	DATE	All	No	Commit time of this transaction.
TIMESTAMP$	DATE	All	Yes	Time when the operation occurred in the source table.

Table 16–8 (Cont.) Control Columns for a Change Table

Column	Datatype	Mode	Optional Column	Description
USERNAME$	VARCHAR2(30)	All	Yes	Name of the user who caused the operation.
ROW_ID$	ROW_ID	All	Yes	Row ID of affected row in source table.
XIDUSN$	NUMBER	Asynchronous	No	Transaction ID undo segment number.
XIDSLT$	NUMBER	Asynchronous	No	Transaction ID slot number.
XIDSEQ$	NUMBER	Asynchronous	No	Transaction ID sequence number.
SYS_NC_OID$	RAW(16)	Synchronous	Yes	Object ID.

[1] If you specify a query based on this column, specify the I or D column values as "I " or "D ", respectively. The OPERATION$ column is a 2-character column; values are left-justified and space-filled. A query that specifies a value of "I" or "D" will return no values.

[2] You can use the RSID$ column to associate the after-image with the before-image of a given operation. The value of the after-image RSID$ column always matches the value of the before-image RSID$ column value for a given update operation.

[3] A bit mask is an array of binary values that indicate which columns in a row have been updated.

Understanding TARGET_COLMAP$ and SOURCE_COLMAP$ Values

The TARGET_COLMAP$ and SOURCE_COLMAP$ columns are used to indicate which columns in a row have changed. The TARGET_COLMAP$ column indicates which columns in the change table row have changed. The SOURCE_COLMAP$ column (which is included for synchronous change tables only) indicates which columns in a source table row have changed.

Because the datatype of the TARGET_COLMAP$ and the SOURCE_COLMAP$ columns is RAW(128), each column can hold 128 bytes of binary information. This binary information consists of words strung together with the low order byte on the left and the high order byte on the right. By default, when the field is displayed, the system interprets each byte and displays a value for each. Example 16–3 provides a sample TARGET_COLMAP$ value. The text following the example explains how to interpret this data.

Example 16–3 Sample TARGET_COLMAP$ VALUE

```
FE1100000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000
```

In Example 16–3, the first 'FE' is the low order byte and the last '00' is the high order byte. To correctly interpret the meaning of the values, you must consider which bits

are set in each byte. The bits in the bitmap are counted starting at zero. The first bit is bit 0, the second bit is bit 1, and so on. Bit 0 is always ignored. For the other bits, if a particular bit is set to 1, it means that the value for that column has been changed.

To interpret the string of bytes as presented in the Example 16–3, you read from left to right. The first byte is the string 'FE'. Broken down into bits (again from left to right) this string is "1111 1110", which maps to columns " 7,6,5,4 3,2,1,-" in the change table (where the hyphen represents the ignored bit). The first bit tells you if column 7 in the change table has changed. The right-most bit is ignored. The values in Example 16–3 indicate that the first 7 columns have a value present. This is typical - the first several columns in a change table are control columns.

The next byte in Example 16–3 is the string '11'. Broken down into bits, this string is "0001 0001", which maps to columns "15,14,13,12 11,10,9,8" in the change table. These bits indicate that columns 8 and 12 are changed. Columns 9, 10, 11, 13, 14, 15, are not changed. The rest of the string is all '00', indicating that none of the other columns has been changed.

A publisher can issue the following query to determine the mapping of column numbers to column names:

```
SELECT COLUMN_NAME, COLUMN_ID FROM ALL_TAB_COLUMNS WHERE OWNER='PUBLISHER_
STEWART' AND TABLE_NAME='MY_CT';

COLUMN_NAME                    COLUMN_ID
------------------------------ ----------
OPERATION$                              1
CSCN$                                   2
COMMIT_TIMESTAMP$                       3
XIDUSN$                                 4
XIDSLT$                                 5
XIDSEQ$                                 6
RSID$                                   7
TARGET_COLMAP$                          8
C_ID                                    9
C_KEY                                  10
C_ZIP                                  11

COLUMN_NAME                     COLUMN_ID
------------------------------ ----------
C_DATE                                 12
C_1                                    13
C_3                                    14
C_5                                    15
C_7                                    16
```

```
C_9                                17
```

Using Example 16–3, the publisher can conclude that following columns were changed in the particular change row in the change table represented by this TARGET_COLMAP$ value: OPERATION$, CSCN$, COMMIT_TIMESTAMP$, XIDUSN$, XIDSLT$, XIDSEQ$, RSID$, TARGET_COLMAP$, and C_DATE.

Note that Change Data Capture generates values for all control columns in all change rows, so the bits corresponding to control columns are always set to 1 in every TARGET_COLMAP$ column. Bits that correspond to user columns that have changed are set to 1 for the OPERATION$ column values UN and I, as appropriate. (See Table 16–8 for information about the OPERATION$ column values.)

A common use for the values in the TARGET_COLMAP$ column is for determining the meaning of a null value in a change table. A column value in a change table can be null for two reasons: the value was changed to null by a user or application, or Change Data Capture inserted a null value into the column because a value was not present in the redo data from the source table. If a user changed the value to null, the bit for that column will be set to 1; if Change Data Capture set the value to null, then the column will be set to 0.

Values in the SOURCE_COLMAP$ column are interpreted in a similar manner, with the following exceptions:

- The SOURCE_COLMAP$ column refers to columns of source tables, not columns of change tables.
- The SOURCE_COLMAP$ column does not reference control columns because these columns are not present in the source table.
- Changed source columns are set to 1 in the SOURCE_COLMAP$ column for OPERATION$ column values UO, UU, UN, and I, as appropriate. (See Table 16–8 for information about the OPERATION$ column values.)
- The SOURCE_COLMAP$ column is valid only for synchronous change tables.

Controlling Subscriber Access to Change Tables

The publisher grants privileges to subscribers to allow them access to change tables. Because privileges on source tables are not propagated to change tables, a subscriber might have privileges to perform a SELECT operation on a source table, but might not have privileges to perform a SELECT operation on a change table.

The publisher controls subscriber access to change data by using the SQL GRANT and REVOKE statements to grant and revoke the SELECT privilege on change tables

for users and roles. The publisher must grant the SELECT privilege before a subscriber can subscribe to the change table.

The publisher must not grant any DML access (use of INSERT, UPDATE, or DELETE statements) to the subscribers on the change tables because of the risk that a subscriber might inadvertently change the data in the change table, making it inconsistent with its source. Furthermore, the publisher should avoid creating change tables in schemas to which subscribers have DML access.

Purging Change Tables of Unneeded Data

This section describes purge operations. For optimum results, purge operations require action from the subscribers. Each subscriber indicates when he or she is done using change data, and then Change Data Capture or the publisher actually removes (purges) data that is no longer being used by any subscriber from the change table, as follows:

- Subscriber

  When finished using change data, a subscriber must call the DBMS_CDC_SUBSCRIBE.PURGE_WINDOW procedure. This indicates to Change Data Capture and the publisher that the change data is no longer needed by this subscriber. The DBMS_CDC_SUBSCRIBE.PURGE_WINDOW procedure does not physically remove rows from the change tables.

  In addition, as shown in "Subscribing to Change Data" beginning on page 16-42, the subscriber should call the DBMS_CDC_SUBSCRIBE.DROP_SUBSCRIPTION procedure to drop unneeded subscriptions.

  See *PL/SQL Packages and Types Reference* for information about the DBMS_CDC_SUBSCRIBE.DROP_SUBSCRIPTION and the DBMS_CDC_SUBSCRIBE.PURGE_WINDOW procedures.

- Change Data Capture

  Change Data Capture creates a purge job using the DBMS_JOB PL/SQL package (which runs under the account of the publisher who created the first change table). This purge job automatically calls the DBMS_CDC_PUBLISH.PURGE procedure to remove data that subscribers are no longer using from the change tables. This ensures that the size of the change tables does not grow without limit. The automatic call to the DBMS_CDC_PUBLISH.PURGE procedure evaluates all active subscription windows to determine which change data is still needed. It will not purge any data that could be referenced by one or more subscribers with active subscription windows.

By default, this purge job runs every 24 hours. The publisher who created the first change table can adjust this interval using the PL/SQL DBMS_JOB.CHANGE procedure. The values for the JOB parameter for this procedure can be found by querying the USER_JOBS view for the job number that corresponds to the WHAT column containing the string 'SYS.DBMS_CDC_PUBLISH.PURGE();'.

See *PL/SQL Packages and Types Reference* for information about the DBMS_JOB package and the *Oracle Database Reference* for information about the USER_JOBS view.

- Publisher

  The publisher can manually execute a purge operation at any time. The publisher has the ability to perform purge operations at a finer granularity than the automatic purge operation performed by Change Data Capture. There are three purge operations available to the publisher:

  – DBMS_CDC_PUBLISH.PURGE

    Purges all change tables on the staging database. This is the same PURGE operation as is performed automatically by Change Data Capture.

  – DBMS_CDC_PUBLISH.PURGE_CHANGE_SET

    Purges all the change tables in a named change set.

  – DBMS_CDC_PUBLISH.PURGE_CHANGE_TABLE

    Purges a named changed table.

Thus, calls to the DBMS_CDC_SUBSCRIBE.PURGE_WINDOW procedure by subscribers and calls to the PURGE procedure by Change Data Capture (or one of the PURGE procedures by the publisher) work together: when each subscriber purges a subscription window, it indicates change data that is no longer needed; the PURGE procedure evaluates the sum of the input from all the subscribers before actually purging data.

Note that it is possible that a subscriber could fail to call PURGE_WINDOW, with the result being that unneeded rows would not be deleted by the purge job. The publisher can query the DBA_SUBSCRIPTIONS view to determine if this is happening. In extreme circumstances, a publisher may decide to manually drop an active subscription so that space can be reclaimed. One such circumstance is a subscriber that is an applications program that fails to call the PURGE_WINDOW procedure when appropriate. The DBMS_CDC_PUBLISH.DROP_SUBSCRIPTION procedure lets the publisher drop active subscriptions if circumstances require it;

however, the publisher should first consider that subscribers may still be using the change data.

Dropping Change Tables

To drop a change table, the publisher must call the DBMS_CDC_PUBLISH.DROP_CHANGE_TABLE procedure. This procedure ensures that both the change table itself and the Change Data Capture metadata for the table are dropped. If the publisher tries to use a SQL DROP TABLE statement on a change table, it will fail with the following error:

```
ORA-31496 must use DBMS_CDC_PUBLISH.DROP_CHANGE_TABLE to drop change tables
```

The DBMS_CDC_PUBLISH.DROP_CHANGE_TABLE procedure also safeguards the publisher from inadvertently dropping a change table while there are active subscribers using the change table. If DBMS_CDC_PUBLISH.DROP_CHANGE_TABLE is called while subscriptions are active, the procedure will fail with the following error:

```
ORA-31424 change table has active subscriptions
```

If the publisher still wants to drop the change table, in spite of active subscriptions, he or she must call the DROP_CHANGE_TABLE procedure using the `force_flag => 'Y'` parameter. This tells Change Data Capture to override its normal safeguards and allow the change table to be dropped despite active subscriptions. The subscriptions will no longer be valid, and subscribers will lose access to the change data.

Note: The SQL DROP USER CASCADE statement will drop all the publisher's change tables, and if any other users have active subscriptions to the (dropped) change table, these will no longer be valid. In addition to dropping the change tables, the DROP USER CASCADE statement drops any change sources, change sets, and subscriptions that are owned by the user specified in the DROP USER CASCADE statement.

Considerations for Exporting and Importing Change Data Capture Objects

Starting in Oracle Databse 10*g*, Oracle Data Pump is the supported export and import utility for Change Data Capture. Change Data Capture change sources, change sets, change tables, and subscriptions are exported and imported by the Oracle Data Pump `expdp` and `impdp` commands with the following restrictions:

- Change Data Capture objects are exported and imported only as part of full database export and import operations (those in which the `expdp` and `impdb` commands specify the `FULL=y` parameter). Schema-level import and export operations include some underlying objects (for example, the table underlying a change table), but not the Change Data Capture metadata needed for change data capture to occur.
- AutoLog change sources, change sets, and change tables are not supported.
- You should export asynchronous change sets and change tables at a time when users are not making DDL and DML changes to the database being exported.
- When importing asynchronous change sets and change tables, you must also import the underlying Oracle Streams configuration; set the Oracle Data Pump import parameter `STREAMS_CONFIGURATION` to `y` explicitly (or implicitly by accepting the default), so that the necessary Streams objects are imported. If you perform an import operation and specify `STREAMS_CONFIGURATION=n`, then imported asynchronous change sets and change tables will not be able to continue capturing change data.
- Change Data Capture objects never overwrite existing objects when they are imported (similar to the effect of the import command `TABLE_EXISTS_ACTION=skip` parameter for tables). Change Data Capture generates warnings in the import log for these cases.
- Change Data Capture objects are validated at the end of an import operation to determine if all expected underlying objects are present in the correct form. Change Data Capture generates validation warnings in the import log if it detects validation problems. Imported Change Data Capture objects with validation warnings usually cannot continue capturing change data.

The following are examples of Data Pump export and import commands that support Change Data Capture objects:

```
> expdp system/manager DIRECTORY=dpump_dir FULL=y
> impdp system/manager DIRECTORY=dpump_dir FULL=y STREAMS_CONFIGURATION=y
```

After a Data Pump full database import operation completes for a database containing AutoLog Change Data Capture objects, the following steps must be performed to restore these objects:

1. The publisher must manually drop the change tables with the SQL `DROP TABLE` command. This is needed because the tables are imported without the accompanying Change Data Capture metadata.

2. The publisher must re-create the AutoLog change sources, change sets, and change tables using the appropriate DBMS_CDC_PUBLISH procedures.
3. Subscribers must re-create their subscriptions to the AutoLog change sets.

Change data may be lost in the interval between a Data Pump full database export operation involving AutoLog Change Data Capture objects and their re-creation after a Data Pump full database import operation in the preceding step. This can be minimized by preventing changes to the source tables during this interval, if possible.

See *Oracle Database Utilities* for information on Oracle Data Pump.

The following are publisher considerations for exporting and importing change tables:

- When change tables are imported, the job queue is checked for a Change Data Capture purge job. If no purge job is found, then one is submitted automatically (using the DBMS_CDC_PUBLISH.PURGE procedure). If a change table is imported, but no subscriptions are taken out before the purge job runs (24 hours later, by default), then all rows in the table will be purged.

  The publisher can use one of the following methods to prevent the purging of data from a change table:

  – Suspend the purge job using the DBMS_JOB package to either disable the job (using the BROKEN procedure) or execute the job sometime in the future when there are subscriptions (using the NEXT_DATE procedure).

    Note: If you disable the purge job by marking it as broken, you need to remember to reset it once subscriptions have been activated. This prevents the change table from growing indefinitely.

  – Create a temporary subscription to preserve the change table data until real subscriptions appear. Then, drop the temporary subscription.

- When importing data into a source table for which a change table already exists, the imported data is also recorded in any associated change tables.

  Assume that the publisher has a source table SALES that has an associated change table ct_sales. When the publisher imports data into SALES, that data is also recorded in ct_sales.

- When importing a change table having the optional control ROW_ID column, the ROW_ID columns stored in the change table have meaning only if the

associated source table has not been imported. If a source table is re-created or imported, each row will have a new ROW_ID that is unrelated to the ROW_ID that was previously recorded in a change table.

The original level of export and import support available in Oracle9*i* is retained for backward compatibility. Synchronous change tables that reside in the SYNC_SET change set can be exported as part of a full database, schema, or individual table export operation and can be imported as needed. The following Change Data Capture objects are not included in the original export and import support: change sources, change sets, change tables that do not reside in the SYNC_SET change set, and subscriptions.

Impact on Subscriptions When the Publisher Makes Changes

The Change Data Capture environment is dynamic. The publisher can add and drop change tables at any time. The publisher can also add columns to and drop columns from existing change tables at any time. The following list describes how changes to the Change Data Capture environment affect subscriptions:

- Subscribers do not get explicit notification if the publisher adds a new change table or adds columns to an existing change table. A subscriber can check the ALL_PUBLISHED_COLUMNS view to see if new columns have been added, and whether or not the subscriber has access to them.
- Table 16–9 describes what happens when the publisher adds a column to a change table.

Table 16–9 Effects of Publisher Adding a Column to a Change Table

If the publisher adds	And . . .	Then . . .
A user column	A new subscription includes this column	The subscription window for this subscription starts at the point the column was added.
A user column	A new subscription does not include this newly added column	The subscription window for this subscription starts at the earliest available change data. The new column will not be seen.
A user column	A subscription exists	The subscription window for this subscription remains unchanged.

Table 16–9 (Cont.) Effects of Publisher Adding a Column to a Change Table

If the publisher adds	And . . .	Then . . .
A control column	A new subscription is created	The subscription window for this subscription starts at the earliest available change data. The subscription can see the control column immediately. All change table rows that existed prior to adding the control column will have the null value for the newly added control column.
A control column	A subscription exists	This subscription can see the new control column when the subscription window is extended (`DBMS_CDC_PUBLISH.EXTEND_WINDOW` procedure) such that the low boundary for the window crosses over the point when the control column was added.

Implementation and System Configuration

Change Data Capture comes packaged with the appropriate Oracle drivers already installed with which you can implement either asynchronous or synchronous data capture. Starting with Oracle Database 10*g*, the synchronous mode of Change Data Capture is included with the Standard Edition; the synchronous and asynchronous modes of Change Data Capture are included with the Enterprise Edition.

In addition, note that Change Data Capture uses Java. Therefore, when you install Oracle Database, ensure that Java is enabled.

Change Data Capture places systemwide triggers on the SQL `CREATE TABLE`, `ALTER TABLE`, and `DROP TABLE` statements. If system triggers are disabled on the source database, Change Data Capture will not function correctly. Therefore, you should never disable system triggers.

To remove Change Data Capture from the database, the SQL script `rmcdc.sql` is provided in the `admin` directory. This will remove the system triggers that Change Data Capture places on the SQL `CREATE TABLE`, `ALTER TABLE`, and `DROP TABLE` statements. In addition, `rmcdc.sql` removes all Java classes used by Change Data Capture. Note that after `rmcdc.sql` is called, Change Data Capture will no longer operate on the system. If the system administrator decides to remove the Java Virtual Machine from a database, `rmcdc.sql` must be called before `rmjvm` is called.

To reinstall Change Data Capture, the SQL script `initcdc.sql` is provided in the `admin` directory. It creates the Change Data Capture system triggers and Java classes that are required by Change Data Capture.

Synchronous Change Data Capture Restriction on Direct-Path INSERT

Synchronous Change Data Capture does not support the direct-path `INSERT` statement (and, by association, the `MERGE` statement and the `multi_table_insert` clause of the `INSERT` statement) in parallel DML mode.

When the publisher creates a change table in synchronous mode, Change Data Capture creates triggers on the source table. Because a direct-path `INSERT` statement disables all database triggers, any rows inserted into the source table using the SQL statement for direct-path `INSERT` in parallel DML mode will not be captured in the change table.

Similarly, Change Data Capture cannot capture the inserted rows from multitable insert and merge operations because these statements use a direct-path `INSERT` statement. The direct-path `INSERT` statement does not return an error message to indicate that the triggers used by Change Data Capture did not fire.

See *Oracle Database SQL Reference* for more information regarding the direct-path `INSERT` statement and triggers.

17

SQLAccess Advisor

This chapter illustrates how to use the SQLAccess Advisor, which is a tuning tool that provides advice on materialized views, indexes, and materialized view logs. The chapter contains:

- Overview of the SQLAccess Advisor in the DBMS_ADVISOR Package
- Using the SQLAccess Advisor
- Tuning Materialized Views for Fast Refresh and Query Rewrite

Overview of the SQLAccess Advisor in the DBMS_ADVISOR Package

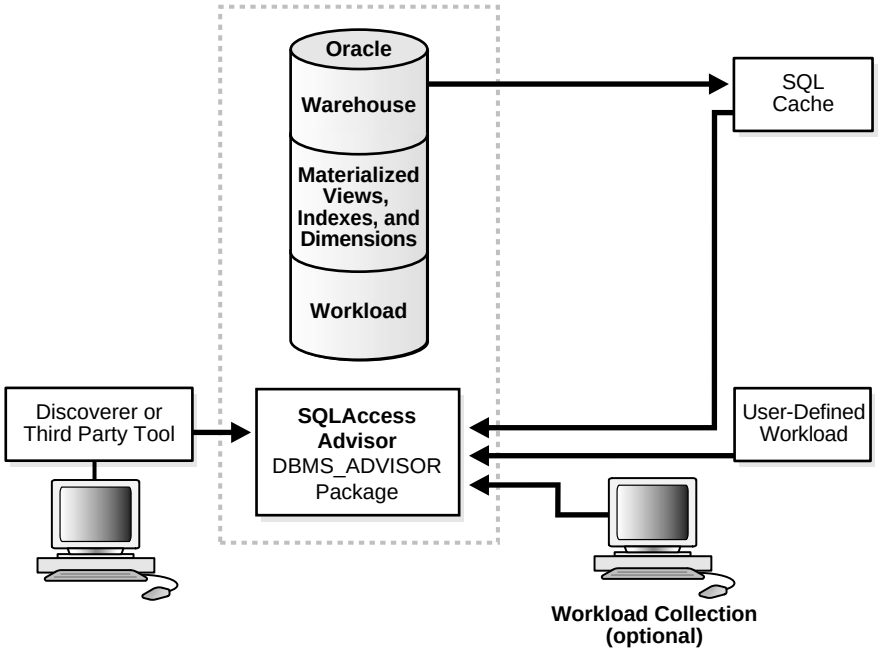
Materialized views and indexes are essential when tuning a database to achieve optimum performance for complex, data-intensive queries. The SQLAccess Advisor helps you achieve your performance goals by recommending the proper set of materialized views, materialized view logs, and indexes for a given workload. Understanding and using these structures is essential when optimizing SQL as they can result in significant performance improvements in data retrieval. The advantages, however, do not come without a cost. Creation and maintenance of these objects can be time consuming, and space requirements can be significant.

The SQLAccess Advisor recommends both bitmap indexes and B-tree indexes. A bitmap index offers a reduced response time for many types of ad hoc queries and reduced storage requirements compared to other indexing techniques. B-tree indexes are most commonly used in a data warehouse to index unique or near-unique keys.

Another component of the SQLAccess Advisor also recommends how to optimize materialized views so that they can be fast refreshable and take advantage of general query rewrite.

The SQLAccess Advisor can be run from Oracle Enterprise Manager (accessible from the Advisor Central page) using the SQLAccess Advisor Wizard or by invoking the `DBMS_ADVISOR` package. The `DBMS_ADVISOR` package consists of a collection of analysis and advisory functions and procedures callable from any PL/SQL program. Figure 17–1 illustrates how the SQLAccess Advisor recommends materialized views for a given workload obtained from a user-defined table or the SQL cache. If a workload is not provided, it can generate and use a hypothetical workload also.

Figure 17–1 Materialized Views and the SQLAccess Advisor



Using the SQLAccess Advisor Wizard or API, you can do the following:

- Recommend materialized views and indexes based on collected or hypothetical workload information.
- Manage workloads.
- Mark, update, and remove recommendations.

In addition, you can use the SQLAccess Advisor API to do the following:

- Perform a quick tune using a single SQL statement.
- Show how to make a materialized view fast refreshable.
- Show how to change a materialized view so that general query rewrite is possible.

The SQLAccess Advisor's recommendations are significantly improved if you gather structural statistics about table and index cardinalities, and the distinct cardinalities of every dimension level column, `JOIN KEY` column, and fact table key

column. You do this by gathering either exact or estimated statistics with the DBMS_STATS package. Because gathering statistics is time-consuming and extreme statistical accuracy is not required, it is generally preferable to estimate statistics. Without these statistics, any queries referencing that table will be marked as invalid in the workload, resulting in no recommendations being made for those queries. It is also recommended that all existing indexes and materialized views have been analyzed. See *PL/SQL Packages and Types Reference* for more information regarding the DBMS_STATS package.

Overview of Using the SQLAccess Advisor

One of the easiest ways to use the SQLAccess Advisor is to invoke its wizard, which is available in Oracle Enterprise Manager from the Advisor Central page. If you prefer to use SQLAccess Advisor through the DBMS_ADVISOR package, this section describes the basic components and the sequence in which the various procedures must be called.

This section describes the four steps in generating a set of recommendations:

- Create a task
- Define the workload
- Generate the recommendations
- View and implement the recommendations

Step 1 Create a task

Before any recommendations can be made, a task must be created. The task is important because it is where all information relating to the recommendation process resides, including the results of the recommendation process. If you use the wizard in Oracle Enterprise Manager or the DBMS_ADVISOR.QUICK_TUNE procedure, the task is created automatically for you. In all other cases, you must create a task using the DBMS_ADVISOR.CREATE_TASK procedure.

You can control what a task does by defining parameters for that task using the DBMS_ADVISOR.SET_TASK_PARAMETER procedure.

See "Creating Tasks" on page 17-10 for more information about creating tasks.

Step 2 Define the workload

The workload is one of the primary inputs for the SQLAccess Advisor, and it consists of one or more SQL statements, plus various statistics and attributes that fully describe each statement. If the workload contains all SQL statements from a

target business application, the workload is considered a full workload; if the workload contains a subset of SQL statements, it is known as a partial workload. The difference between a full and a partial workload is that in the former case, the SQLAccess Advisor may recommend dropping certain existing materialized views and indexes if it finds that they are not being used effectively.

Typically, the SQLAccess Advisor uses the workload as the basis for all analysis activities. Although the workload may contain a wide variety of statements, it carefully ranks the entries according to a specific statistic, business importance, or a combination of statistics and business importance. This ranking is critical in that it enables the SQLAccess Advisor to process the most important SQL statements ahead of those with less business impact.

For a collection of data to be considered a valid workload, the SQLAccess Advisor may require particular attributes to be present. Although analysis can be performed if some of the items are missing, the quality of the recommendations may be greatly diminished. For example, the SQLAccess Advisor requires a workload to contain a SQL query and the user who executed the query. All other attributes are optional; however, if the workload also contained I/O and CPU information, then SQLAccess Advisor may be able to better evaluate the current efficiency of the statement. The workload is stored as a separate object, which is created using the `DBMS_ADVISOR.CREATE_SQLWKLD` procedure, and can easily be shared among many Advisor tasks. Because the workload is independent, it must be linked to a task using the `DBMS_ADVISOR.ADD_SQLWKLD_REF` procedure. Once this link has been established, the workload cannot be deleted or modified until all Advisor tasks have removed their dependency on the workload. A workload reference will be removed when a parent Advisor task is deleted or when the workload reference is manually removed from the Advisor task by the user using the `DBMS_ADVISOR.DELETE_SQLWKLD_REF` procedure.

You can use the SQLAccess Advisor without a workload, however, for best results, a workload must be provided in the form of a user-supplied table, SQL Tuning Set or imported from the SQL Cache. If a workload is not provided, the SQLAccess Advisor can generate and use a hypothetical workload based on the dimensions defined in your schema.

Once the workload is loaded into the repository or at the time the recommendations are generated, a filter can be applied to the workload to restrict what is analyzed. This provides the ability to generate different sets of recommendations based on different workload scenarios.

The recommendation process and customization of the workload are controlled by SQLAccess Advisor parameters. These parameters control various aspects of the recommendation process, such as the type of recommendation that is required and

the naming conventions for what it recommends. With respect to the workload, parameters control how long the workload exists and what filtering is to be applied to the workload.

To set these parameters, use the SET_TASK_PARAMETER and SET_SQLWKLD_PARAMETER procedures. Parameters are persistent in that they remain set for the lifespan of the task or workload object. When a parameter value is set using the SET_TASK_PARAMETER procedure, it does not change until you make another call to SET_TASK_PARAMETER.

See "Defining the Contents of a Workload" on page 17-14 for more information about workloads.

Step 3 Generate the recommendations

Once a task exists and a workload is linked to the task and the appropriate parameters are set, you can generate recommendations using the DBMS_ADVISOR.EXECUTE_TASK procedure. These recommendations are stored in the SQLAccess Advisor Repository.

The recommendation process generates a number of recommendations and each recommendation will comprise of one or more actions. For example, create a materialized view and then analyze it to gather statistical information.

A task recommendation can range from a simple suggestion to a complex solution that requires implementation of a set of database objects such as indexes, materialized views, and materialized view logs. When an Advisor task is executed, it carefully analyzes collected data and user-adjusted task parameters. The SQLAccess Advisor will then attempt to form a resolution based on its built-in knowledge. The resolutions are then refined and stored in the form of a structured recommendation that can be viewed and implemented by the user.

See "Generating Recommendations" on page 17-25 for more information about generating recommendations.

Step 4 View and implement the recommendations

There are two ways to view the recommendations from the SQLAccess Advisor: using the catalog views or by generating a script using the DBMS_ADVISOR.GET_TASK_SCRIPT procedure. In Enterprise Manager, the recommendations may be displayed once the SQLAccess Advisor process has completed.

See "Viewing the Recommendations" on page 17-26 for a description of using the catalog views to view the recommendations. See "Generating SQL Scripts" on page 17-34 to see how to create a script.

Not all recommendations have to be accepted and you can mark the ones that should be included in the recommendation script.

The final step is then implementing the recommendations and verifying that query performance has improved.

SQLAccess Advisor Repository

All the information needed and generated by the SQLAccess Advisor is held in the Advisor repository, which is a part of the database dictionary. The benefits of using the repository are that it:

- Collects a complete workload for the SQLAccess Advisor.
- Supports historical data.
- Is managed by the server.

Using the SQLAccess Advisor

This section discusses the steps in using the SQLAccess Advisor, and includes:

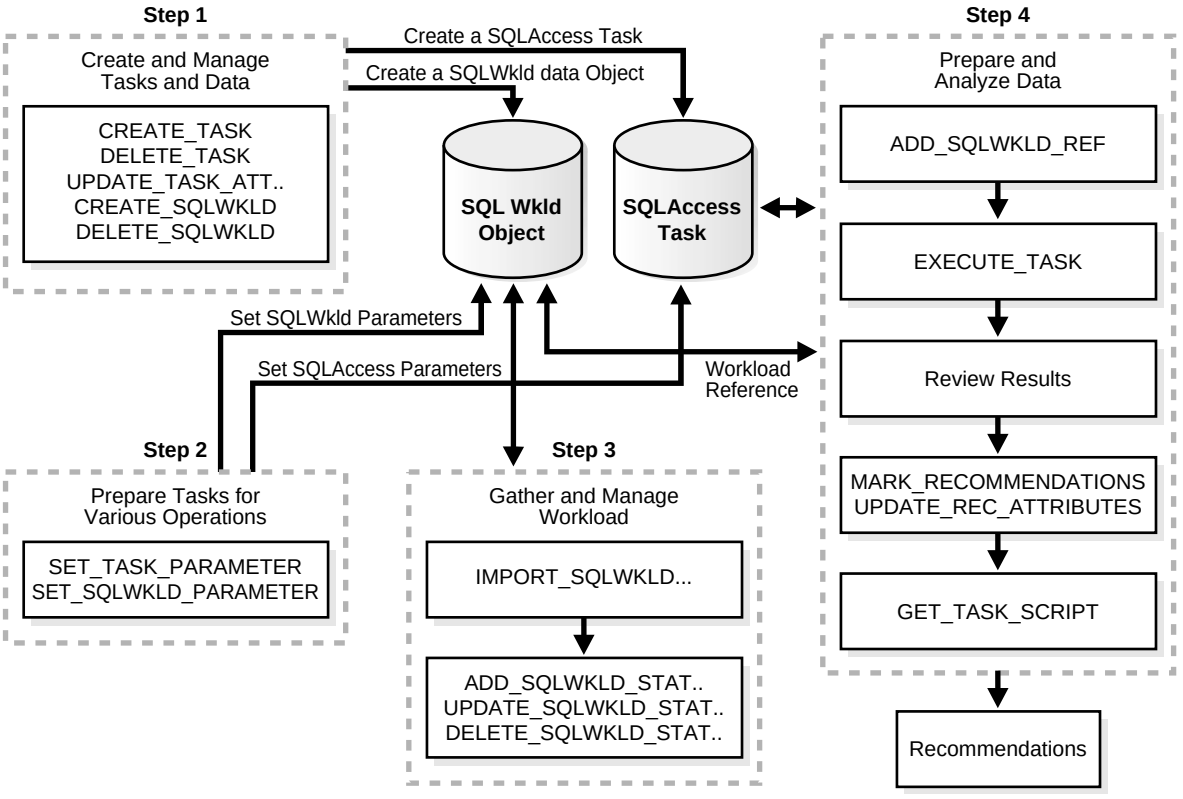
- SQLAccess Advisor Flowchart
- SQLAccess Advisor Privileges
- Creating Tasks
- SQLAccess Advisor Templates
- Creating Templates
- Workload Objects
- Managing Workloads
- Linking a Task and a Workload
- Defining the Contents of a Workload
- SQL Workload Journal
- Adding SQL Statements to a Workload
- Deleting SQL Statements from a Workload
- Changing SQL Statements in a Workload
- Maintaining Workloads
- Removing Workloads

- Recommendation Options
- Generating Recommendations
- Viewing the Recommendations
- Access Advisor Journal
- Stopping the Recommendation Process
- Marking Recommendations
- Modifying Recommendations
- Generating SQL Scripts
- When Recommendations are No Longer Required
- Performing a Quick Tune
- Managing Tasks
- Using SQLAccess Advisor Constants

SQLAccess Advisor Flowchart

Figure 17–2 illustrates the steps in using the SQLAccess Advisor as well as an overview of all of the parameters in the SQLAccess Advisor and when it is appropriate to use them.

Figure 17–2 SQLAccess Advisor Flowchart



SQLAccess Advisor Privileges

You need to have the ADVISOR privilege to manage or use the SQLAccess Advisor. When processing a workload, the SQLAccess Advisor attempts to validate each statement in order to identify table and column references. Validation is achieved by processing each statement as if it is being executed by the statement's original user. If that user does not have SELECT privileges to a particular table, the SQLAccess Advisor bypasses the statement referencing the table. This can cause many statements to be excluded from analysis. If the SQLAccess Advisor excludes all statements in a workload, the workload is invalid and the SQLAccess Advisor returns the following message:

```
QSM-00774, there are no SQL statements to process for task TASK_NAME
```

To avoid missing critical workload queries, the current database user must have SELECT privileges on the tables targeted for materialized view analysis. For those tables, these SELECT privileges cannot be obtained through a role.

Creating Tasks

An Advisor task is where you define what it is you want to analyze and where the results of this analysis should go. A user can create any number of tasks, each with its own specialization. All are based on the same Advisor task model and share the same repository.

You create a task using the CREATE_TASK procedure. The syntax is as follows:

```
DBMS_ADVISOR.CREATE_TASK (
   advisor_name          IN VARCHAR2,
   task_id               OUT NUMBER,
   task_name             IN OUT VARCHAR2,
   task_desc             IN VARCHAR2 := NULL,
   task_or_template      IN VARCHAR2 := NULL,
   is_template           IN VARCHAR2 := 'FALSE');
```

The following illustrates an example of using this procedure:

```
VARIABLE task_id NUMBER;
VARIABLE task_name VARCHAR2(255);
EXECUTE :task_name := 'MYTASK';
EXECUTE DBMS_ADVISOR.CREATE_TASK ('SQL Access Advisor', :task_id, :task_name);
```

See *PL/SQL Packages and Types Reference* for more information regarding the CREATE_TASK and CREATE_SQLWKLD procedures and their parameters.

SQLAccess Advisor Templates

When an ideal configuration for a task or workload has been identified, this configuration can be saved as a template upon which future tasks and workloads can be based.

This enables you to set up any number of tasks or workloads that can be used as intelligent starting points or templates for future task creation. By setting up a template, you can save time when performing tuning analysis. It also enables you to custom fit a tuning analysis to the business operation.

To create a task from a template, you specify the template to be used when a new task is created. At that time, the SQLAccess Advisor copies the data and parameter settings from the template into the newly created task. You can also set an existing

task to be a template by setting the template attribute when creating the task or later using the UPDATE_TASK_ATTRIBUTE procedure.

To use a task as a template, you tell the SQLAccess Advisor to use a task when a new task is created. At that time, the SQLAccess Advisor copies the task template's data and parameter settings into the newly created task. You can also set an existing task to be a template by setting the template attribute.

A workload object can also be used as a template for creating new workload objects. Following the same guidelines for using a task as a template, a workload object can benefit from having a well-defined starting point. Like a task template, a template workload object can only be used to create similar workload objects.

Creating Templates

You can create a template as in the following example.

1. Create a template called MY_TEMPLATE.

```
VARIABLE template_id NUMBER;
VARIABLE template_name VARCHAR2(255);
EXECUTE :template_name := 'MY_TEMPLATE';
EXECUTE DBMS_ADVISOR.CREATE_TASK('SQL Access Advisor',:template_id, -
                                 :template_name, is_template => 'TRUE');
```

2. Set template parameters. For example, the following sets the naming conventions for recommended indexes and materialized views and the default tablespaces:

```
-- set naming conventions for recommended indexes/mvs
EXECUTE DBMS_ADVISOR.SET_TASK_PARAMETER ( -
   :template_name, 'INDEX_NAME_TEMPLATE', 'SH_IDX$$_<SEQ>');

EXECUTE DBMS_ADVISOR.SET_TASK_PARAMETER ( -
   :template_name, 'MVIEW_NAME_TEMPLATE', 'SH_MV$$_<SEQ>');

-- set default tablespace for recommended indexes/mvs
EXECUTE DBMS_ADVISOR.SET_TASK_PARAMETER ( -
   :template_name, 'DEF_INDEX_TABLESPACE', 'SH_INDEXES');

EXECUTE DBMS_ADVISOR.SET_TASK_PARAMETER ( -
   :template_name, 'DEF_MVIEW_TABLESPACE', 'SH_MVIEWS');
```

3. This template can now be used as a starting point to create a task as follows:

```
VARIABLE task_id NUMBER;
```

```
VARIABLE task_name VARCHAR2(255);
EXECUTE :task_name := 'MYTASK';
EXECUTE DBMS_ADVISOR.CREATE_TASK('SQL Access Advisor', :task_id, -
                                 :task_name, template=>'MY_TEMPLATE');
```

The following example uses a pre-defined template SQLACCESS_WAREHOUSE. See Table 17–4 for more information.

```
EXECUTE DBMS_ADVISOR.CREATE_TASK('SQL Access Advisor', -
   :task_id, :task_name,  template=>'SQLACCESS_WAREHOUSE');
```

Workload Objects

Because the workload is stored as a separate workload object, it can easily be shared among many Advisor tasks. Once a workload object has been referenced by an Advisor task, a workload object cannot be deleted or modified until all Advisor tasks have removed their dependency on the data. A workload reference will be removed when a parent Advisor task is deleted or when the workload reference is manually removed from the Advisor task by the user.

The SQLAccess Advisor performs best when a workload based on usage is available. The SQLAccess Workload Repository is capable of storing multiple workloads, so that the different uses of a real-world data warehousing or transaction processing environment can be viewed over a long period of time and across the life cycle of database instance startup and shutdown.

Before the actual SQL statements for a workload can be defined, the workload must be created using the CREATE_SQLWKLD procedure. Then, the workload is loaded using the appropriate IMPORT_SQLWKLD procedure. A specific workload can be removed by calling the DELETE_SQLWKLD procedure and passing it a valid workload name. To remove all workloads for the current user, call DELETE_SQLWKLD and pass the constant value ADVISOR_ALL or %.

Managing Workloads

The CREATE_SQLWKLD procedure creates a workload and it must exist prior to performing any other workload operations, such as importing or updating SQL statements. The workload is identified by its name, so you should define a name that is unique and is relevant for the operation.

Its syntax is as follows:

```
DBMS_ADVISOR.CREATE_SQLWKLD (
   workload_name            IN VARCHAR2,
   description              IN VARCHAR2 := NULL,
```

```
   template                IN VARCHAR2 := NULL.
   is_template             IN BOOLEAN);
```

The following examples illustrate using this procedure:

Example 17–1 Creating a Workload

```
VARIABLE workload_name VARCHAR2(255);
EXECUTE :workload_name := 'MYWORKLOAD';
EXECUTE DBMS_ADVISOR.CREATE_SQLWKLD(:workload_name,'This is my first workload');
```

Example 17–2 Creating a Workload from a Template

1. Create the variables.

   ```
   VARIABLE template_id NUMBER;
   VARIABLE template_name VARCHAR2(255);
   ```

2. Create a template called MY_WK_TEMPLATE.

   ```
   EXECUTE :template_name := 'MY_WK_TEMPLATE';
   EXECUTE DBMS_ADVISOR.CREATE_SQLWKLD(:template_name, is_template=>'TRUE');
   ```

3. Set template parameters. For example, the following sets the filter so only tables in the sh schema are tuned:

   ```
   -- set USERNAME_LIST filter to SH
   EXECUTE DBMS_ADVISOR.SET_SQLWKLD_PARAMETER( -
      :template_name, 'USERNAME_LIST', 'SH');
   ```

4. Now create a workload using the template:

   ```
   VARIABLE workload_name VARCHAR2(255);
   EXECUTE :workload_name := 'MYWORKLOAD';
   EXECUTE DBMS_ADVISOR.CREATE_SQLWKLD ( -
      :workload_name, 'This is my first workload', 'MY_WK_TEMPLATE');
   ```

See *PL/SQL Packages and Types Reference* for more information regarding the CREATE_SQLWKLD procedure and its parameters.

Linking a Task and a Workload

Before the recommendation process can begin, the task must be linked to a workload. You achieve this by using the ADD_SQKLWKLD_REF procedure and the task and workload are linked by using their respective names. This procedure establishes a link between the Advisor task and a workload. Once a connection

between an Advisor task and a workload is made, the workload is protected from removal. The syntax is as follows:

```
DBMS_ADVISOR.ADD_SQLWKLD_REF (task_name      IN VARCHAR2,
                              workload_name  IN VARCHAR2);
```

The following example links the MYTASK task created to the MYWORKLOAD SQL workload.

```
EXECUTE DBMS_ADVISOR.ADD_SQLWKLD_REF('MYTASK', 'MYWORKLOAD');
```

See *PL/SQL Packages and Types Reference* for more information regarding the ADD_SQLWKLD_REF procedure and its parameters.

Defining the Contents of a Workload

Once a workload has been created, it must then be populated with information. Ideally, a workload will consist of the SQL statements (unless it is a hypothetical workload) that are being used against the database. The SQLAccess Advisor can obtain its workload from the following sources:

- SQL Tuning Set
- Loading a User-Defined Workload
- Loading a SQL Cache Workload
- Using a Hypothetical Workload
- Using a Summary Advisor 9i Workload

SQL Tuning Set

A SQL Tuning Set is the workload from the workload repository. You can use a SQL Tuning Set as the workload for the SQLAccess Advisor by importing it using the IMPORT_WORKLOAD_STS procedure. The syntax of this procedure is as follows:

```
DBMS_ADVISOR.IMPORT_SQLWKLD_STS (workload_name  IN VARCHAR2,
                                 sqlset_name    IN VARCHAR2,
                                 import_mode    IN VARCHAR2 := 'NEW',
                                 priority       IN NUMBER := 2,
                                 saved_rows     OUT NUMBER,
                                 failed_rows    OUT NUMBER);
```

After a workload has been collected and the statements filtered, the SQLAccess Advisor computes usage statistics with respect to the DML statements in the workload.

The following example creates a workload from a SQL Tuning Set named MY_STS_WORKLOAD.

```
VARIABLE sqlsetname  VARCHAR2(30);
VARIABLE workload_name VARCHAR2(30);
VARIABLE saved_stmts NUMBER;
VARIABLE failed_stmts NUMBER;
EXECUTE   :sqlsetname := 'MY_STS_WORKLOAD';
EXECUTE   :workload_name := 'MY_WORKLOAD';
EXECUTE DBMS_ADVISOR.CREATE_SQLWKLD (:workload_name);
EXECUTE DBMS_ADVISOR.IMPORT_SQLWKLD_STS (:workload_name , -
     :sqlsetname, 'NEW', 1, :saved_stmts, :failed_stmts);
```

Loading a User-Defined Workload

To load a user-defined workload, use the IMPORT_SQLWKLD_USER procedure. This procedure collects an application workload from a user-constructed table or view and saves it in the Advisor repository. The two parameters, owner_name and table_name identify the table where the workload is to be retrieved from.

There are no restrictions on which schema the workload resides in, the name for the table, or how many of these user-defined tables exist. The only requirements are that the format of the user table must correspond to the USER_WORKLOAD table, as described in Table 17–1, and that the user have SELECT access to the workload table or view. The syntax is as follows:

```
DBMS_ADVISOR.IMPORT_SQLWKLD_USER (
   workload_name         IN VARCHAR2,
   import_mode           IN VARCHAR2,
   owner_name            IN VARCHAR2,
   table_name            IN VARCHAR2,
   saved_rows            OUT NUMBER,
   failed_rows           OUT NUMBER);
```

The following example loads MYWORKLOAD workload created earlier, using a user table SH.USER_WORKLOAD. The table is assumed to be populated with SQL statements and conforms to the format specified in Table 17–1.

```
VARIABLE saved_stmts NUMBER;
VARIABLE failed_stmts NUMBER;
EXECUTE DBMS_ADVISOR.IMPORT_SQLWKLD_USER( -
   'MYWORKLOAD', 'NEW', 'SH', 'USER_WORKLOAD', :saved_stmts, :failed_stmts);
```

See *PL/SQL Packages and Types Reference* for more information regarding the IMPORT_SQLWKLD_USER procedure and its parameters.

Table 17–1 USER_WORKLOAD Table Format

Column	Type	Default	Comments
MODULE	VARCHAR2(64)	Empty string	Application module name.
ACTION	VARCHAR2(64)	Empty string	Application action.
BUFFER_GETS	NUMBER	0	Total buffer-gets for the statement.
CPU_TIME	NUMBER	0	Total CPU time in seconds for the statement.
ELAPSED_TIME	NUMBER	0	Total elapsed time in seconds for the statement.
DISK_READS	NUMBER	0	Total number of disk-read operations used by the statement.
ROWS_PROCESSED	NUMBER	0	Total number of rows process by this SQL statement.
EXECUTIONS	NUMBER	1	Total number of times the statement is executed.
OPTIMIZER_COST	NUMBER	0	Optimizer's calculated cost value for executing the query.
LAST_EXECUTION_DATE	DATE	SYSDATE	Last time the query is used. Defaults to not available.
PRIORITY	NUMBER	2	Must be one of the following values: 1- HIGH, 2- MEDIUM, or 3- LOW
SQL_TEXT	CLOB or LONG or VARCHAR2	None	The SQL statement. This is a required column.
STAT_PERIOD	NUMBER	1	Period of time that corresponds to the execution statistics in seconds.
USERNAME	VARCHAR(30)	Current user	User submitting the query. This is a required column.

Loading a SQL Cache Workload

You obtain a SQL cache workload using the procedure IMPORT_SQLWKLD_SQLCACHE. At the time this procedure is called, the current contents of the SQL cache are analyzed and placed into the workload. The IMPORT_SQLWKLD_SQLCACHE procedure loads a SQL workload from the SQL cache. The syntax is as follows:

```
DBMS_ADVISOR.IMPORT_SQLWKLD_SQLCACHE (
   workload_name         IN VARCHAR2,
   import_mode           IN VARCHAR2,
   priority              IN NUMBER := 2,
```

```
   saved_rows            OUT NUMBER,
   failed_rows           OUT NUMBER);
```

See *PL/SQL Packages and Types Reference* for more information regarding the IMPORT_SQLWKLD_SQLCACHE procedure and its parameters.

The following example loads the MYWORKLOAD workload created earlier from the SQL Cache. The priority of the loaded workload statements is 2 (medium).

```
VARIABLE saved_stmts NUMBER;
VARIABLE failed_stmts NUMBER;
EXECUTE DBMS_ADVISOR.IMPORT_SQLWKLD_SQLCACHE (-
   'MYWORKLOAD', 'APPEND', 2, :saved_stmts, :failed_stmts);
```

The SQLAccess Advisor can retrieve workload information from the SQL cache. If the collected data was retrieved from a server with the instance parameter cursor_sharing set to SIMILAR or FORCE, then user queries with embedded literal values will be converted to a statement that contains system-generated bind variables. If you are going to use the SQLAccess Advisor to recommend materialized views, then the server should set the instance parameter cursor_sharing to EXACT so that materialized views with WHERE clauses can be recommended.

Using a Hypothetical Workload

In many situations, an application workload may not yet exist. In this case, the SQLAccess Advisor can examine the current logical schema design and form recommendations based on defined relationships among tables. This type of workload is also referred to as a hypothetical workload. The SQLAccess Advisor can produce an initial set of recommendations and become a solid base for tuning an application.

The benefits of using hypothetical workloads are that they:

- Require only schema and table relationships.
- Are effective for modeling what-if scenarios.

Some of the disadvantages of hypothetical workloads are that they:

- Only work if dimensions or primary and foreign key constraints have been defined.
- Offer no information about the impact of DML on the recommended access structures.
- Are not necessarily complete.

To successfully import a hypothetical workload, the target schemas must contain dimension or primary and foreign key information. You use the IMPORT_SQLWKLD_SCHEMA procedure. The syntax is as follows:

```
DBMS_ADVISOR.IMPORT_SQLWKLD_SCHEMA (
   workload_name         IN VARCHAR2,
   import_mode           IN VARCHAR2,
   priority              IN VARCHAR2,
   saved_rows            OUT NUMBER,
   failed_rows           OUT NUMBER);
```

See *PL/SQL Packages and Types Reference* for more information regarding the IMPORT_SQLWKLD_SCHEMA procedure and its parameters. You must configure external procedures to use this procedure.

The following example creates a hypothetical workload called SCHEMA_WKLD, sets VALID_TABLE_LIST to sh and calls IMPORT_SQLWKLD_SCHEMA to generate a hypothetical workload.

```
VARIABLE workload_name VARCHAR2(255);
VARIABLE saved_stmts NUMBER;
VARIABLE failed_stmts NUMBER;
EXECUTE :workload_name := 'SCHEMA_WKLD';
EXECUTE DBMS_ADVISOR.CREATE_SQLWKLD(:workload_name);
EXECUTE DBMS_ADVISOR.SET_SQLWKLD_PARAMETER (:workload_name, -
                                            VALID_TABLE_LIST, 'SH');
EXECUTE DBMS_ADVISOR.IMPORT_SQLWKLD_SCHEMA ( -
  :workload_name, 'NEW', 2, :saved_stmts, :failed_stmts);
```

When using IMPORT_SQLWKLD_SCHEMA, the VALID_TABLE_LIST parameter cannot contain wildcards such as SCO% or SCOTT.EMP%. The only form of wildcards supported is SCOTT.%, which specifies all tables in a given schema.

Using a Summary Advisor 9i Workload

You may have created workloads using the Oracle9*i* Summary Advisor. These workloads can be used by the SQLAccess Advisor by importing them using the IMPORT_SQLWLD_SUMADV procedure. To use this procedure, you must know the Oracle9*i* workload ID.

This procedure collects a SQL workload from a Summary Advisor workload. This procedure is intended to assist Oracle9*i* Summary Advisor users in the migration to SQLAccess Advisor. The syntax is as follows:

```
DBMS_ADVISOR.IMPORT_SQLWKLD_SUMADV (
   workload_name         IN VARCHAR2,
```

```
   import_mode           IN VARCHAR2,
   priority              IN NUMBER := 2,
   sumadv_id             IN NUMBER,
   saved_rows            OUT NUMBER,
   failed_rows           OUT NUMBER);
```

See *PL/SQL Packages and Types Reference* for more information regarding the IMPORT_SQLWKLD_SUMADV procedure and its parameters.

The following example creates a SQL workload from a Oracle9*i* Summary Advisor workload. The workload_id of the Oracle9*i* workload is 777.

1. Create some variables.

```
VARIABLE workload_name VARCHAR2(255);
VARIABLE saved_stmts NUMBER;
VARIABLE failed_stmts NUMBER;
```

2. Create a workload named WKLD_9I.

```
EXECUTE :workload_name := 'WKLD_9I';
EXECUTE DBMS_ADVISOR.CREATE_SQLWKLD(:workload_name);
```

3. Import the workload from Oracle9*i* Summary Advisor.

```
EXECUTE DBMS_ADVISOR.IMPORT_SQLWKLD_SUMADV ( -
   :workload_name, 'NEW', 2, 777, :saved_stmts, :failed_stmts);
```

SQLAccess Advisor Workload Parameters

A SQL workload can be filtered at the time of loading by setting one of more of the parameters listed in *PL/SQL Packages and Types Reference* using SET_SQLWKLD_PARAMETER.

The following example illustrates setting of SQL Workload parameters. Here we set the SQL_LIMIT to 3 and ORDER_LIST to OPTIMIZER_COST. This means that when importing the workload, the statements will be ordered by OPTIMIZER_COST and the top three statements will be kept.

```
-- Order statements by OPTIMIZER_COST
EXECUTE DBMS_ADVISOR.SET_SQLWKLD_PARAMETER ( -
   'MYWORKLOAD', 'ORDER_LIST', 'OPTIMIZER_COST');
-- Max number of statements 3
EXECUTE DBMS_ADVISOR.SET_SQLWKLD_PARAMETER('MYWORKLOAD', 'SQL_LIMIT', 3);
```

SQL Workload Journal

During the import of a workload, various informational messages are recorded in the SQL Workload Journal. These can be viewed using the view USER_ADVISOR_SQLW_JOURNAL. The journal is useful to identify why some statements were filtered out of the workload. For example, if a certain SQL statement refers to invalid tables, tables with missing statistics or has privilege errors, the information will be recorded in the journal. The amount of information output can be controlled by setting the JOURNALING parameter.

You can turn journaling off before importing workload as follows:

```
EXECUTE DBMS_ADVISOR.SET_SQLWKLD_PARAMETER('MYWORKLOAD', 'JOURNALING', 0);
```

To view only fatal messages:

```
EXECUTE DBMS_ADVISOR.SET_SQLWKLD_PARAMETER('MYWORKLOAD', 'JOURNALING', 4);
```

See *PL/SQL Packages and Types Reference* for details of all the settings for the JOURNALING parameter.

The information in the journal is for diagnostic purposes only and subject to change in future releases. It should not be used within any application.

Adding SQL Statements to a Workload

An alternative to importing a workload is to manually specify the SQL statements and add them to your workload using the ADD_SQLWKLD_STATEMENT procedure. This procedure adds a SQL statement to the specified workload. The syntax is as follows:

```
DBMS_ADVISOR.ADD_SQLWKLD_STATEMENT (workload_name    IN VARCHAR2,
                                    module           IN VARCHAR2,
                                    action           IN VARCHAR2,
                                    cpu_time         IN NUMBER := 0,
                                    elapsed_time     IN NUMBER := 0,
                                    disk_reads       IN NUMBER := 0,
                                    buffer_gets      IN NUMBER := 0,
                                    rows_processed   IN NUMBER := 0,
                                    optimizer_cost   IN NUMBER := 0,
                                    executions       IN NUMBER := 1,
                                    priority         IN NUMBER := 2,
                                    last_execution_date IN DATE := 'SYSDATE',
                                    stat_period   IN NUMBER := 0,
                                    username  IN VARCHAR2,
                                    sql_text  IN CLOB);
```

See *PL/SQL Packages and Types Reference* for more information regarding the ADD_SQLWKLD_STATEMENT procedure and its parameters. The following example adds a single statement to the MYWORKLOAD workload.

```
VARIABLE sql_text VARCHAR2(400);
EXECUTE :sql_text := 'SELECT AVG(amount_sold) FROM sales';
EXECUTE DBMS_ADVISOR.ADD_SQLWKLD_STATEMENT ( -
   'MYWORKLOAD', 'MONTHLY', 'ROLLUP', priority=>1, executions=>10, -
    username => 'SH',  sql_text => :sql_text);
```

Deleting SQL Statements from a Workload

You can delete an existing SQL statement from a specified workload using the DELETE_SQLWKLD_STATEMENT procedure. Its syntax has two forms. The first form lets you delete a statement specified by a given sql_id.

```
DBMS_ADVISOR.DELETE_SQLWKLD_STATEMENT (workload_name  IN VARCHAR2,
                                       sql_id         IN NUMBER);
```

The second form lets you delete statements that match a specified search condition.

```
DBMS_ADVISOR.DELETE_SQLWKLD_STATEMENT (workload_name  IN  VARCHAR2,
                                       search         IN  VARCHAR2,
                                       deleted        OUT NUMBER);
```

The following example deletes from MYWORKLOAD with sql_id 10:

```
EXECUTE DBMS_ADVISOR.DELETE_SQLWKLD_STATEMENT('MYWORKLOAD', 10);
```

The following example deletes from MYWORKLOAD all statements that satisfy the condition executions less than 5:

```
VARIABLE deleted_stmts NUMBER;
EXECUTE DBMS_ADVISOR.DELETE_SQLWKLD_STATEMENT ( -
   'MYWORKLOAD', 'executions < 5', :deleted_stmts);
```

A workload cannot be modified or deleted if it is currently referenced by an active task. A task is considered active if it is not in its initial state. See the RESET_TASK procedure to set a task to its initial state. See *PL/SQL Packages and Types Reference* for more information regarding the DELETE_SQLWKLD_STATEMENT procedure and its parameters.

Changing SQL Statements in a Workload

You can modify SQL statements in a workload by using the UPDATE_SQLWKLD_STATEMENT procedure. This procedure updates an existing SQL statement in the specified workload.

The syntax takes two forms. The first form enables you to update a SQL statement by specifying its sql_id.

```
DBMS_ADVISOR.UPDATE_SQLWKLD_STATEMENT (workload_name  IN VARCHAR2,
                                       sql_id         IN NUMBER,
                                       module         IN VARCHAR2,
                                       action         IN VARCHAR2,
                                       priority       IN NUMBER,
                                       username       IN VARCHAR2);
```

The second forms enables you to update all SQL statements satisfying a given search condition.

```
DBMS_ADVISOR.UPDATE_SQLWKLD_STATEMENT (workload_name  IN VARCHAR2,
                                       search         IN VARCHAR2,
                                       updated        OUT NUMBER,
                                       module         IN VARCHAR2,
                                       action         IN VARCHAR2,
                                       priority       IN NUMBER,
                                       username       IN VARCHAR2);
```

The following example changes the priority to 3 for statement id 10:

```
EXECUTE DBMS_ADVISOR.UPDATE_SQLWKLD_STATEMENT('MYWORKLOAD', 10, priority=>3);
```

The following examples changes the priority to 3 for all statements in MYWORKLOAD that have executions less than 10. The count of updated statements is returned in the updated_stmts variable.

```
VARIABLE updated_stmts NUMBER;
EXECUTE DBMS_ADVISOR.UPDATE_SQLWKLD_STATEMENT ( -
   'MYWORKLOAD', 'executions < 10', :updated_stmts, priority => 3);
```

See *PL/SQL Packages and Types Reference* for more information regarding the UPDATE_SQLWKLD_STATEMENT procedure and its parameters.

Maintaining Workloads

There are several other operations that can be performed upon a workload, including the following:

- Setting Workload Attributes
- Resetting Workloads
- Removing a Link Between a Workload and a Task

Setting Workload Attributes

The UPDATE_SQLWKLD_ATTRIBUTES procedure changes various attributes of a workload object or template. Some of these attributes are its description, and whether it is a template or read only. The syntax is as follows:

```
DBMS_ADVISOR.UPDATE_SQLWKLD_ATTRIBUTES (
   workload_name        IN VARCHAR2,
   new_name             IN VARCHAR2 := NULL,
   description          IN VARCHAR2 := NULL,
   read_only            IN VARCHAR2 := NULL,
   is_template          IN VARCHAR2 := NULL,
   source               IN VARCHAR2 := NULL);
```

The following example changes the workload MYWORKLOAD to read-only:

```
EXECUTE DBMS_ADVISOR.UPDATE_SQLWKLD_ATTRIBUTES ( -
   'MYWORKLOAD', read_only=> 'TRUE');
```

See *PL/SQL Packages and Types Reference* for more information regarding the UPDATE_SQLWKLD_ATTRIBUTES procedure and its parameters.

Resetting Workloads

The RESET_SQLWKLD procedure resets a workload to its initial starting point. This has the effect of removing all journal and log messages, recalculating volatility statistics, while the workload data remains untouched. This procedure should be executed after any workload adjustments such as adding or removing SQL statements. The following example resets workload MYWORKLOAD.

```
EXECUTE DBMS_ADVISOR.RESET_SQLWKLD('MYWORKLOAD');
```

See *PL/SQL Packages and Types Reference* for more information regarding the RESET_SQLWKLD procedure and its parameters.

Removing a Link Between a Workload and a Task

Before a task or a workload can be deleted, if it is linked to a workload or task respectively, then the link between the task and the workload must be removed

using DELETE_SQLWKLD_REF procedure. The following example deletes the link between task MYTASK and SQL workload MYWORKLOAD:

```
EXECUTE DBMS_ADVISOR.DELETE_SQLWKLD_REF('MYTASK', 'MYWORKLOAD');
```

Removing Workloads

When workloads are no longer needed, they can be removed using the procedure DELETE_SQLWKLD. You can delete all workloads or a specific collection, but a workload cannot be deleted if it is still linked to a task.

The following procedure is an example of removing a specific workload. It deletes an existing workload from the repository.

```
DBMS_ADVISOR.DELETE_SQLWKLD (workload_name  IN VARCHAR2);
EXECUTE DBMS_ADVISOR.DELETE_SQLWKLD('MYWORKLOAD');
```

See *PL/SQL Packages and Types Reference* for more information regarding the DELETE_SQLWKLD procedure and its parameters.

Recommendation Options

Before recommendations can be generated, the parameters for the task must first be defined using the SET_TASK_PARAMETER procedure. If parameters are not defined, then the defaults are used.

You can set task parameters by using the SET_TASK_PARAMETER procedure. The syntax is as follows.

```
DBMS_ADVISOR.SET_TASK_PARAMETER (
   task_name           IN VARCHAR2
   parameter           IN VARCHAR2,
   value               IN VARCHAR2);
```

There are many task parameters and, to help identify the relevant ones, they have been grouped into categories in Table 17–2.

Table 17–2 Types of Advisor Task Parameters And Their Uses

Workload Filtering	Task Configuration	Schema Attributes	Recommendation Options
ACTION_LIST	DAYS_TO_EXPIRE	DEF_INDEX_OWNER	DML_VOLATILITY
MODULE_LIST	REPORT_DATE_FORMAT	DEF_INDEX_TABLESPACE	EVALUATION_ONLY
ORDER_LIST	JOURNALING	DEF_MVIEW_OWNER	EXECUTION_TYPE

Table 17–2 (Cont.) Types of Advisor Task Parameters And Their Uses

Workload Filtering	Task Configuration	Schema Attributes	Recommendation Options
SQL_LIMIT		DEF_MVIEW_TABLESPACE	MODE
START_TIME		DEF_MVLOG_TABLSPACE	REFRESH_MODE
USERNAME_LIST		INDEX_NAME_TEMPLATE	STORAGE_CHANGE
VALID_TABLE_LIST		MVIEW_NAME_TEMPLATE	CREATION_COST
WORKLOAD_SCOPE			
MODULE_LIMIT			
TIME_LIMIT			
END_TIME			
COMMENTED_FILTER_LIST			

In the following example, set the storage change of task MYTASK to 100MB. This indicates 100MB of additional space for recommendations. A zero value would indicate that no additional space can be allocated. A negative value indicates that the advisor must attempt to trim the current space utilization by the specified amount.

```
EXECUTE DBMS_ADVISOR.SET_TASK_PARAMETER('MYTASK','STORAGE_CHANGE', 100000000);
```

In the following example, we set the VALID_TABLE_LIST parameter to filter out all queries that do no consist of tables SH.SALES and SH.CUSTOMERS.

```
EXECUTE DBMS_ADVISOR.SET_TASK_PARAMETER ( -
   'MYTASK', 'VALID_TABLE_LIST', 'SH.SALES, SH.CUSTOMERS');
```

See *PL/SQL Packages and Types Reference* for more information regarding the SET_TASK_PARAMETER procedure and its parameters.

Generating Recommendations

You can generate recommendations by using the EXECUTE_TASK procedure with your task name. After the procedure finishes, you can check the DBA_ADVISOR_LOG table for the actual execution status and the number of recommendations and actions that have been produced. The recommendations can be queried by task name in {DBA, USER}_ADVISOR_RECOMMENDATIONS and the actions for these recommendations can be viewed by task in {DBA, USER}_ADVISOR_ACTIONS.

EXECUTE_TASK Procedure

This procedure performs the SQLAccess Advisor analysis or evaluation for the specified task. Task execution is a synchronous operation, so control will not be returned to the user until the operation has completed, or a user-interrupt was detected. Upon return or execution of the task, you can check the DBA_ADVISOR_LOG table for the actual execution status.

Running EXECUTE_TASK generates recommendations, where a recommendation comprises one or more actions, such as creating a materialized view log and a materialized view. The syntax is as follows:

```
DBMS_ADVISOR.EXECUTE_TASK (task_name   IN VARCHAR2);
```

The following illustrates an example of using this procedure:

```
EXECUTE DBMS_ADVISOR.EXECUTE_TASK('MYTASK');
```

See *PL/SQL Packages and Types Reference* for more information regarding the EXECUTE_TASK procedure and its parameters.

Viewing the Recommendations

Each recommendation generated by the SQLAccess Advisor can be viewed using several catalog views, such as (DBA, USER)_ADVISOR_RECOMMENDATIONS. However, it is easier to use the GET_TASK_SCRIPT procedure or use the SQLAccess Advisor in Enterprise Manager, which graphically displays the recommendations and provides hyperlinks to quickly see which SQL statements benefit from a recommendation. Each recommendation produced by the SQLAccess Advisor is linked to the SQL statement it benefits.

The following shows the recommendation (rec_id) produced by an Advisor run, with their rank and total benefit. The rank is a measure of the importance of the queries that the recommendation helps. The benefit is the total improvement in execution cost (in terms of optimizer cost) of all the queries using the recommendation.

```
VARIABLE workload_name VARCHAR2(255);
VARIABLE task_name VARCHAR2(255);
EXECUTE :task_name := 'MYTASK';
EXECUTE :workload_name := 'MYWORKLOAD';

SELECT REC_ID, RANK, BENEFIT
FROM USER_ADVISOR_RECOMMENDATIONS WHERE TASK_NAME = :task_name;

    REC_ID       RANK    BENEFIT
```

```
---------- ---------- ----------
         1          2       2754
         2          3       1222
         3          1       5499
         4          4        594
```

To identify which query benefits from which recommendation, you can use the views DBA_* and USER_ADVISOR_SQLA_WK_STMTS. The precost and postcost numbers are in terms of the estimated optimizer cost (shown in EXPLAIN PLAN) without and with the recommended access structure changes, respectively. To see recommendations for each query, issue the following statement:

```
SELECT sql_id, rec_id, precost, postcost,
       (precost-postcost)*100/precost AS percent_benefit
FROM USER_ADVISOR_SQLA_WK_STMTS
WHERE TASK_NAME = :task_name AND workload_name = :workload_name;

    SQL_ID     REC_ID    PRECOST   POSTCOST PERCENT_BENEFIT
---------- ---------- ---------- ---------- ---------------
       121          1       3003        249      91.7082917
       122          2       1404        182       87.037037
       123          3       5503          4      99.9273124
       124          4        730        136       81.369863
```

Each recommendation consists of one or more actions, which must be implemented together to realize the benefit provided by the recommendation. The SQLAccess Advisor produces the following types of actions:

- CREATE|DROP|RETAIN MATERIALIZED VIEW
- CREATE|ALTER|RETAIN MATERIALIZED VIEW LOG
- CREATE|DROP|RETAIN INDEX
- GATHER STATS

The CREATE actions corresponds to new access structures. RETAIN recommendation indicate that existing access structures must be kept. DROP recommendations are only produced if the WORKLOAD_SCOPE parameter is set to FULL. The GATHER STATS action will generate a call to DBMS_STATS procedure to gather statistics on a newly generated access structure. Note that multiple recommendations may refer to the same action, however when generating a script for the recommendation, you will only see each action once.

In the following example, you can see how many distinct actions there are for this set of recommendations.

```
SELECT 'Action Count', COUNT(DISTINCT action_id) cnt
FROM user_advisor_actions WHERE task_name = :task_name;

'ACTIONCOUNT        CNT
------------ ----------
Action Count          20

-- see the actions for each recommendations
SELECT rec_id, action_id, SUBSTR(command,1,30) AS command
FROM user_advisor_actions WHERE task_name = :task_name
ORDER BY rec_id, action_id;

    REC_ID  ACTION_ID COMMAND
---------- ---------- ------------------------------
         1          5 CREATE MATERIALIZED VIEW LOG
         1          6 ALTER MATERIALIZED VIEW LOG
         1          7 CREATE MATERIALIZED VIEW LOG
         1          8 ALTER MATERIALIZED VIEW LOG
         1          9 CREATE MATERIALIZED VIEW LOG
         1         10 ALTER MATERIALIZED VIEW LOG
         1         11 CREATE MATERIALIZED VIEW
         1         12 GATHER TABLE STATISTICS
         1         19 CREATE INDEX
         1         20 GATHER INDEX STATISTICS
         2          5 CREATE MATERIALIZED VIEW LOG
         2          6 ALTER MATERIALIZED VIEW LOG
         2          9 CREATE MATERIALIZED VIEW LOG
         ...
```

Each action has several attributes that pertain to the properties of the access structure. The name and tablespace for each access structure when applicable are placed in `attr1` and `attr2` respectively. The space occupied by each new access structure is in `num_attr1`. All other attributes are different for each action.

Table 17–3 maps SQLAccess Advisor action information to the corresponding column in `DBA_ADVISOR_ACTIONS`.

Table 17–3 SQLAccess Advisor Action Attributes

	ATTR1	ATTR2	ATTR3	ATTR4	ATTR5	ATTR6	NUM_ ATTR1
CREATE INDEX	Index name	Index tablespace	Target table	BITMAP or BTREE	Index column list	Unused	Storage size in bytes for the index
CREATE MATERIALIZED VIEW	MV name	MV tablespace	REFRESH COMPLETE REFRESH FAST, REFRESH FORCE, NEVER REFRESH	ENABLE QUERY REWRITE, DISABLE QUERY REWRITE	SQL SELECT statement	Unused	Storage size in bytes for the MV
CREATE MATERIALIZED VIEW LOG	Target table name	MV log tablespace	ROWID PRIMARY KEY, SEQUENCE OBJECT ID	INCLUDING NEW VALUES, EXCLUDING NEW VALUES	Table column list	Partition-ing subclauses	Unused
CREATE REWRITE EQUIVALENCE	Name of equivalence	Checksum value	Unused	Unused	Source SQL statement	Equivalent SQL statement	Unused
DROP INDEX	Index name	Unused	Unused	Unused	Index columns	Unused	Storage size in bytes for the index
DROP MATERIALIZED VIEW	MV name	Unused	Unused	Unused	Unused	Unused	Storage size in bytes for the MV
DROP MATERIALIZED VIEW LOG	Target table name	Unused	Unused	Unused	Unused	Unused	Unused

Table 17–3 (Cont.) SQLAccess Advisor Action Attributes

	ATTR1	ATTR2	ATTR3	ATTR4	ATTR5	ATTR6	NUM_ ATTR1
RETAIN INDEX	Index name	Unused	Target table	BITMAP or BTREE	Index columns	Unused	Storage size in bytes for the index
RETAIN MATERIALIZED VIEW	MV name	Unused	REFRESH COMPLETE or REFRESH FAST	Unused	SQL SELECT statement	Unused	Storage size in bytes for the MV
RETAIN MATERIALIZED VIEW LOG	Target table name	Unused	Unused	Unused	Unused	Unused	Unused

The following PL/SQL procedure can be used to print out some of the attributes of the recommendations.

```
CONNECT SH/SH;
CREATE OR REPLACE PROCEDURE show_recm (in_task_name IN VARCHAR2) IS
CURSOR curs IS
  SELECT DISTINCT action_id, command, attr1, attr2, attr3, attr4
  FROM user_advisor_actions
  WHERE task_name = in_task_name
  ORDER BY action_id;
  v_action        number;
  v_command     VARCHAR2(32);
  v_attr1       VARCHAR2(4000);
  v_attr2       VARCHAR2(4000);
  v_attr3       VARCHAR2(4000);
  v_attr4       VARCHAR2(4000);
  v_attr5       VARCHAR2(4000);
BEGIN
  OPEN curs;
  DBMS_OUTPUT.PUT_LINE('========================================');
  DBMS_OUTPUT.PUT_LINE('Task_name = ' || in_task_name);
  LOOP
     FETCH curs INTO
       v_action, v_command, v_attr1, v_attr2, v_attr3, v_attr4 ;
   EXIT when curs%NOTFOUND;
   DBMS_OUTPUT.PUT_LINE('Action ID: ' || v_action);
   DBMS_OUTPUT.PUT_LINE('Command : ' || v_command);
   DBMS_OUTPUT.PUT_LINE('Attr1 (name)      : ' || SUBSTR(v_attr1,1,30));
```

```
   DBMS_OUTPUT.PUT_LINE('Attr2 (tablespace): ' || SUBSTR(v_attr2,1,30));
   DBMS_OUTPUT.PUT_LINE('Attr3             : ' || SUBSTR(v_attr3,1,30));
   DBMS_OUTPUT.PUT_LINE('Attr4             : ' || v_attr4);
   DBMS_OUTPUT.PUT_LINE('Attr5             : ' || v_attr5);
   DBMS_OUTPUT.PUT_LINE('----------------------------------------');
   END LOOP;
   CLOSE curs;
   DBMS_OUTPUT.PUT_LINE('=========END RECOMMENDATIONS============');
END show_recm;
/
-- see what the actions are using sample procedure
set serveroutput on size 99999
EXECUTE show_recm(:task_name);
A fragment of a sample output from this procedure is as follows:
Task_name = MYTASK
Action ID: 1
Command : CREATE MATERIALIZED VIEW LOG
Attr1 (name)      : "SH"."CUSTOMERS"
Attr2 (tablespace):
Attr3             : ROWID, SEQUENCE
Attr4             :  INCLUDING NEW VALUES
Attr5             :
----------------------------------------
..
----------------------------------------
Action ID: 15
Command : CREATE MATERIALIZED VIEW
Attr1 (name)      : "SH"."SH_MV$$_0004"
Attr2 (tablespace): "SH_MVIEWS"
Attr3             : REFRESH FAST WITH ROWID
Attr4             : ENABLE QUERY REWRITE
Attr5             :
----------------------------------------
..
----------------------------------------
Action ID: 19
Command : CREATE INDEX
Attr1 (name)      : "SH"."SH_IDX$$_0013"
Attr2 (tablespace): "SH_INDEXES"
Attr3             : "SH"."SH_MV$$_0002"
Attr4             : BITMAP
Attr5             :
```

See *PL/SQL Packages and Types Reference* for details regarding `Attr5` and `Attr6`.

Access Advisor Journal

During the analysis process (EXECUTE_TASK), the Access Advisor will save useful information regarding the analysis to a journal. The journal can be viewed using the view USER_ADVISOR_JOURNAL. The amount of information output varies depending on the setting of task parameter JOURNALING.

You can turn journaling off as follows:

```
EXECUTE DBMS_ADVISOR.SET_TASK_PARAMETER('MYTASK', 'JOURNALING', 0);
```

To view only informational messages:

```
EXECUTE DBMS_ADVISOR.SET_TASK_PARAMETER('MYTASK', 'JOURNALING', 1);
```

See *PL/SQL Packages and Types Reference* for details of all the settings for the `journaling` parameter.

The information in the journal is for diagnostic purposes only and subject to change in future releases. It should not be used within any application.

Stopping the Recommendation Process

If the SQLAccess Advisor takes too long to make its recommendations using the procedure EXECUTE_TASK, you can stop it by calling the CANCEL_TASK procedures and passing in the task_name for this recommendation process. If you use CANCEL_TASK, no recommendations will be made.

Canceling Tasks

The CANCEL_TASK procedure causes a currently executing operation to terminate. An Advisor operation may take a few seconds to respond to the call. Because all Advisor task procedures are synchronous, to cancel an operation, you must use a separate database session.

A cancel command effective restores the task to its condition prior to the start of the cancelled operation. Therefore, a cancelled task or data object cannot be restarted.

```
DBMS_ADVISOR.CANCEL_TASK (task_name   IN  VARCHAR2);
```

The following illustrates an example of using this procedure:

```
EXECUTE DBMS_ADVISOR.CANCEL_TASK('MYTASK');
```

See *PL/SQL Packages and Types Reference* for more information regarding the CANCEL_TASK procedure and its parameters.

Marking Recommendations

By default, all SQLAccess Advisor recommendations are ready to be implemented, however, the user can choose to skip or exclude selected recommendations by using the MARK_RECOMMENDATION procedure. MARK_RECOMMENDATION allows the user to annotate a recommendation with a REJECT or IGNORE setting, which will cause the GET_TASK_SCRIPT to skip it when producing the implementation procedure.

```
DBMS_ADVISOR.MARK_RECOMMENDATION (
   task_name          IN VARCHAR2
   id                 IN NUMBER,
   action             IN VARCHAR2);
```

The following example marks a recommendation with ID 2 as REJECT. This recommendation and any dependent recommendations will not appear in the script.

```
EXECUTE DBMS_ADVISOR.MARK_RECOMMENDATION('MYTASK', 2, 'REJECT');
```

See *PL/SQL Packages and Types Reference* for more information regarding the MARK_RECOMMENDATIONS procedure and its parameters.

Modifying Recommendations

Using the UPDATE_REC_ATTRIBUTES procedure, the SQLAccess Advisor names and assigns ownership to new objects such as indexes and materialized views during the analysis operation. However, it does not necessarily choose appropriate names, so you may manually set the owner, name, and tablespace values for new objects. For recommendations referencing existing database objects, owner and name values cannot be changed. The syntax is as follows:

```
DBMS_ADVISOR.UPDATE_REC_ATTRIBUTES (
   task_name            IN VARCHAR2
   rec_id               IN NUMBER,
   action_id            IN NUMBER,
   attribute_name       IN VARCHAR2,
   value                IN VARCHAR2);
```

The attribute_name parameter can take the following values:

- BASE_TABLE

  Specifies the base table reference for the recommended object.

- OWNER

Specifies the owner name of the recommended object.

- NAME

  Specifies the name of the recommended object.

- TABLESPACE

  Specifies the tablespace of the recommended object.

The following example modifies the attribute TABLESPACE for recommendation ID 1, action ID 1 to SH_MVIEWS.

```
EXECUTE DBMS_ADVISOR.UPDATE_REC_ATTRIBUTES('MYTASK', 1, 1, -
                                           'TABLESPACE', 'SH_MVIEWS');
```

See *PL/SQL Packages and Types Reference* for more information regarding the UPDATE_REC_ATTRIBUTES procedure and its parameters.

Generating SQL Scripts

An alternative to querying the metadata to see the recommendations, is to create a script of the SQL statements for the recommendations, using the procedure GET_TASK_SCRIPT. The resulting script is an executable SQL file that can contain DROP, CREATE, and ALTER statements. For new objects, the names of the materialized views, materialized view logs, and indexes are auto-generated by using the user-specified name template. You should review the generated SQL script before attempting to execute it.

There are four task parameters that control the naming conventions (MVIEW_NAME_TEMPLATE and INDEX_NAME_TEMPLATE), the owner for these new objects (DEF_INDEX_OWNER and DEF_MVIEW_OWNER), and the tablespaces (DEF_MVIEW_TABLESPACE and DEF_INDEX_TABLESPACE).

The following example shows how to generate a CLOB containing the script for the recommendations.

```
EXECUTE DBMS_ADVISOR.CREATE_FILE(DBMS_ADVISOR.GET_TASK_SCRIPT('MYTASK'), -
               'ADVISOR_RESULTS', 'advscript.sql');
```

To save the script to a file, a directory path must be supplied so that the procedure CREATE_FILE knows where to store the script. In addition, read and write privileges must be granted on this directory. The following example shows how to save an advisor script CLOB to a file:

```
-- create a directory and grant permissions to read/write to it
CONNECT SH/SH;
```

```
CREATE DIRECTORY ADVISOR_RESULTS AS '/mydir';
GRANT READ ON DIRECTORY ADVISOR_RESULTS TO PUBLIC;
GRANT WRITE ON DIRECTORY ADVISOR_RESULTS TO PUBLIC;
```

The following is a fragment of a script generated by this procedure. The script also includes PL/SQL calls to gather stats on the recommended access structures and marks the recommendations as IMPLEMENTED at the end.

```
Rem  Access Advisor V10.0.0.0.0 - Beta
Rem
Rem  Username:        SH
Rem  Task:            MYTASK
Rem  Execution date:  15/04/2003 11:35
Rem
set feedback 1
set linesize 80
set trimspool on
set tab off
set pagesize 60
whenever sqlerror CONTINUE

CREATE MATERIALIZED VIEW LOG ON "SH"."PRODUCTS"
    WITH ROWID, SEQUENCE("PROD_ID","PROD_SUBCATEGORY")
    INCLUDING NEW VALUES;
ALTER MATERIALIZED VIEW LOG FORCE ON "SH"."PRODUCTS"
    ADD ROWID, SEQUENCE("PROD_ID","PROD_SUBCATEGORY")
    INCLUDING NEW VALUES;
..
CREATE MATERIALIZED VIEW "SH"."MV$$_00510002"
    REFRESH FAST WITH ROWID
    ENABLE QUERY REWRITE
    AS SELECT SH.CUSTOMERS.CUST_STATE_PROVINCE C1, COUNT(*) M1 FROM
SH.CUSTOMERS WHERE (SH.CUSTOMERS.CUST_STATE_PROVINCE = 'CA') GROUP
BY SH.CUSTOMERS.CUST_STATE_PROVINCE;
BEGIN
  DBMS_STATS.GATHER_TABLE_STATS('"SH"', '"MV$$_00510002"', NULL,
     DBMS_STATS.AUTO_SAMPLE_SIZE);
END;
/
..
CREATE BITMAP INDEX "SH"."MV$$_00510004_IDX$$_00510013"
    ON "SH"."MV$$_00510004" ("C4");
whenever sqlerror EXIT SQL.SQLCODE
BEGIN
  DBMS_ADVISOR.MARK_RECOMMENDATION('"MYTASK"',1,'IMPLEMENTED');
```

```
  DBMS_ADVISOR.MARK_RECOMMENDATION('"MYTASK"',2,'IMPLEMENTED');
  DBMS_ADVISOR.MARK_RECOMMENDATION('"MYTASK"',3,'IMPLEMENTED');
  DBMS_ADVISOR.MARK_RECOMMENDATION('"MYTASK"',4,'IMPLEMENTED');
END;
/
```

> **See Also:** *Oracle Database SQL Reference* for CREATE DIRECTORY syntax and *PL/SQL Packages and Types Reference* for detailed information about the GET_TASK_SCRIPT procedure

When Recommendations are No Longer Required

The RESET_TASK procedure resets a task to its initial starting point. This has the effect of removing all recommendations, and intermediate data from the task. The actual task status is set to INITIAL. The syntax is as follows:

```
DBMS_ADVISOR.RESET_TASK (task_name     IN VARCHAR2);
```

The following illustrates an example of using this procedure:

```
EXECUTE DBMS_ADVISOR.RESET_TASK('MYTASK');
```

See *PL/SQL Packages and Types Reference* for more information regarding the RESET_TASK procedure and its parameters.

Performing a Quick Tune

If you only want to tune a single SQL statement, the QUICK_TUNE procedure accepts as its input a task_name and a SQL statement. It will then create a task and workload and execute that task. There is no difference in the results from using QUICK_TUNE. They are exactly the same as those from using EXECUTE_TASK, but this approach is easier to use when there is only a single SQL statement to be tuned. The syntax is as follows:

```
DBMS_ADVISOR.QUICK_TUNE (
   advisor_name           IN VARCHAR2,
   task_name              IN VARCHAR2,
   attr1                  IN CLOB,
   attr2                  IN VARCHAR2 := NULL,
   attr3                  IN NUMBER := NULL,
   task_or_template       IN VARCHAR2 := NULL);
```

The following example shows how to quick tune a single SQL statement:

```
VARIABLE task_name VARCHAR2(255);
```

```
VARIABLE sql_stmt VARCHAR2(4000);
EXECUTE :sql_stmt := 'SELECT COUNT(*) FROM customers
                      WHERE cust_state_province=''CA''';
EXECUTE :task_name  := 'MY_QUICKTUNE_TASK';
EXECUTE DBMS_ADVISOR.QUICK_TUNE(DBMS_ADVISOR.SQLACCESS_ADVISOR, -
              :task_name, :sql_stmt);
```

See *PL/SQL Packages and Types Reference* for more information regarding the QUICK_TUNE procedure and its parameters.

Managing Tasks

Every time recommendations are generated, tasks are created and, unless some maintenance is performed on these tasks, they will grow over time and will occupy storage space. There may be tasks that you want to keep and prevent accidental deletion. Therefore, there are several management operations that can be performed on tasks:

- Updating Task Attributes
- Deleting Tasks
- Setting DAYS_TO_EXPIRE

Updating Task Attributes

Using the UPDATE_TASK_ATTRIBUTES procedure, you can:

- Change the name of a task.
- Give a task a description.
- Set the task to be read only so it cannot be changed.
- Make the task a template upon which other tasks can be defined.
- Changes various attributes of a task or a task template.

The syntax is as follows:

```
DBMS_ADVISOR.UPDATE_TASK_ATTRIBUTES (
   task_name          IN VARCHAR2
   new_name           IN VARCHAR2 := NULL,
   description        IN VARCHAR2 := NULL,
   read_only          IN VARCHAR2 := NULL,
   is_template        IN VARCHAR2 := NULL,
   source             IN VARCHAR2 := NULL);
```

The following example updates the name of an task MYTASK to TUNING1:

```
EXECUTE DBMS_ADVISOR.UPDATE_TASK_ATTRIBUTES('MYTASK', 'TUNING1');
```

The following example marks the task TUNING1 to read only

```
EXECUTE DBMS_ADVISOR.UPDATE_TASK_ATTRIBUTES('TUNING1', read_only => 'TRUE');
```

The following example marks the task MYTASK as a template.

```
EXECUTE DBMS_ADVISOR.UPDATE_TASK_ATTRIBUTES('TUNING1', is_template=>'TRUE');
```

See *PL/SQL Packages and Types Reference* for more information regarding the UPDATE_TASK_ATTRIBUTES procedure and its parameters.

Deleting Tasks

The DELETE_TASK procedure deletes existing Advisor tasks from the repository. The syntax is as follows:

```
DBMS_ADVISOR.DELETE_TASK (task_name  IN VARCHAR2);
```

The following illustrates an example of using this procedure:

```
EXECUTE DBMS_ADVISOR.DELETE_TASK('MYTASK');
```

See *PL/SQL Packages and Types Reference* for more information regarding the DELETE_TASK procedure and its parameters.

Setting DAYS_TO_EXPIRE

When a task or workload object is created, the parameter DAYS_TO_EXPIRE is set to 30. The value indicates the number of days until the task or object will automatically be deleted by the system. If you wish to save a task or workload indefinitely, the DAYS_TO_EXPIRE parameter should be set to ADVISOR_UNLIMITED.

Using SQLAccess Advisor Constants

You can use the constants shown in Table 17–4 with the SQLAccess Advisor.

Table 17–4 SQLAccess Advisor Constants

Constant	Description
ADVISOR_ALL	A value that is used to indicate all possible values. For string parameters, this value is equivalent to the wildcard % character.
ADVISOR_CURRENT	Indicates the current time or active set of elements. Typically, this is used in time parameters.
ADVISOR_DEFAULT	Indicates the default value. Typically used when setting task or workload parameters.
ADVISOR_UNLIMITED	A value that represents an unlimited numeric value.
ADVISOR_UNUSED	A value that represents an unused entity. When a parameter is set to ADVISOR_UNUSED, it will have no effect on the current operation. This is typically used for setting a parameter as unused for its dependent operations.
SQLACCESS_GENERAL	Specifies the name of a default SQLAccess general-purpose task template. This template will set the DML_VOLATILITY task parameter to TRUE and EXECUTION_TYPE to FULL.
SQLACCESS_OLTP	Specifies the name of a default SQLAccess OLTP task template. This template will set the DML_VOLATILITY task parameter to TRUE and EXECUTION_TYPE to INDEX ONLY.
SQLACCESS_WAREHOUSE	Specifies the name of a default SQLAccess warehouse task template. This template will set the DML_VOLATILITY task parameter to FALSE and EXECUTION_TYPE to FULL.
SQLACCESS_ADVISOR	Contains the formal name of the SQLAccess Advisor. It can be used when procedures require the Advisor name as an argument.

Examples of Using the SQLAccess Advisor

This section illustrates some typical scenarios for using the SQLAccess Advisor. Oracle Database provides a script that contains this chapter's examples, aadvdemo.sql.

Recommendations From a User-Defined Workload

The following example imports workload from a user-defined table, SH.USER_WORKLOAD. It then creates a task called MYTASK, sets the storage budget to 100 MB and runs the task. The recommendations are printed out using a PL/SQL procedure. Finally, it generates a script, which can be used to implement the recommendations.

Step 1 Prepare the USER_WORKLOAD table

The USER_WORKLOAD table is loaded with SQL statements as follows:

```
CONNECT SH/SH;
-- aggregation with selection
INSERT INTO user_workload (username, module, action, priority, sql_text)
VALUES ('SH', 'Example1', 'Action', 2,
'SELECT   t.week_ending_day, p.prod_subcategory,
```

```
          SUM(s.amount_sold) AS dollars, s.channel_id, s.promo_id
 FROM sales s, times t, products p WHERE s.time_id = t.time_id
 AND s.prod_id = p.prod_id AND s.prod_id > 10 AND s.prod_id < 50
 GROUP BY t.week_ending_day, p.prod_subcategory,
          s.channel_id, s.promo_id')
/

-- aggregation with selection
INSERT INTO user_workload (username, module, action, priority, sql_text)
VALUES ('SH', 'Example1', 'Action', 2,
 'SELECT   t.calendar_month_desc, SUM(s.amount_sold) AS dollars
  FROM     sales s , times t
  WHERE    s.time_id = t.time_id
  AND    s.time_id between TO_DATE(''01-JAN-2000'', ''DD-MON-YYYY'')
                       AND TO_DATE(''01-JUL-2000'', ''DD-MON-YYYY'')
GROUP BY t.calendar_month_desc')
/

--Load all SQL queries.
INSERT INTO user_workload (username, module, action, priority, sql_text)
VALUES ('SH', 'Example1', 'Action', 2,
'SELECT ch.channel_class, c.cust_city, t.calendar_quarter_desc,
   SUM(s.amount_sold) sales_amount
FROM sales s, times t, customers c, channels ch
WHERE s.time_id = t.time_id AND s.cust_id = c.cust_id
AND s.channel_id = ch.channel_id AND c.cust_state_province = ''CA''
AND   ch.channel_desc IN (''Internet'',''Catalog'')
AND   t.calendar_quarter_desc IN (''1999-Q1'',''1999-Q2'')
GROUP BY ch.channel_class, c.cust_city, t.calendar_quarter_desc')
/

-- order by
INSERT INTO user_workload (username, module, action, priority, sql_text)
VALUES ('SH', 'Example1', 'Action', 2,
  'SELECT c.country_id, c.cust_city, c.cust_last_name
FROM customers c WHERE c.country_id IN (52790, 52789)
ORDER BY c.country_id, c.cust_city, c.cust_last_name')
/
COMMIT;

CONNECT SH/SH;
set serveroutput on;

VARIABLE task_id NUMBER;
VARIABLE task_name VARCHAR2(255);
```

```
VARIABLE workload_name VARCHAR2(255);
VARIABLE saved_stmts NUMBER;
VARIABLE failed_stmts NUMBER;
```

Step 2 Create a workload named MYWORKLOAD

```
EXECUTE :workload_name := 'MYWORKLOAD';
EXECUTE DBMS_ADVISOR.CREATE_SQLWKLD(:workload_name);
```

Step 3 Load the workload from user-defined table SH.USER_WORKLOAD

```
EXECUTE DBMS_ADVISOR.IMPORT_SQLWKLD_USER (:workload_name, 'APPEND', 'SH', -
   'USER_WORKLOAD', :saved_stmts, :failed_stmts);
PRINT :saved_stmts;
PRINT :failed_stmts;
```

Step 4 Create a task named MYTASK

```
EXECUTE :task_name := 'MYTASK';
EXECUTE DBMS_ADVISOR.CREATE_TASK('SQL Access Advisor', :task_id, :task_name);
```

Step 5 Set task parameters

```
EXECUTE DBMS_ADVISOR.SET_TASK_PARAMETER(:task_name, 'STORAGE_CHANGE', 100);
EXECUTE DBMS_ADVISOR.SET_TASK_PARAMETER ( -
   :task_name, 'EXECUTION_TYPE', 'INDEX_ONLY');
```

Step 6 Create a link between workload and task

```
EXECUTE DBMS_ADVISOR.ADD_SQLWKLD_REF(:task_name, :workload_name);
```

Step 7 Execute the task

```
EXECUTE DBMS_ADVISOR.EXECUTE_TASK(:task_name);
```

Step 8 View the recommendations

```
-- See the number of recommendations and the status of the task.
SELECT rec_id, rank, benefit
FROM user_advisor_recommendations WHERE task_name = :task_name;
```

See "Viewing the Recommendations" on page 17-26 or "Generating SQL Scripts" on page 17-34 for further details.

```
-- See recommendation for each query.
SELECT sql_id, rec_id, precost, postcost,
      (precost-postcost)*100/precost AS percent_benefit
FROM user_advisor_sqla_wk_stmts
```

```
WHERE task_name = :task_name AND workload_name = :workload_name;

-- See the actions for each recommendations.
SELECT rec_id, action_id, SUBSTR(command,1,30) AS command
FROM user_advisor_actions
WHERE task_name = :task_name
ORDER BY rec_id, action_id;

-- See what the actions are using sample procedure.
SET SERVEROUTPUT ON SIZE 99999
EXECUTE show_recm(:task_name);
```

Step 9 Generate a script to Implement the recommendations

```
EXECUTE DBMS_ADVISOR.CREATE_FILE(DBMS_ADVISOR.GET_TASK_SCRIPT(:task_name),-
                                 'ADVISOR_RESULTS', 'Example1_script.sql');
```

Generate Recommendations Using a Task Template

The following example creates a template and then uses it to create a task. It then uses this task to generate recommendations from a user-defined table, similar to "Recommendations From a User-Defined Workload" on page 17-39.

```
CONNECT SH/SH;
VARIABLE template_id NUMBER;
VARIABLE template_name VARCHAR2(255);
```

Step 1 Create a template called MY_TEMPLATE

```
EXECUTE :template_name := 'MY_TEMPLATE';
EXECUTE DBMS_ADVISOR.CREATE_TASK ( -
   'SQL Access Advisor',:template_id, :template_name, is_template=>'TRUE');
```

Step 2 Set template parameters

Set naming conventions for recommended indexes/materialized views.

```
EXECUTE DBMS_ADVISOR.SET_TASK_PARAMETER ( -
   :template_name,  'INDEX_NAME_TEMPLATE', 'SH_IDX$$_<SEQ>');
EXECUTE DBMS_ADVISOR.SET_TASK_PARAMETER ( -
   :template_name, 'MVIEW_NAME_TEMPLATE', 'SH_MV$$_<SEQ>');

--Set default owners for recommended indexes/materialized views.
EXECUTE DBMS_ADVISOR.SET_TASK_PARAMETER ( -
   :template_name, 'DEF_INDEX_OWNER', 'SH');
EXECUTE DBMS_ADVISOR.SET_TASK_PARAMETER ( -
   :template_name, 'DEF_MVIEW_OWNER', 'SH');
```

```
--Set default tablespace for recommended indexes/materialized views.
EXECUTE DBMS_ADVISOR.SET_TASK_PARAMETER ( -
   :template_name, 'DEF_INDEX_TABLESPACE', 'SH_INDEXES');
EXECUTE DBMS_ADVISOR.SET_TASK_PARAMETER ( -
   :template_name, 'DEF_MVIEW_TABLESPACE', 'SH_MVIEWS');
```

Step 3 Create a task using the template

```
VARIABLE task_id NUMBER;
VARIABLE task_name VARCHAR2(255);
EXECUTE :task_name := 'MYTASK';
EXECUTE DBMS_ADVISOR.CREATE_TASK ( -
   'SQL Access Advisor', :task_id, :task_name, template => 'MY_TEMPLATE');

--See the parameter settings for task
SELECT parameter_name, parameter_value
FROM user_advisor_parameters
WHERE task_name = :task_name AND (parameter_name LIKE '%MVIEW%'
   OR parameter_name LIKE '%INDEX%');
```

Step 4 Create a workload named MYWORKLOAD

```
VARIABLE workload_name VARCHAR2(255);
VARIABLE saved_stmts NUMBER;
VARIABLE failed_stmts NUMBER;
EXECUTE :workload_name := 'MYWORKLOAD';
EXECUTE DBMS_ADVISOR.CREATE_SQLWKLD(:workload_name);
```

Step 5 Load the workload from user-defined table SH.USER_WORKLOAD

```
EXECUTE DBMS_ADVISOR.IMPORT_SQLWKLD_USER ( -
   :workload_name, 'APPEND', 'SH', 'USER_WORKLOAD', :saved_stmts,:failed_stmts);
```

Step 6 Create a link between the workload and the task

```
EXECUTE DBMS_ADVISOR.ADD_SQLWKLD_REF(:task_name, :workload_name);
```

Step 7 Execute the task

```
EXECUTE DBMS_ADVISOR.EXECUTE_TASK(:task_name);
```

Step 8 Generate a script

```
EXECUTE DBMS_ADVISOR.CREATE_FILE(DBMS_ADVISOR.GET_TASK_SCRIPT(:task_name),-
                                 'ADVISOR_RESULTS', 'Example2_script.sql');
```

Filter a Workload from the SQL Cache

The following example illustrates collection of a workload from a SQL cache. We first load the cache with a bunch of SQL statements. We then setup some filters to pick only a subset of those statements and import them into a SQLAccess Advisor workload. The workload is then used to generate recommendations.

Step 1 Loading the SQL cache

The following statements are executed so they will be in the SQL cache:

```
CONNECT / AS SYSDBA

--Clear any prior contents of the cache.
ALTER SYSTEM FLUSH SHARED_POOL;
CONNECT SH/SH;

SELECT   t.calendar_month_desc, SUM(s.amount_sold) AS dollars
FROM     sales s, times t WHERE s.time_id = t.time_id
AND  s.time_id between TO_DATE('01-JAN-2000', 'DD-MON-YYYY')
                      AND TO_DATE('01-JUL-2000', 'DD-MON-YYYY')
GROUP BY t.calendar_month_desc;

-- Order by
SELECT c.country_id, c.cust_city, c.cust_last_name
FROM customers c WHERE c.country_id IN ('52790', '52789')
ORDER BY c.country_id, c.cust_city, c.cust_last_name;

-- Queries to illustrate filtering
CONNECT scott/tiger;
SELECT e.ename, d.dname
FROM emp e, dept d WHERE e.deptno = d.deptno;

SELECT COUNT(*) FROM dept;

CONNECT sh/sh
VARIABLE task_id NUMBER;
VARIABLE task_name VARCHAR2(255);
VARIABLE workload_name VARCHAR2(255);
VARIABLE saved_stmts NUMBER;
VARIABLE failed_stmts NUMBER;
```

Step 2 Create a workload named MY_CACHE_WORKLOAD

```
EXECUTE :workload_name := 'MY_CACHE_WORKLOAD';
EXECUTE DBMS_ADVISOR.CREATE_SQLWKLD(:workload_name);
```

Step 3 Set up filters

Load only SQL statements containing SH tables

```
EXECUTE DBMS_ADVISOR.SET_SQLWKLD_PARAMETER ( -
   :workload_name, 'USERNAME_LIST', 'SH');
```

Step 4 Load the workload from SQL Cache

```
EXECUTE DBMS_ADVISOR.IMPORT_SQLWKLD_SQLCACHE ( -
   :workload_name, 'APPEND', 2, :saved_stmts, :failed_stmts);
PRINT :saved_stmts;
PRINT :failed_stmts;

--See the workload statements in catalog views
SELECT num_select_stmt, create_date
FROM user_advisor_sqlw_sum
WHERE workload_name = :workload_name;

SELECT sql_id, username, optimizer_cost, SUBSTR(sql_text, 1, 30)
FROM user_advisor_sqlw_stmts
WHERE workload_name = :workload_name
ORDER BY sql_id;
```

Step 5 Add a single statement to the workload

```
EXECUTE DBMS_ADVISOR.ADD_SQLWKLD_STATEMENT (:workload_name, username => 'SH', -
    priority => 1, executions => 10,  sql_text =>  -
   'select count(*) from customers where cust_state_province=''CA''');

SELECT num_select_stmt, create_date
FROM user_advisor_sqlw_sum
WHERE workload_name = :workload_name;
```

Step 6 Update a statement in the workload

```
VARIABLE updated_stmts NUMBER;
EXECUTE DBMS_ADVISOR.UPDATE_SQLWKLD_STATEMENT ( -
   :workload_name, 'executions < 10', :updated_stmts, priority => 3);

PRINT :updated_stmts;

--See that the change has been made.
SELECT sql_id, username, executions, priority
FROM user_advisor_sqlw_stmts
WHERE workload_name = :workload_name
ORDER BY sql_id;
```

Step 7 Create a task named MYTASK

```
EXECUTE :task_name := 'MYTASK';
EXECUTE DBMS_ADVISOR.CREATE_TASK ('SQL Access Advisor', :task_id, :task_name);
```

Step 8 Create a link between a workload and a task

```
EXECUTE DBMS_ADVISOR.ADD_SQLWKLD_REF(:task_name, :workload_name);
```

Step 9 Execute the task

```
EXECUTE DBMS_ADVISOR.EXECUTE_TASK(:task_name);
```

Step 10 Generate a script

```
EXECUTE DBMS_ADVISOR.CREATE_FILE(DBMS_ADVISOR.GET_TASK_SCRIPT(:task_name),-
                                 'ADVISOR_RESULTS', 'Example3_script.sql');
```

Evaluate Current Usage of Indexes and Materialized Views

This example illustrates how SQLAccess Advisor may be used to evaluate the utilization of existing indexes and materialized views. We assume the workload is loaded into USER_WORKLOAD table as in "Recommendations From a User-Defined Workload" on page 17-39. The indexes and materialized views that are being currently used (by the given workload) will appear as RETAIN actions in the SQLAccess Advisor recommendations.

```
CONNECT SH/SH;
VARIABLE task_id NUMBER;
VARIABLE task_name VARCHAR2(255);
VARIABLE workload_name VARCHAR2(255);
VARIABLE saved_stmts NUMBER;
VARIABLE failed_stmts NUMBER;
```

Step 1 Create a workload named WORKLOAD

```
EXECUTE :workload_name := 'MYWORKLOAD';
EXECUTE DBMS_ADVISOR.CREATE_SQLWKLD(:workload_name);
```

Step 2 Load the workload from user-defined table SH.USER_WORKLOAD

```
EXECUTE DBMS_ADVISOR.IMPORT_SQLWKLD_USER ( -
   :workload_name, 'APPEND', 'SH','USER_WORKLOAD', :saved_stmts, :failed_stmts);

PRINT :saved_stmts;
PRINT :failed_stmts;
```

Step 3 Create a task named MY_EVAL_TASK

```
EXECUTE :task_name := 'MY_EVAL_TASK';
EXECUTE DBMS_ADVISOR.CREATE_TASK ('SQL Access Advisor', :task_id, :task_name);
```

Step 4 Create a link between workload and task

```
EXECUTE DBMS_ADVISOR.ADD_SQLWKLD_REF(:task_name, :workload_name);
```

Step 5 Set task parameters to indicate EVALUATION ONLY task

```
EXECUTE DBMS_ADVISOR.SET_TASK_PARAMETER (:task_name, 'EVALUATION_ONLY', 'TRUE');
```

Step 6 Execute the task

```
EXECUTE DBMS_ADVISOR.EXECUTE_TASK(:task_name);
```

Step 7 View evaluation results

```
--See the number of recommendations and the status of the task.
SELECT rec_id, rank, benefit
FROM user_advisor_recommendations WHERE task_name = :task_name;

--See the actions for each recommendation.
SELECT rec_id, action_id, SUBSTR(command, 1, 30) AS command
FROM user_advisor_actions WHERE task_name = :task_name
ORDER BY rec_id, action_id;
```

Tuning Materialized Views for Fast Refresh and Query Rewrite

Several DBMS_MVIEW procedures help you with materialized view fast refresh and query rewrite. The EXPLAIN_MVIEW procedure can tell you whether a materialized view is fast refreshable or eligible for general query rewrite and EXPLAIN_REWRITE will tell you whether query rewrite will occur. However, neither tells you how to achieve fast refresh or query rewrite.

To further facilitate the use of materialized views, the TUNE_MVIEW procedure shows you how to optimize your CREATE MATERIALIZED VIEW statement and to meet other requirements such as materialized view log and rewrite equivalence relationship for fast refresh and general query rewrite. TUNE_MVIEW analyzes and processes the input statement and generates two sets of output results: one for the materialized view implementation and the other for undoing the create materialized view operations. The two sets of output results can be accessed through Oracle views or be stored in external script files created by the SQLAccess Advisor. These external script files are ready to execute to implement the materialized view.

With the TUNE_MVIEW procedure, you no longer require a detailed understanding of materialized views to create a materialized view in an application because the materialized view and its required components (such as materialized view log) will be created correctly through the procedure.

See *PL/SQL Packages and Types Reference* for detailed information about the TUNE_MVIEW procedure.

DBMS_ADVISOR.TUNE_MVIEW Procedure

This section discusses the following information:

- TUNE_MVIEW Syntax and Operations
- Accessing TUNE_MVIEW Output Results
- USER_TUNE_MVIEW and DBA_TUNE_MVIEW Views

TUNE_MVIEW Syntax and Operations

The syntax for TUNE_MVIEW is as follows:

```
DBMS_ADVISOR.TUNE_MVIEW (
  task_name IN OUT VARCHAR2, mv_create_stmt IN [CLOB | VARCHAR2])
```

The TUNE_MVIEW procedure takes two input parameters: task_name and mv_create_stmt. task_name is a user-provided task identifier used to access the output results. mv_create_stmt is a complete CREATE MATERIALIZED VIEW statement that is to be tuned. If the input CREATE MATERIALIZED VIEW statement does not have the clauses of REFRESH FAST or ENABLE QUERY REWRITE, or both, TUNE_MVIEW will use the default clauses REFRESH FORCE and DISABLE QUERY REWRITE to tune the statement to be fast refreshable if possible or only complete refreshable otherwise.

The TUNE_MVIEW procedure handles a broad range of CREATE MATERIALIZED VIEW statements that can have arbitrary defining queries in them. The defining query could be a simple SELECT statement or a complex query with set operators or inline views. When the defining query of the materialized view contains the clause REFRESH FAST, TUNE_MVIEW analyzes the query and checks to see if it is fast refreshable. If it is already fast refreshable, the procedure will return a message saying "the materialized view is already optimal and cannot be further tuned". Otherwise, the TUNE_MVIEW procedure will start the tuning work on the given statement.

The TUNE_MVIEW procedure can generate the output statements that correct the defining query by adding extra columns such as required aggregate columns or fix the materialized view logs to achieve the FAST REFRESH goal. In the case of a complex defining query, the TUNE_MVIEW procedure decomposes the query and generates two or more fast refreshable materialized views or will restate the materialized view in a way to fulfill fast refresh requirements as much as possible. The TUNE_MVIEW procedure supports defining queries with the following complex query constructs:

- Set operators (UNION, UNION ALL, MINUS, and INTERSECT)
- COUNT DISTINCT
- SELECT DISTINCT
- Inline views

When the ENABLE QUERY REWRITE clause is specified, TUNE_MVIEW will also fix the statement using a process similar to REFRESH FAST, that will redefine the materialized view so that as many of the advanced forms of query rewrite are possible.

The TUNE_MVIEW procedure generates two sets of output results as executable statements. One set of the output (IMPLEMENTATION) is for implementing materialized views and required components such as materialized view logs or rewrite equivalences to achieve fast refreshability and query rewritablity as much as possible. The other set of the output (UNDO) is for dropping the materialized views and the rewrite equivalences in case you decide they are not required.

The output statements for the IMPLEMENTATION process include:

- CREATE MATERIALIZED VIEW LOG statements: creates any missing materialized view logs required for fast refresh.
- ALTER MATERIALIZED VIEW LOG FORCE statements: fixes any materialized view log related requirements such as missing filter columns, sequence, and so on, required for fast refresh.
- One or more CREATE MATERIALIZED VIEW statements: In case of one output statement, the original defining query is directly restated and transformed. Simple query transformation could be just adding required columns. For example, add rowid column for materialized join view and add aggregate column for materialized aggregate view. In the case of decomposition, multiple CREATE MATERIALIZED VIEW statements are generated and form a nested materialized view hierarchy in which one or more submaterialized views are referenced by a new top-level materialized view modified from the original

statement. This is to achieve fast refresh and query rewrite as much as possible. Submaterialized views are often fast refreshable.

- BUILD_SAFE_REWRITE_EQUIVALENCE statement: enables the rewrite of top-level materialized view using submaterialized views. It is required to enable query rewrite when a composition occurs.

Note that the decomposition result implies no sharing of submaterialized views. That is, in the case of decomposition, the TUNE_MVIEW output will always contain new submaterialized view and it will not reference existing materialized views.

The output statements for the UNDO process include:

- DROP MATERIALIZED VIEW statements to reverse the materialized view creations (including submaterialized views) in the IMPLEMENTATION process.
- DROP_REWRITE_EQUIVALENCE statement to remove the rewrite equivalence relationship built in the IMPLEMENTATION process if needed.

Note that the UNDO process does not include statement to drop materialized view logs. This is because materialized view logs can be shared by many different materialized views, some of which may reside on remote Oracle instances.

Accessing TUNE_MVIEW Output Results

There are two ways to access TUNE_MVIEW output results:

- Script generation using DBMS_ADVISOR.GET_TASK_SCRIPT function and DBMS_ADVISOR.CREATE_FILE procedure.
- Use USER_TUNE_MVIEW or DBA_TUNE_MVIEW views.

USER_TUNE_MVIEW and DBA_TUNE_MVIEW Views

After executing TUNE_MVIEW, the results are output into the SQLAccess Advisor repository tables and are accessible through the Oracle views, USER_TUNE_MVIEW and DBA_TUNE_MVIEW. See *Oracle Database Reference* for further details.

Script Generation DBMS_ADVISOR Function and Procedure

It is straightforward to have the SQLAccess Advisor generate scripts using the function DBMS_ADVISOR.GET_TASK_SCRIPT. The following is a simple example. First, a directory must be defined which is where the results will be stored:

```
CREATE DIRECTORY TUNE_RESULTS AS  '/tmp/script_dir';
GRANT READ, WRITE ON DIRECTORY TUNE_RESULTS TO PUBLIC;
```

Now generate both the implementation and undo scripts and place them in /tmp/script_dir/mv_create.sql and /tmp/script_dir/mv_undo.sql, respectively.

```
EXECUTE DBMS_ADVISOR.CREATE_FILE(DBMS_ADVISOR.GET_TASK_SCRIPT(:task_name),-
      'TUNE_RESULTS', 'mv_create.sql');
EXECUTE DBMS_ADVISOR.CREATE_FILE(DBMS_ADVISOR.GET_TASK_SCRIPT(:task_name, -
      'UNDO'), 'TUNE_RESULTS', 'mv_undo.sql');
```

Now let us review some examples using TUNE_MVIEW.

Example 17–3 Optimizing the Defining Query for Fast Refresh

This example shows how TUNE_MVIEW changes the defining query to be fast refreshable. A MATERIALIZED VIEW CREATE statement is defined in variable create_mv_ddl. This create statement has the REFRESH FAST clause specified. Its defining query contains a single query block in which an aggregate column, SUM(s.amount_sold), does not have the required aggregate columns to support fast refresh. If you execute the TUNE_MVIEW statement with this MATERIALIZED VIEW CREATE statement, the output produced will be fast refreshable:

```
VARIABLE task_cust_mv VARCHAR2(30);
VARIABLE create_mv_ddl VARCHAR2(4000);
EXECUTE :task_cust_mv := 'cust_mv';

EXECUTE :create_mv_ddl := ' -
CREATE MATERIALIZED VIEW cust_mv -
REFRESH FAST -
DISABLE QUERY REWRITE AS -
SELECT s.prod_id, s.cust_id, SUM(s.amount_sold) sum_amount -
FROM sales s, customers cs -
WHERE s.cust_id = cs.cust_id -
GROUP BY s.prod_id, s.cust_id';

EXECUTE DBMS_ADVISOR.TUNE_MVIEW(:task_cust_mv, :create_mv_ddl);
```

The projected output of TUNE_MVIEW includes an optimized materialized view defining query as follows:

```
CREATE MATERIALIZED VIEW SH.CUST_MV
REFRESH FAST WITH ROWID
DISABLE QUERY REWRITE AS
SELECT SH.SALES.PROD_ID C1, SH.CUSTOMERS.CUST_ID C2,
          SUM("SH"."SALES"."AMOUNT_SOLD") M1,
          COUNT("SH"."SALES"."AMOUNT_SOLD") M2,
```

```
     COUNT(*) M3
FROM SH.SALES, SH.CUSTOMERS
WHERE SH.CUSTOMERS.CUST_ID = SH.SALES.CUST_ID
GROUP BY SH.SALES.PROD_ID, SH.CUSTOMERS.CUST_ID;
```

The UNDO output is as follows:

```
DROP MATERIALIZED VIEW SH.CUST_MV;
```

The original defining query of `cust_mv` has been modified by adding aggregate columns in order to be fast refreshable.

Example 17–4 Access IMPLEMENTATION Output Through USER_TUNE_MVIEW View

```
SELECT * FROM USER_TUNE_MVIEW
WHERE TASK_NAME= :task_cust_mv AND SCRIPT_TYPE='IMPLEMENTATION';
```

Example 17–5 Save IMPLEMENTATION Output in a Script File

```
CREATE DIRECTORY TUNE_RESULTS AS '/myscript'
GRANT READ, WRITE ON DIRECTORY TUNE_RESULTS TO PUBLIC;

EXECUTE DBMS_ADVISOR.CREATE_FILE(DBMS_ADVISOR.GET_TASK_SCRIPT(:task_cust_mv), -
   'TUNE_RESULTS', 'mv_create.sql');
```

Example 17–6 Enable Query Rewrite by Creating Multiple Materialized Views

This example shows how a materialized view's defining query with set operators can be decomposed into a number of submaterialized views. The input detail tables are assumed to be sales, customers, and countries, and they do not have materialized view logs.

First, you need to execute the TUNE_MVIEW statement with the materialized view CREATE statement defined in the variable `create_mv_ddl`.

```
EXECUTE :task_cust_mv := 'cust_mv2';

EXECUTE :create_mv_ddl := ' -
CREATE MATERIALIZED VIEW cust_mv -
ENABLE QUERY REWRITE AS -
SELECT s.prod_id, s.cust_id, COUNT(*) cnt, SUM(s.amount_sold) sum_amount -
FROM sales s, customers cs, countries cn -
WHERE s.cust_id = cs.cust_id AND cs.country_id = cn.country_id -
AND cn.country_name IN (''USA'',''Canada'') -
GROUP BY s.prod_id, s.cust_id -
UNION -
```

```
SELECT s.prod_id, s.cust_id, COUNT(*) cnt, SUM(s.amount_sold) sum_amount -
FROM sales s, customers cs -
WHERE s.cust_id = cs.cust_id AND s.cust_id IN (1005,1010,1012) -
GROUP BY s.prod_id, s.cust_id';

EXECUTE DBMS_ADVISOR.TUNE_MVIEW(:task_cust_mv, :create_mv_ddl);
```

The materialized view defining query contains a UNION set operator and does not support general query rewrite. In order to support general query rewrite, the MATERIALIZED VIEW defining query will be decomposed.

The projected output for the IMPLEMENTATION statement will be created along with materialized view log statements and two submaterialized views as follows:

```
CREATE MATERIALIZED VIEW LOG ON "SH"."SALES"
WITH ROWID, SEQUENCE("CUST_ID")
    INCLUDING NEW VALUES;

ALTER MATERIALIZED VIEW LOG FORCE ON
    "SH"."CUSTOMERS"
    ADD ROWID, SEQUENCE("CUST_ID")
    INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW LOG ON
    "SH"."SALES"
    WITH ROWID, SEQUENCE("PROD_ID","CUST_ID","AMOUNT_SOLD")
    INCLUDING NEW VALUES;

ALTER MATERIALIZED VIEW LOG FORCE ON
    "SH"."SALES"
    ADD ROWID, SEQUENCE("PROD_ID","CUST_ID","AMOUNT_SOLD")
    INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW LOG ON
    "SH"."COUNTRIES"
    WITH ROWID, SEQUENCE("COUNTRY_ID","COUNTRY_NAME")
    INCLUDING NEW VALUES;

ALTER MATERIALIZED VIEW LOG FORCE ON
    "SH"."COUNTRIES"
    ADD ROWID, SEQUENCE("COUNTRY_ID","COUNTRY_NAME")
    INCLUDING NEW VALUES;

ALTER MATERIALIZED VIEW LOG FORCE ON
    "SH"."CUSTOMERS"
```

```
    ADD ROWID, SEQUENCE("CUST_ID","COUNTRY_ID")
    INCLUDING NEW VALUES;

ALTER MATERIALIZED VIEW LOG FORCE ON
    "SH"."SALES"
    ADD ROWID, SEQUENCE("PROD_ID","CUST_ID","AMOUNT_SOLD")
    INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW SH.CUST_MV$SUB1
    REFRESH FAST WITH ROWID ON COMMIT
    ENABLE QUERY REWRITE
    AS SELECT SH.SALES.PROD_ID C1, SH.CUSTOMERS.CUST_ID C2,
 SUM("SH"."SALES"."AMOUNT_SOLD")
        M1, COUNT("SH"."SALES"."AMOUNT_SOLD") M2, COUNT(*) M3 FROM SH.SALES,
        SH.CUSTOMERS WHERE SH.CUSTOMERS.CUST_ID = SH.SALES.CUST_ID AND
 (SH.SALES.CUST_ID
        IN (1012, 1010, 1005)) GROUP BY SH.SALES.PROD_ID, SH.CUSTOMERS.CUST_ID;

CREATE MATERIALIZED VIEW SH.CUST_MV$SUB2
    REFRESH FAST WITH ROWID ON COMMIT
    ENABLE QUERY REWRITE
    AS SELECT SH.SALES.PROD_ID C1, SH.CUSTOMERS.CUST_ID C2,
 SH.COUNTRIES.COUNTRY_NAME
        C3, SUM("SH"."SALES"."AMOUNT_SOLD") M1, COUNT("SH"."SALES".
"AMOUNT_SOLD")
        M2, COUNT(*) M3 FROM SH.SALES, SH.CUSTOMERS, SH.COUNTRIES WHERE
 SH.CUSTOMERS.CUST_ID
        = SH.SALES.CUST_ID AND SH.COUNTRIES.COUNTRY_ID = SH.CUSTOMERS.COUNTRY_ID
        AND (SH.COUNTRIES.COUNTRY_NAME IN ('USA', 'Canada')) GROUP BY
 SH.SALES.PROD_ID,
        SH.CUSTOMERS.CUST_ID, SH.COUNTRIES.COUNTRY_NAME;

CREATE MATERIALIZED VIEW SH.CUST_MV
    REFRESH FORCE WITH ROWID
    ENABLE QUERY REWRITE
    AS  (SELECT "CUST_MV$SUB2"."C1" "PROD_ID","CUST_MV$SUB2"."C2"
"CUST_ID",SUM("CUST_MV$SUB2"."M3")
        "CNT",SUM("CUST_MV$SUB2"."M1") "SUM_AMOUNT" FROM "SH"."CUST_MV$SUB2"
        "CUST_MV$SUB2" GROUP BY "CUST_MV$SUB2"."C1","CUST_MV$SUB2"."C2")UNION
        (SELECT "CUST_MV$SUB1"."C1" "PROD_ID","CUST_MV$SUB1"."C2"
"CUST_ID",SUM("CUST_MV$SUB1"."M3")
        "CNT",SUM("CUST_MV$SUB1"."M1") "SUM_AMOUNT" FROM "SH"."CUST_MV$SUB1"
        "CUST_MV$SUB1" GROUP BY "CUST_MV$SUB1"."C1","CUST_MV$SUB1"."C2");

BEGIN
```

```
DBMS_ADVANCED_REWRITE.BUILD_SAFE_REWRITE_EQUIVALENCE ('SH.CUST_MV$RWEQ',
  'SELECT s.prod_id, s.cust_id, COUNT(*) cnt,
          SUM(s.amount_sold) sum_amount
   FROM sales s, customers cs, countries cn
   WHERE s.cust_id = cs.cust_id AND cs.country_id = cn.country_id
         AND cn.country_name IN (''USA'',''Canada'')
   GROUP BY s.prod_id, s.cust_id
   UNION
   SELECT s.prod_id, s.cust_id, COUNT(*) cnt,
          SUM(s.amount_sold) sum_amount
   FROM sales s, customers cs
   WHERE s.cust_id = cs.cust_id AND s.cust_id IN (1005,1010,1012)
   GROUP BY s.prod_id, s.cust_id',
  '(SELECT "CUST_MV$SUB2"."C3" "PROD_ID","CUST_MV$SUB2"."C2" "CUST_ID",
           SUM("CUST_MV$SUB2"."M3") "CNT",
           SUM("CUST_MV$SUB2"."M1") "SUM_AMOUNT"
    FROM "SH"."CUST_MV$SUB2" "CUST_MV$SUB2"
    GROUP BY "CUST_MV$SUB2"."C3","CUST_MV$SUB2"."C2")
   UNION
   (SELECT "CUST_MV$SUB1"."C2" "PROD_ID","CUST_MV$SUB1"."C1" "CUST_ID",
           "CUST_MV$SUB1"."M3" "CNT","CUST_MV$SUB1"."M1" "SUM_AMOUNT"
    FROM "SH"."CUST_MV$SUB1" "CUST_MV$SUB1")',-1553577441)
END;
/;
```

The DROP output is as follows:

```
DROP MATERIALIZED VIEW SH.CUST_MV$SUB1
DROP MATERIALIZED VIEW SH.CUST_MV$SUB2
DROP MATERIALIZED VIEW SH.CUST_MV
DBMS_ADVANCED_REWRITE.DROP_REWRITE_EQUIVALENCE('SH.CUST_MV$RWEQ')
```

The original defining query of `cust_mv` has been decomposed into two submaterialized views seen as `cust_mv$SUB1` and `cust_mv$SUB2`. One additional column `count(amount_sold)` has been added in `cust_mv$SUB1` to make that materialized view fast refreshable.

The original defining query of `cust_mv` has been modified to query the two submaterialized views instead where both submaterialized views are fast refreshable and support general query rewrite.

The required materialized view logs are added to enable fast refresh of the submaterialized views. It is noted that for each detail table, two materialized view log statements are generated: one is `CREATE MATERIALIZED VIEW` statement and

the other is ALTER MATERIALIZED VIEW FORCE statement. This is to ensure the CREATE script can be run multiple times.

The BUILD_SAFE_REWRITE_EQUIVALENCE statement is to connect the old defining query to the defining query of the new top-level materialized view. It is to ensure that query rewrite will make use of the new top-level materialized view to answer the query.

Example 17–7 Access IMPLEMENTATION Output Through USER_TUNE_MVIEW View

```
SELECT * FROM USER_TUNE_MVIEW
WHERE TASK_NAME='cust_mv2'
AND SCRIPT_TYPE='IMPLEMENTATION';
```

Example 17–8 Save IMPLEMENTATION Output in a Script File

The following statements save the IMPLEMENTATION output in a script file located at /myscript/mv_create2.sql:

```
CREATE DIRECTORY TUNE_RESULTS AS '/myscript'
GRANT READ, WRITE ON DIRECTRY TUNE_RESULTS TO PUBLIC;
EXECUTE DBMS_ADVISOR.CREATE_FILE(DBMS_ADVISOR.GET_TASK_SCRIPT('cust_mv2'),-
   'TUNE_RESULTS', 'mv_create2.sql');
```

Fast Refreshable with Optimized Sub-Materialized View

The following example is to show some optimization in TUNE_MVIEW. In the example, the materialized view's defining query with set operators is transformed into one sub-materialized view and one top-level materialized view. The sub-SELECT queries in the original defining query are of similar shape and their predicate expressions are combined.

Assume that detail tables sales and customers do not have materialized view logs. You execute the following statement with a given CREATE MATERIALIZED VIEW statement.

Example 17–9 Optimized Sub-Materialized View for Fast Refresh

```
EXECUTE :task_cust_mv := 'cust_mv3';
EXECUTE :create_mv_ddl := '-
CREATE MATERIALIZED VIEW cust_mv -
REFRESH FAST ON DEMAND -
ENABLE QUERY REWRITE AS -
SELECT s.prod_id, s.cust_id, COUNT(*) cnt, SUM(s.amount_sold) sum_amount -
FROM sales s, customers cs -
WHERE s.cust_id = cs.cust_id AND s.cust_id IN (2005,1020) -
```

```
GROUP BY s.prod_id, s.cust_id UNION -
SELECT s.prod_id, s.cust_id, COUNT(*) cnt, SUM(s.amount_sold) sum_amount -
FROM sales s, customers cs -
WHERE s.cust_id = cs.cust_id AND s.cust_id IN (1005,1010,1012) -
GROUP BY s.prod_id, s.cust_id';

EXECUTE DBMS_ADVISOR.TUNE_MVIEW(:task_cust_mv, :create_mv_ddl);
```

The materialized view defining query contains a UNION set-operator so that the materialized view itself is not fast-refreshable. However, two subselect queries in the materialized view defining query can be combined as one single query.

The projected output for CREATE statement will be created with an optimized submaterialized view combining the two subselect queries and the submaterialized view is referenced by a new top-level materialized view as follows:

```
CREATE MATERIALIZED VIEW LOG ON "SH"."SALES"
  WITH ROWID, SEQUENCE ("PROD_ID","CUST_ID","AMOUNT_SOLD")
  INCLUDING NEW VALUES
ALTER MATERIALIZED VIEW LOG FORCE ON "SH"."SALES"
  ADD ROWID, SEQUENCE ("PROD_ID","CUST_ID","AMOUNT_SOLD")
  INCLUDING NEW VALUES
CREATE MATERIALIZED VIEW LOG ON "SH"."CUSTOMERS"
  WITH ROWID, SEQUENCE ("CUST_ID")  INCLUDING NEW VALUES
ALTER MATERIALIZED VIEW LOG FORCE ON "SH"."CUSTOMERS"
  ADD ROWID, SEQUENCE ("CUST_ID")  INCLUDING NEW VALUES
CREATE MATERIALIZED VIEW SH.CUST_MV$SUB1
  REFRESH FAST WITH ROWID
  ENABLE QUERY REWRITE AS
  SELECT SH.SALES.CUST_ID C1, SH.SALES.PROD_ID C2,
    SUM("SH"."SALES"."AMOUNT_SOLD") M1,
    COUNT("SH"."SALES"."AMOUNT_SOLD")M2, COUNT(*) M3
    FROM SH.CUSTOMERS, SH.SALES
    WHERE SH.SALES.CUST_ID = SH.CUSTOMERS.CUST_ID AND
    (SH.SALES.CUST_ID IN (2005, 1020, 1012, 1010, 1005))
    GROUP BY SH.SALES.CUST_ID, SH.SALES.PROD_ID
CREATE MATERIALIZED VIEW SH.CUST_MV
  REFRESH FORCE WITH ROWID ENABLE QUERY REWRITE AS
  (SELECT "CUST_MV$SUB1"."C2" "PROD_ID","CUST_MV$SUB1"."C1" "CUST_ID",
    "CUST_MV$SUB1"."M3" "CNT","CUST_MV$SUB1"."M1" "SUM_AMOUNT"
    FROM "SH"."CUST_MV$SUB1" "CUST_MV$SUB1"
    WHERE "CUST_MV$SUB1"."C1"=2005 OR "CUST_MV$SUB1"."C1"=1020)
    UNION
    (SELECT "CUST_MV$SUB1"."C2" "PROD_ID","CUST_MV$SUB1"."C1" "CUST_ID",
      "CUST_MV$SUB1"."M3" "CNT","CUST_MV$SUB1"."M1" "SUM_AMOUNT"
```

```
      FROM "SH"."CUST_MV$SUB1" "CUST_MV$SUB1"
      WHERE "CUST_MV$SUB1"."C1"=1012 OR "CUST_MV$SUB1"."C1"=1010 OR
            "CUST_MV$SUB1"."C1"=1005)

DBMS_ADVANCED_REWRITE.BUILD_SAFE_REWRITE_EQUIVALENCE ('SH.CUST_MV$RWEQ',
      'SELECT s.prod_id, s.cust_id, COUNT(*) cnt,
       SUM(s.amount_sold) sum_amount
       FROM sales s, customers cs
       WHERE s.cust_id = cs.cust_id AND s.cust_id in (2005,1020)
       GROUP BY s.prod_id, s.cust_id UNION
       SELECT s.prod_id, s.cust_id, COUNT(*) cnt,
       SUM(s.amount_sold) sum_amount
       FROM sales s, customers cs
       WHERE s.cust_id = cs.cust_id AND s.cust_id IN (1005,1010,1012)
       GROUP BY s.prod_id, s.cust_id',
      '(SELECT "CUST_MV$SUB1"."C2" "PROD_ID",
        "CUST_MV$SUB1"."C1" "CUST_ID",
        "CUST_MV$SUB1"."M3" "CNT","CUST_MV$SUB1"."M1" "SUM_AMOUNT"
        FROM "SH"."CUST_MV$SUB1" "CUST_MV$SUB1"
        WHERE "CUST_MV$SUB1"."C1"=2005OR "CUST_MV$SUB1"."C1"=1020)
       UNION
       (SELECT "CUST_MV$SUB1"."C2" "PROD_ID",
        "CUST_MV$SUB1"."C1" "CUST_ID",
        "CUST_MV$SUB1"."M3" "CNT","CUST_MV$SUB1"."M1" "SUM_AMOUNT"
        FROM "SH"."CUST_MV$SUB1" "CUST_MV$SUB1"
        WHERE "CUST_MV$SUB1"."C1"=1012 OR "CUST_MV$SUB1"."C1"=1010 OR
              "CUST_MV$SUB1"."C1"=1005)',
      1811223110)
```

The DROP output is as follows:

```
DROP MATERIALIZED VIEW SH.CUST_MV$SUB1
DROP MATERIALIZED VIEW SH.CUST_MV
DBMS_ADVANCED_REWRITE.DROP_REWRITE_EQUIVALENCE('SH.CUST_MV$RWEQ')
```

The original defining query of cust_mv has been optimized by combining the predicate of the two subselect queries in the sub-materialized view CUST_MV$SUB1. The required materialized view logs are also added to enable fast refresh of the submaterialized views.

Example 17–10 Access IMPLEMENTATION Output Through USER_TUNE_MVIEW View

The following query accesses the IMPLEMENTATION output through USER_TUNE_MVIEW:

```
SELECT * FROM USER_TUNE_MVIEW
WHERE TASK_NAME= 'cust_mv3'
AND SCRIPT_TYPE='IMPLEMENTATION';
```

Example 17–11 Save IMPLEMENTATION Output in a Script File

The following statements save the `IMPLEMENTATION` output in a script file located at `/myscript/mv_create3.sql`:

```
CREATE DIRECTORY TUNE_RESULTS AS '/myscript'
GRANT READ, WRITE ON DIRECTORY TUNE_RESULTS TO PUBLIC;
EXECUTE DBMS_ADVISOR.CREATE_FILE(DBMS_ADVISOR.GET_TASK_SCRIPT('cust_mv3'), -
   'TUNE_RESULTS', 'mv_create3.sql')
```

Part V

Data Warehouse Performance

This section deals with ways to improve your data warehouse's performance, and contains the following chapters:

18

Query Rewrite

This chapter discusses query rewrite in Oracle, and contains:

Overview of Query Rewrite

One of the major benefits of creating and maintaining materialized views is the ability to take advantage of query rewrite, which transforms a SQL statement expressed in terms of tables or views into a statement accessing one or more materialized views that are defined on the detail tables. The transformation is transparent to the end user or application, requiring no intervention and no reference to the materialized view in the SQL statement. Because query rewrite is transparent, materialized views can be added or dropped just like indexes without invalidating the SQL in the application code.

A query undergoes several checks to determine whether it is a candidate for query rewrite. If the query fails any of the checks, then the query is applied to the detail tables rather than the materialized view. This can be costly in terms of response time and processing power.

The optimizer uses two different methods to recognize when to rewrite a query in terms of a materialized view. The first method is based on matching the SQL text of the query with the SQL text of the materialized view definition. If the first method fails, the optimizer uses the more general method in which it compares joins, selections, data columns, grouping columns, and aggregate functions between the query and materialized views.

Query rewrite operates on queries and subqueries in the following types of SQL statements:

- `SELECT`
- `CREATE TABLE ... AS SELECT`
- `INSERT INTO ... SELECT`

It also operates on subqueries in the set operators `UNION`, `UNION ALL`, `INTERSECT`, and `MINUS`, and subqueries in DML statements such as `INSERT`, `DELETE`, and `UPDATE`.

Several factors affect whether or not a given query is rewritten to use one or more materialized views:

- Enabling or disabling query rewrite
  - By the `CREATE` or `ALTER` statement for individual materialized views
  - By the initialization parameter `QUERY_REWRITE_ENABLED`
  - By the `REWRITE` and `NOREWRITE` hints in SQL statements
- Rewrite integrity levels

- Dimensions and constraints

The DBMS_MVIEW.EXPLAIN_REWRITE procedure advises whether query rewrite is possible on a query and, if so, which materialized views will be used. It also explains why a query cannot be rewritten.

Cost-Based Rewrite

Query rewrite is available with cost-based optimization. Oracle Database optimizes the input query with and without rewrite and selects the least costly alternative. The optimizer rewrites a query by rewriting one or more query blocks, one at a time.

If query rewrite has a choice between several materialized views to rewrite a query block, it will select the ones which can result in reading in the least amount of data. After a materialized view has been selected for a rewrite, the optimizer then tests whether the rewritten query can be rewritten further with other materialized views. This process continues until no further rewrites are possible. Then the rewritten query is optimized and the original query is optimized. The optimizer compares these two optimizations and selects the least costly alternative.

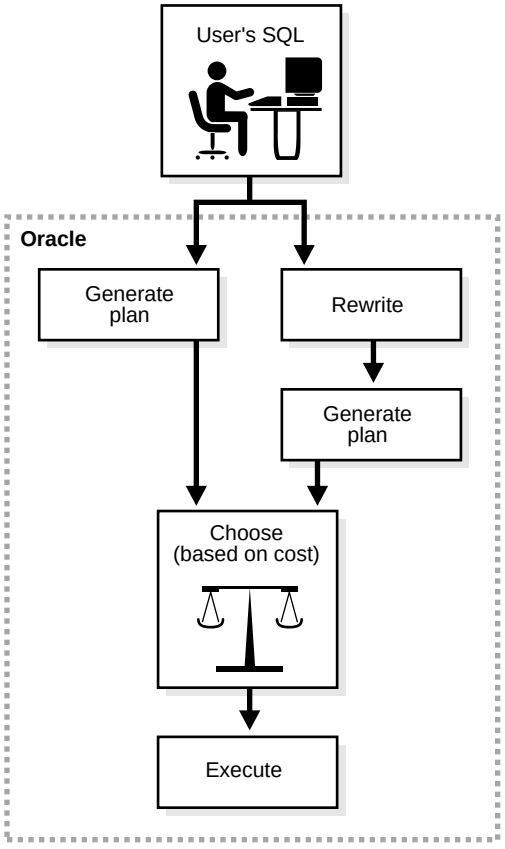
Because optimization is based on cost, it is important to collect statistics both on tables involved in the query and on the tables representing materialized views. Statistics are fundamental measures, such as the number of rows in a table, that are used to calculate the cost of a rewritten query. They are created by using the DBMS_STATS package.

Queries that contain inline or named views are also candidates for query rewrite. When a query contains a named view, the view name is used to do the matching between a materialized view and the query. When a query contains an inline view, the inline view can be merged into the query before matching between a materialized view and the query occurs.

In addition, if the inline view's text definition exactly matches with that of an inline view present in any eligible materialized view, general rewrite may be possible. This is because, whenever a materialized view contains exactly identical inline view text to the one present in a query, query rewrite treats such an inline view as a named view or a table.

Figure 18–1 presents a graphical view of the cost-based approach used during the rewrite process.

Figure 18–1 The Query Rewrite Process



When Does Oracle Rewrite a Query?

A query is rewritten only when a certain number of conditions are met:

- Query rewrite must be enabled for the session.
- A materialized view must be enabled for query rewrite.
- The rewrite integrity level should allow the use of the materialized view. For example, if a materialized view is not fresh and query rewrite integrity is set to ENFORCED, then the materialized view is not used.

- Either all or part of the results requested by the query must be obtainable from the precomputed result stored in the materialized view or views.

To determine this, the optimizer may depend on some of the data relationships declared by the user using constraints and dimensions. Such data relationships include hierarchies, referential integrity, and uniqueness of key data, and so on.

Enabling Query Rewrite

You must follow several steps to enable query rewrite:

1. Individual materialized views must have the ENABLE QUERY REWRITE clause.
2. The initialization parameter QUERY_REWRITE_ENABLED must be set to TRUE (this is the default).
3. Cost-based optimization must be used either by setting the initialization parameter OPTIMIZER_MODE to ALL_ROWS or FIRST_ROWS, or by analyzing the tables and setting OPTIMIZER_MODE to CHOOSE.

If step 1 has not been completed, a materialized view will never be eligible for query rewrite. You can specify ENABLE QUERY REWRITE either with the ALTER MATERIALIZED VIEW statement or when the materialized view is created, as illustrated in the following:

```
CREATE MATERIALIZED VIEW join_sales_time_product_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_id, p.prod_name, t.time_id, t.week_ending_day,
       s.channel_id, s.promo_id, s.cust_id, s.amount_sold
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id  AND s.prod_id = p.prod_id;
```

The NOREWRITE hint disables query rewrite in a SQL statement, overriding the QUERY_REWRITE_ENABLED parameter, and the REWRITE hint (when used with mv_name) restricts the eligible materialized views to those named in the hint.

You can use TUNE_MVIEW to optimize a CREATE MATERIALIZED VIEW statement to enable general QUERY REWRITE. This procedure is described in "Tuning Materialized Views for Fast Refresh and Query Rewrite" on page 17-47.

Initialization Parameters for Query Rewrite

Query rewrite requires three initialization parameter settings, and offers one optional setting as well. They are the following:

- OPTIMIZER_MODE = ALL_ROWS, FIRST_ROWS, or CHOOSE

  With OPTIMIZER_MODE set to choose, a query will not be rewritten unless at least one table referenced by it has been analyzed. This is because the rule-based optimizer is used when OPTIMIZER_MODE is set to choose and none of the tables referenced in a query have been analyzed.

- QUERY_REWRITE_ENABLED = TRUE.

  This option enables the query rewrite feature of the optimizer, enabling the optimizer to utilize materialized view to enhance performance. If set to FALSE, this option disables the query rewrite feature of the optimizer and directs the optimizer to rewrite queries using materialized views even when the estimated query cost of the unrewritten query is lower.

- QUERY_REWRITE_ENABLED = FORCE

  This option enables the query rewrite feature of the optimizer and directs the optimizer to rewrite queries using materialized views even when the estimated query cost of the unwritten query is lower.

- QUERY_REWRITE_INTEGRITY

  This parameter is optional, but must be set to STALE_TOLERATED, TRUSTED, or ENFORCED if it is specified (see "Accuracy of Query Rewrite" on page 18-7). It defaults to ENFORCED if it is undefined.

  By default, the integrity level is set to ENFORCED. In this mode, all constraints must be validated. Therefore, if you use ENABLE NOVALIDATE, certain types of query rewrite might not work. To enable query rewrite in this environment (where constraints have not been validated), you should set the integrity level to a lower level of granularity such as TRUSTED or STALE_TOLERATED.

Controlling Query Rewrite

A materialized view is only eligible for query rewrite if the ENABLE QUERY REWRITE clause has been specified, either initially when the materialized view was first created or subsequently with an ALTER MATERIALIZED VIEW statement.

You can set the initialization parameters described previously using the ALTER SYSTEM SET statement. For a given user's session, ALTER SESSION can be used to disable or enable query rewrite for that session only. An example is the following:

```
ALTER SESSION SET QUERY_REWRITE_ENABLED = TRUE;
```

You can set the level of query rewrite for a session, thus allowing different users to work at different integrity levels. The possible statements are:

```
ALTER SESSION SET QUERY_REWRITE_INTEGRITY = STALE_TOLERATED;
ALTER SESSION SET QUERY_REWRITE_INTEGRITY = TRUSTED;
ALTER SESSION SET QUERY_REWRITE_INTEGRITY = ENFORCED;
```

Accuracy of Query Rewrite

Query rewrite offers three levels of rewrite integrity that are controlled by the initialization parameter QUERY_REWRITE_INTEGRITY, which can either be set in your parameter file or controlled using an ALTER SYSTEM or ALTER SESSION statement. The three values are as follows:

- ENFORCED

  This is the default mode. The optimizer only uses fresh data from the materialized views and only use those relationships that are based on ENABLED VALIDATED primary, unique, or foreign key constraints.

- TRUSTED

  In TRUSTED mode, the optimizer trusts that the data in the materialized views is fresh and the relationships declared in dimensions and RELY constraints are correct. In this mode, the optimizer also uses prebuilt materialized views or materialized views based on views, and it uses relationships that are not enforced as well as those that are enforced. In this mode, the optimizer also trusts declared but not ENABLED VALIDATED primary or unique key constraints and data relationships specified using dimensions.

- STALE_TOLERATED

  In STALE_TOLERATED mode, the optimizer uses materialized views that are valid but contain stale data as well as those that contain fresh data. This mode offers the maximum rewrite capability but creates the risk of generating inaccurate results.

If rewrite integrity is set to the safest level, ENFORCED, the optimizer uses only enforced primary key constraints and referential integrity constraints to ensure that the results of the query are the same as the results when accessing the detail tables directly. If the rewrite integrity is set to levels other than ENFORCED, there are several situations where the output with rewrite can be different from that without it:

- A materialized view can be out of synchronization with the master copy of the data. This generally happens because the materialized view refresh procedure is

pending following bulk load or DML operations to one or more detail tables of a materialized view. At some data warehouse sites, this situation is desirable because it is not uncommon for some materialized views to be refreshed at certain time intervals.

- The relationships implied by the dimension objects are invalid. For example, values at a certain level in a hierarchy do not roll up to exactly one parent value.
- The values stored in a prebuilt materialized view table might be incorrect.
- A wrong answer can occur because of bad data relationships defined by unenforced table or view constraints.

Query Rewrite Hints

You can include hints in the SELECT blocks of your SQL statements to control whether query rewrite occurs. Using the NOREWRITE hint in a query prevents the optimizer from rewriting it.

The REWRITE hint with no argument in a query forces the optimizer to use a materialized view (if any) to rewrite it regardless of the cost. If you use the REWRITE(mv1,mv2,...) hint with arguments, this forces rewrite to select the most suitable materialized view from the list of names specified.

To prevent a rewrite, you can use the following statement:

```
SELECT /*+ NOREWRITE */ p.prod_subcategory, SUM(s.amount_sold)
FROM   sales s, products p WHERE  s.prod_id=p.prod_id
GROUP BY p.prod_subcategory;
```

To force a rewrite using sum_sales_pscat_week_mv, use the following statement:

```
SELECT /*+ REWRITE (sum_sales_pscat_week_mv) */
       p.prod_subcategory,  SUM(s.amount_sold)
FROM   sales s, products p WHERE  s.prod_id=p.prod_id
GROUP BY p.prod_subcategory;
```

Note that the scope of a rewrite hint is a query block. If a SQL statement consists of several query blocks (SELECT clauses), you need to specify a rewrite hint on each query block to control the rewrite for the entire statement.

Using the REWRITE_OR_ERROR hint in a query causes the following error if the query failed to rewrite:

```
ORA-30393: a query block in the statement did not rewrite
```

For example, the following query issues the ORA-30393 error when there are no suitable materialized views for query rewrite to use:

```
SELECT /*+ REWRITE_OR_ERROR */ p.prod_subcategory, SUM(s.amount_sold)
FROM sales s, products p WHERE s.prod_id = p.prod_id
GROUP BY p.prod_subcategory;
```

Privileges for Enabling Query Rewrite

Use of a materialized view is based not on privileges the user has on that materialized view, but on the privileges the user has on detail tables or views in the query.

The system privilege GRANT QUERY REWRITE lets you enable materialized views in your own schema for query rewrite only if all tables directly referenced by the materialized view are in that schema. The GRANT GLOBAL QUERY REWRITE privilege enables you to enable materialized views for query rewrite even if the materialized view references objects in other schemas. Alternatively, you can use the QUERY REWRITE object privilege on tables and views outside your schema.

The privileges for using materialized views for query rewrite are similar to those for definer's rights procedures.

Sample Schema and Materialized Views

The following sections use the sh sample schema and a few materialized views to illustrate how the optimizer uses data relationships to rewrite queries.

The query rewrite examples in this chapter mainly refer to the following materialized views. These materialized views do not necessarily represent the most efficient implementation for the sh schema. Instead, they are a base for demonstrating rewrite capabilities. Further examples demonstrating specific functionality can be found throughout this chapter.

The following materialized views contain joins and aggregates:

```
CREATE MATERIALIZED VIEW sum_sales_pscat_week_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_subcategory, t.week_ending_day,
       SUM(s.amount_sold) AS sum_amount_sold
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id  AND s.prod_id=p.prod_id
GROUP BY p.prod_subcategory, t.week_ending_day;

CREATE MATERIALIZED VIEW sum_sales_prod_week_mv
```

```
ENABLE QUERY REWRITE AS
SELECT p.prod_id, t.week_ending_day, s.cust_id,
       SUM(s.amount_sold) AS sum_amount_sold
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id AND s.prod_id=p.prod_id
GROUP BY p.prod_id, t.week_ending_day, s.cust_id;

CREATE MATERIALIZED VIEW sum_sales_pscat_month_city_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_subcategory, t.calendar_month_desc, c.cust_city,
       SUM(s.amount_sold) AS sum_amount_sold,
       COUNT(s.amount_sold) AS count_amount_sold
FROM   sales s, products p, times t, customers c
WHERE  s.time_id=t.time_id AND s.prod_id=p.prod_id AND s.cust_id=c.cust_id
GROUP BY p.prod_subcategory, t.calendar_month_desc, c.cust_city;
```

The following materialized views contain joins only:

```
CREATE MATERIALIZED VIEW join_sales_time_product_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_id, p.prod_name, t.time_id, t.week_ending_day,
       s.channel_id, s.promo_id, s.cust_id, s.amount_sold
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id AND s.prod_id = p.prod_id;

CREATE MATERIALIZED VIEW join_sales_time_product_oj_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_id, p.prod_name, t.time_id, t.week_ending_day,
       s.channel_id, s.promo_id, s.cust_id, s.amount_sold
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id AND s.prod_id=p.prod_id(+);
```

Although it is not a requirement, it is recommended that you collect statistics on the materialized views so that the optimizer can determine whether to rewrite the queries. You can do this either on a per-object base or for all newly created objects without statistics. The following is an example of a per-object base, shown for `join_sales_time_product_mv`:

```
EXECUTE DBMS_STATS.GATHER_TABLE_STATS ( -
   'SH','JOIN_SALES_TIME_PRODUCT_MV', estimate_percent   => 20, -
    block_sample   => TRUE, cascade   => TRUE);
```

The following illustrates a statistics collection for all newly created objects without statistics:

```
EXECUTE DBMS_STATS.GATHER_SCHEMA_STATS ( 'SH', -
```

```
options            => 'GATHER EMPTY', -
estimate_percent   =>  20, block_sample  =>  TRUE, -
cascade            =>  TRUE);
```

How Oracle Rewrites Queries

The optimizer uses a number of different methods to rewrite a query. The first step in determining whether query rewrite is possible is to see if the query satisfies the following prerequisites:

- Joins present in the materialized view are present in the SQL.
- There is sufficient data in the materialized view or views to answer the query.

After that, it must determine how it will rewrite the query. The simplest case occurs when the result stored in a materialized view exactly matches what is requested by a query. The optimizer makes this type of determination by comparing the text of the query with the text of the materialized view definition. This text match method is most straightforward but the number of queries eligible for this type of query rewrite is minimal.

When the text comparison test fails, the optimizer performs a series of generalized checks based on the joins, selections, grouping, aggregates, and column data fetched. This is accomplished by individually comparing various clauses (`SELECT`, `FROM`, `WHERE`, `HAVING`, or `GROUP BY`) of a query with those of a materialized view.

There are many different types of query rewrite that are possible and they can be categorized into the following areas:

- Text Match Rewrite Methods
- General Query Rewrite Methods

And general query rewrite can be divided into:

- Join Back
- Rollup Using a Dimension
- Compute Aggregates
- Filtering the Data

Text Match Rewrite Methods

The optimizer uses two methods, full and partial. In full text match, the entire text of a query is compared against the entire text of a materialized view definition (that

is, the entire SELECT expression), ignoring the white space during text comparison. Given the following query:

```
SELECT p.prod_subcategory, t.calendar_month_desc, c.cust_city,
       SUM(s.amount_sold) AS sum_amount_sold,
       COUNT(s.amount_sold) AS count_amount_sold
FROM   sales s, products p, times t, customers c
WHERE  s.time_id=t.time_id
AND    s.prod_id=p.prod_id
AND    s.cust_id=c.cust_id
GROUP BY p.prod_subcategory, t.calendar_month_desc, c.cust_city;
```

This query matches sum_sales_pscat_month_city_mv (white space excluded) and is rewritten as:

```
SELECT prod_subcategory, calendar_month_desc, cust_city,
       sum_amount_sold, count_amount_sold
FROM   sum_sales_pscat_month_city_mv;
```

When full text match fails, the optimizer then attempts a partial text match. In this method, the text starting from the FROM clause of a query is compared against the text starting with the FROM clause of a materialized view definition. Therefore, the following query can be rewritten:

```
SELECT p.prod_subcategory, t.calendar_month_desc, c.cust_city,
       AVG(s.amount_sold)
FROM   sales s, products p, times t, customers c
WHERE  s.time_id=t.time_id AND s.prod_id=p.prod_id
AND    s.cust_id=c.cust_id
GROUP BY p.prod_subcategory, t.calendar_month_desc, c.cust_city;
```

This query is rewritten as:

```
SELECT prod_subcategory, calendar_month_desc, cust_city,
       sum_amount_sold/count_amount_sold
FROM   sum_sales_pscat_month_city_mv;
```

Note that, under the partial text match rewrite method, the average of sales aggregate required by the query is computed using the sum of sales and count of sales aggregates stored in the materialized view.

When neither text match succeeds, the optimizer uses a general query rewrite method.

Text Match Capabilities

Text match rewrite can distinguish uppercase from lowercase. For example, the following statements are equivalent:

```
SELECT X, 'aBc' FROM Y

Select x, 'aBc' From y
```

General Query Rewrite Methods

The optimizer has a number of different types of query rewrite methods that it can choose from to answer a query. When text match rewrite is not possible, this group of rewrite methods is known as general query rewrite. The advantage of using these more advanced techniques is that one or more materialized views can be used to answer a number of different queries and the query does not always have to match the materialized view exactly for query rewrite to occur.

When using general query rewrite methods, the optimizer uses data relationships on which it can depend, such as primary and foreign key constraints and dimension objects. For example, primary key and foreign key relationships tell the optimizer that each row in the foreign key table joins with at most one row in the primary key table. Furthermore, if there is a NOT NULL constraint on the foreign key, it indicates that each row in the foreign key table must join to exactly one row in the primary key table. A dimension object will describe the relationship between, say, day, months, and year, which can be used to roll up data from the day to the month level.

Data relationships such as these are very important for query rewrite because they tell what type of result is produced by joins, grouping, or aggregation of data. Therefore, to maximize the rewritability of a large set of queries when such data relationships exist in a database, you should declare constraints and dimensions.

When are Constraints and Dimensions Needed?

Table 18–1 illustrates when dimensions and constraints are required for different types of query rewrite.

Table 18–1 Dimension and Constraint Requirements for Query Rewrite

Rewrite Checks	Dimensions	Primary Key/Foreign Key/Not Null Constraints
Matching SQL Text	Not Required	Not Required
Filtering the Data	Not Required	Not Required

Table 18–1 (Cont.) Dimension and Constraint Requirements for Query Rewrite

Rewrite Checks	Dimensions		Primary Key/Foreign Key/Not Null Constraints
Join Back	Required	OR	Required
Rollup Using a Dimension	Required		Not Required
Aggregate Rollup	Not Required		Not Required
Compute Aggregates	Not Required		Not Required

Join Back

If some column data requested by a query cannot be obtained from a materialized view, the optimizer further determines if it can be obtained based on a data relationship called a functional dependency. When the data in a column can determine data in another column, such a relationship is called a functional dependency or functional determinance. For example, if a table contains a primary key column called `prod_id` and another column called `prod_name`, then, given a `prod_id` value, it is possible to look up the corresponding `prod_name`. The opposite is not true, which means a `prod_name` value need not relate to a unique `prod_id`.

When the column data required by a query is not available from a materialized view, such column data can still be obtained by joining the materialized view back to the table that contains required column data provided the materialized view contains a key that functionally determines the required column data. For example, consider the following query:

```
SELECT p.prod_category, t.week_ending_day, SUM(s.amount_sold)
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id  AND s.prod_id=p.prod_id AND p.prod_category='CD'
GROUP BY p.prod_category, t.week_ending_day;
```

The materialized view `sum_sales_prod_week_mv` contains `p.prod_id`, but not `p.prod_category`. However, you can join `sum_sales_prod_week_mv` back to `products` to retrieve `prod_category` because `prod_id` functionally determines `prod_category`. The optimizer rewrites this query using `sum_sales_prod_week_mv` as follows:

```
SELECT p.prod_category, mv.week_ending_day, SUM(mv.sum_amount_sold)
FROM   sum_sales_prod_week_mv mv, products p
WHERE  mv.prod_id=p.prod_id AND p.prod_category='Photo'
GROUP BY p.prod_category, mv.week_ending_day;
```

Here the products table is called a joinback table because it was originally joined in the materialized view but joined again in the rewritten query.

You can declare functional dependency in two ways:

- Using the primary key constraint (as shown in the previous example)
- Using the DETERMINES clause of a dimension

The DETERMINES clause of a dimension definition might be the only way you could declare functional dependency when the column that determines another column cannot be a primary key. For example, the products table is a denormalized dimension table that has columns prod_id, prod_name, and prod_subcategory that functionally determines prod_subcat_desc and prod_category that determines prod_cat_desc.

The first functional dependency can be established by declaring prod_id as the primary key, but not the second functional dependency because the prod_subcategory column contains duplicate values. In this situation, you can use the DETERMINES clause of a dimension to declare the second functional dependency.

The following dimension definition illustrates how functional dependencies are declared:

```
CREATE DIMENSION products_dim
        LEVEL product           IS (products.prod_id)
        LEVEL subcategory       IS (products.prod_subcategory)
        LEVEL category          IS (products.prod_category)
        HIERARCHY prod_rollup (
                product         CHILD OF
                subcategory     CHILD OF
                category
        )
        ATTRIBUTE product DETERMINES products.prod_name
        ATTRIBUTE product DETERMINES products.prod_desc
        ATTRIBUTE subcategory DETERMINES products.prod_subcategory_desc
        ATTRIBUTE category DETERMINES products.prod_category_desc;
```

The hierarchy prod_rollup declares hierarchical relationships that are also 1:n functional dependencies. The 1:1 functional dependencies are declared using the DETERMINES clause, as seen when prod_subcategory functionally determines prod_subcat_desc.

Consider the following query:

```
SELECT p.prod_subcategory_desc, t.week_ending_day, SUM(s.amount_sold)
FROM   sales s, products p, times t
```

```
WHERE  s.time_id=t.time_id AND s.prod_id=p.prod_id
AND    p.prod_subcategory_desc LIKE '%Audio'
GROUP BY p.prod_subcategory_desc, t.week_ending_day;
```

This can be rewritten by joining `sum_sales_pscat_week_mv` to the `products` table so that `prod_subcat_desc` is available to evaluate the predicate. However, the join will be based on the `prod_subcategory` column, which is not a primary key in the `products` table; therefore, it allows duplicates. This is accomplished by using an inline view that selects distinct values and this view is joined to the materialized view as shown in the rewritten query.

```
SELECT iv.prod_subcat_desc, mv.week_ending_day, SUM(mv.sum_amount_sold)
FROM  sum_sales_pscat_week_mv mv,
     (SELECT DISTINCT prod_subcategory, prod_subcategory_desc
      FROM products) iv
WHERE  mv.prod_subcategory=iv.prod_subcategory
AND iv.prod_subcategory_desc LIKE '%Men'
GROUP BY iv.prod_subcategory_desc, mv.week_ending_day;
```

This type of rewrite is possible because `prod_subcategory` functionally determines `prod_subcategory_desc` as declared in the dimension.

Rollup Using a Dimension

When reporting is required at different levels in a hierarchy, materialized views do not have to be created at each level in the hierarchy provided dimensions have been defined. This is because query rewrite can use the relationship information in the dimension to roll up the data in the materialized view to the required level in the hierarchy.

In the following example, a query requests data grouped by `prod_category` while a materialized view stores data grouped by `prod_subcategory`. If `prod_subcategory` is a `CHILD OF` `prod_category` (see the dimension example earlier), the grouped data stored in the materialized view can be further grouped by `prod_category` when the query is rewritten. In other words, aggregates at `prod_subcategory` level (finer granularity) stored in a materialized view can be rolled up into aggregates at `prod_category` level (coarser granularity).

For example, consider the following query:

```
SELECT p.prod_category, t.week_ending_day, SUM(s.amount_sold) AS sum_amount
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id AND s.prod_id=p.prod_id
GROUP BY p.prod_category, t.week_ending_day;
```

Because prod_subcategory functionally determines prod_category, sum_sales_pscat_week_mv can be used with a joinback to products to retrieve prod_category column data, and then aggregates can be rolled up to prod_category level, as shown in the following:

```
SELECT pv.prod_category, mv.week_ending_day, SUM(mv.sum_amount_sold)
FROM   sum_sales_pscat_week_mv mv,
       (SELECT DISTINCT prod_subcategory, prod_category
        FROM products) pv
WHERE mv.prod_subcategory= pv.prod_subcategory
GROUP BY pv.prod_category, mv.week_ending_day;
```

Compute Aggregates

Query rewrite can also occur when the optimizer determines if the aggregates requested by a query can be derived or computed from one or more aggregates stored in a materialized view. For example, if a query requests AVG(X) and a materialized view contains SUM(X) and COUNT(X), then AVG(X) can be computed as SUM(X)/COUNT(X).

In addition, if it is determined that the rollup of aggregates stored in a materialized view is required, then, if it is possible, query rewrite also rolls up each aggregate requested by the query using aggregates in the materialized view.

For example, SUM(sales) at the city level can be rolled up to SUM(sales) at the state level by summing all SUM(sales) aggregates in a group with the same state value. However, AVG(sales) cannot be rolled up to a coarser level unless COUNT(sales) is also available in the materialized view. Similarly, VARIANCE(sales) or STDDEV(sales) cannot be rolled up unless COUNT(sales) and SUM(sales) are also available in the materialized view. For example, consider the following query:

```
ALTER TABLE times MODIFY CONSTRAINT time_pk RELY;
ALTER TABLE customers MODIFY CONSTRAINT customers_pk RELY;
ALTER TABLE sales MODIFY CONSTRAINT sales_time_pk RELY;
ALTER TABLE sales MODIFY CONSTRAINT sales_customer_fk RELY;
SELECT  p.prod_subcategory, AVG(s.amount_sold) AS avg_sales
FROM  sales s, products p WHERE s.prod_id = p.prod_id
GROUP BY p.prod_subcategory;
```

This statement can be rewritten with materialized view sum_sales_pscat_month_city_mv provided the join between sales and times and sales and customers are lossless and non-duplicating. Further, the query groups by prod_subcategory whereas the materialized view groups by prod_subcategory, calendar_month_desc and cust_city, which means the aggregates stored in

the materialized view will have to be rolled up. The optimizer rewrites the query as the following:

```
SELECT mv.prod_subcategory, SUM(mv.sum_amount_sold)/COUNT(mv.count_amount_sold)
   AS avg_sales
FROM sum_sales_pscat_month_city_mv mv
GROUP BY mv.prod_subcategory;
```

The argument of an aggregate such as SUM can be an arithmetic expression such as A+B. The optimizer tries to match an aggregate SUM(A+B) in a query with an aggregate SUM(A+B) or SUM(B+A) stored in a materialized view. In other words, expression equivalence is used when matching the argument of an aggregate in a query with the argument of a similar aggregate in a materialized view. To accomplish this, Oracle converts the aggregate argument expression into a canonical form such that two different but equivalent expressions convert into the same canonical form. For example, A*(B-C), A*B-C*A, (B-C)*A, and -A*C+A*B all convert into the same canonical form and, therefore, they are successfully matched.

Filtering the Data

Oracle supports rewriting of queries so that they will use materialized views in which the HAVING or WHERE clause of the materialized view contains a selection of a subset of the data in a table or tables. For example, only those customers who live in New Hampshire.

To perform this type of query rewrite, Oracle must determine if the data requested in the query is contained in, or is a subset of, the data stored in the materialized view. The following sections detail the conditions where Oracle can solve this problem and thus rewrite a query to use a materialized view that contains a filtered portion of the data in the detail table.

To determine if query rewrite can occur on filtered data, a selection compatibility check is performed when both the query and the materialized view contain selections (non-joins) and the check is done on the WHERE as well as the HAVING clause. If the materialized view contains selections and the query does not, then the selection compatibility check fails because the materialized view is more restrictive than the query. If the query has selections and the materialized view does not, then the selection compatibility check is not needed.

A materialized view's WHERE or HAVING clause can contain a join, a selection, or both, and still be used by a rewritten query. Predicate clauses containing expressions, or selecting rows based on the values of particular columns, are examples of non-join predicates.

Query Rewrite Definitions Before describing what is possible when query rewrite works with filtered data, the following definitions are useful:

- *join relop*

  Is one of the following (=, <, <=, >, >=)

- *selection relop*

  Is one of the following (=, <, <=, >, >=, !=, [NOT] BETWEEN | IN| LIKE |NULL)

- *join predicate*

  Is of the form (*column1 join relop column2*), where columns are from different tables within the same FROM clause in the current query block. So, for example, an outer reference is not possible.

- *selection predicate*

  Is of the form *LHS-expression relop RHS-expression*, where LHS means left-hand side and RHS means right-hand side. All non-join predicates are selection predicates. The left-hand side usually contains a column and the right-hand side contains the values. For example, color='red' means the left-hand side is color and the right-hand side is 'red' and the relational operator is (=).

- *LHS-constrained*

  When comparing a selection from the query with a selection from the materialized view, if the left-hand side of both selections match, the selections are said to be LHS-constrained or just constrained for short.

- *RHS-constrained*

  When comparing a selection from the query with a selection from the materialized view, if the right-hand side of both selections match, the selections are said to be RHS-constrained or just constrained. Note that before comparing the selections, the LHS/RHS-expression is converted to a canonical form and then the comparison is done. This means that expressions such as column1 + 5 and 5 + column1 will match and be constrained.

WHERE Clause Guidelines Although query rewrite on filtered data does not restrict the general form of the WHERE clause, there is an optimal pattern and, normally, most queries fall into this pattern as follows:

```
(join predicate AND join predicate AND ....)  AND
   (selection predicate  AND|OR  selection predicate .... )
```

If the WHERE clause has an OR at the top, then the optimizer first checks for common predicates under the OR. If found, the common predicates are factored out from under the OR, then joined with an AND back to the OR. This helps to put the WHERE clause into the optimal pattern. This is done only if OR occurs at the top of the WHERE clause. For example, if the WHERE clause is the following:

```
(sales.prod_id = prod.prod_id AND prod.prod_name = 'Kids Polo Shirt')
  OR (sales.prod_id = prod.prod_id  AND prod.prod_name = 'Kids Shorts')
```

The join is factored out and the WHERE clause becomes:

```
(sales.prod_id = prod.prod_id) AND (prod.prod_name = 'Kids Polo Shirt'
  OR prod.prod_name = 'Kids Shorts')
```

Thus putting the WHERE clause into the most optimal pattern.

Selection Categories Selections are categorized into the following cases:

- Simple

  Simple selections are of the form *expression relop constant*.

- Complex

  Complex selections are of the form *expression relop expression*.

- Range

  Range selections are of a form such as `WHERE (cust_last_name BETWEEN 'abacrombe' AND 'anakin')`.

  Note that simple selections with relational operators `(<,<=,>,>=)` are also considered range selections.

- IN-lists

  Single and multi-column IN-lists such as `WHERE(prod_id) IN (102, 233, ....)`.

  Note that selections of the form `(column1='v1' OR column1='v2' OR column1='v3' OR ....)` are treated as a group and classified as an IN-list.

- `IS [NOT] NULL`
- `[NOT] LIKE`
- Other

  Other selections are when it cannot determine the boundaries for the data. For example, EXISTS.

When comparing a selection from the query with a selection from the materialized view, the left-hand side of both selections are compared and if they match they are said to be LHS-constrained or constrained for short.

If the selections are constrained, then the right-hand side values are checked for containment. That is, the RHS values of the query selection must be contained by right-hand side values of the materialized view selection.

Examples of Query Rewrite Selection Here are a number of examples showing how query rewrite can still occur when the data is being filtered.

Example 18–1 Single Value Selection

If the query contains the following clause:

```
WHERE prod_id = 102
```

And, if a materialized view contains the following clause:

```
WHERE prod_id BETWEEN 0 AND 200
```

Then, the selections are constrained on `prod_id` and the right-hand side value of the query `102` is within the range of the materialized view, so query rewrite is possible.

Example 18–2 Bounded Range Selection

A selection can be a bounded range (a range with an upper and lower value). For example, if the query contains the following clause:

```
WHERE prod_id > 10 AND prod_id < 50
```

And if a materialized view contains the following clause:

```
WHERE prod_id BETWEEN 0 AND 200
```

Then, the selections are constrained on `prod_id` and the query range is within the materialized view range. In this example, notice that both query selections are constrained by the same materialized view selection.

Example 18–3 Selection With Expression

If the query contains the following clause:

```
WHERE (sales.amount_sold * .07) BETWEEN 1.00 AND 100.00
```

And if a materialized view contains the following clause:

```
WHERE (sales.amount_sold * .07) BETWEEN 0.0 AND 200.00
```

Then, the selections are constrained on `(sales.amount_sold *.07)` and the right-hand side value of the query is within the range of the materialized view, therefore query rewrite is possible. Complex selections require that both the left-hand side and right-hand side be matched (for example, when the left-hand side and the right-hand side are constrained).

Example 18–4 Exact Match Selections

If the query contains the following clause:

```
WHERE (cost.unit_price * 0.95) > (cost_unit_cost  * 1.25)
```

And if a materialized view contains the following:

```
WHERE (cost.unit_price * 0.95) > (cost_unit_cost  * 1.25)
```

If the left-hand side and the right-hand side are constrained and the *selection_relop* is the same, then the selection can usually be dropped from the rewritten query. Otherwise, the selection must be kept to filter out extra data from the materialized view.

If query rewrite can drop the selection from the rewritten query, all columns from the selection may not have to be in the materialized view so more rewrites can be done. This ensures that the materialized view data is not more restrictive that the query.

Example 18–5 More Selection in the Query

Selections in the query do not have to be constrained by any selections in the materialized view but, if they are, then the right-hand side values must be contained by the materialized view. For example, if the query contains the following clause:

```
WHERE prod_name = 'Shorts' AND prod_category = 'Men'
```

And if a materialized view contains the following clause:

```
WHERE prod_category = 'Men'
```

Then, in this example, only selection with `prod_category` is constrained. The query has an extra selection that is not constrained but this is acceptable because if

the materialized view selects prod_name or selects a column that can be joined back to the detail table to get prod_name, then the query rewrite is possible.

Example 18–6 No Rewrite Because of Fewer Selections in the Query

If the query contains the following clause:

```
WHERE prod_category = 'Men'
```

And if a materialized view contains the following clause:

```
WHERE prod_name = 'Shorts' AND prod_category = 'Men'
```

Then, the materialized view selection with prod_name is not constrained. The materialized view is more restrictive that the query because it only contains the product Shorts, therefore, query rewrite will not occur.

Example 18–7 Multi-Column IN-List Selections

Query rewrite also checks for cases where the query has a multi-column IN-list where the columns are fully constrained by individual columns from the materialized view single column IN-lists. For example, if the query contains the following:

```
WHERE (prod_id, cust_id) IN ((1022, 1000), (1033, 2000))
```

And if a materialized view contains the following:

```
WHERE prod_id IN (1022,1033) AND cust_id IN (1000, 2000)
```

Then, the materialized view IN-lists are constrained by the columns in the query multi-column IN-list. Furthermore, the right-hand side values of the query selection are contained by the materialized view so that rewrite will occur.

Example 18–8 Selections Using IN-Lists

Selection compatibility also checks for cases where the materialized view has a multi-column IN-list where the columns are fully constrained by individual columns or columns from IN-lists in the query. For example, if the query contains the following:

```
WHERE prod_id = 1022 AND cust_id IN (1000, 2000)
```

And if a materialized view contains the following:

```
WHERE (prod_id, cust_id) IN ((1022, 1000), (1022, 2000))
```

Then, the materialized view IN-list columns are fully constrained by the columns in the query selections. Furthermore, the right-hand side values of the query selection are contained by the materialized view. So rewrite succeeds.

Example 18–9 Multiple Selections and Disjuncts

If the query contains the following clause:

```
WHERE (city_population > 15000 AND city_population < 25000
   AND state_name = 'New Hampshire')
```

And if a materialized view contains the following clause:

```
WHERE (city_population < 5000 AND state_name = 'New York') OR
   (city_population BETWEEN 10000 AND 50000 AND state_name = 'New Hampshire')
```

Then, the query has a single disjunct (group of selections separated by AND) and the materialized view has two disjuncts separated by OR. The query disjunct is contained by the second materialized view disjunct so selection compatibility succeeds. It is clear that the materialized view contains more data than needed by the query so the query can be rewritten.

Dropping Selections in the Rewritten Query

For example, consider the following simple materialized view definition:

```
CREATE MATERIALIZED VIEW cal_month_sales_id_mv
BUILD IMMEDIATE
REFRESH FORCE
ENABLE QUERY REWRITE AS
SELECT    t.calendar_month_desc, SUM(s.amount_sold) AS dollars
FROM   sales s, times t
WHERE  s.time_id = t.time_id AND s.cust_id = 10
GROUP BY t.calendar_month_desc;
```

The following query could be rewritten to use cal_month_sales_id_mv because the query asks for the amount where the cust_id is 10 and this is contained in the materialized view.

```
SELECT t.calendar_month_desc, SUM(s.amount_sold) AS dollars
FROM times t, sales s WHERE s.time_id = t.time_id AND s.cust_id = 10
GROUP BY t.calendar_month_desc;
```

Because the predicate `s.cust_id = 10` selects the same data in the query and in the materialized view, it is dropped from the rewritten query. This means the rewritten query is the following:

```
SELECT mv.calendar_month_desc, mv.dollars FROM cal_month_sales_id_mv mv;
```

Handling of HAVING Clause in Query Rewrite

Query rewrite can also occur when the query specifies a range of values for an aggregate in the `HAVING` clause, such as `SUM(s.amount_sold) BETWEEN 10000 AND 20000`, as long as the range specified is within the range specified in the materialized view.

```
CREATE MATERIALIZED VIEW product_sales_mv
BUILD IMMEDIATE
REFRESH FORCE
ENABLE QUERY REWRITE AS
SELECT p.prod_name, SUM(s.amount_sold) AS dollar_sales
FROM products p, sales s
WHERE p.prod_id = s.prod_id
GROUP BY prod_name
HAVING SUM(s.amount_sold) BETWEEN 5000 AND 50000;
```

Then, a query such as the following could be rewritten:

```
SELECT p.prod_name, SUM(s.amount_sold) AS dollar_sales
FROM products p, sales s WHERE p.prod_id = s.prod_id
GROUP BY prod_name
HAVING SUM(s.amount_sold) BETWEEN 10000 AND 20000;
```

This query is rewritten as follows:

```
SELECT prod_name, dollar_sales FROM product_sales_mv
WHERE dollar_sales BETWEEN 10000 AND 20000;
```

Handling Expressions in Query Rewrite

Rewrite with some expressions is also supported when the expression evaluates to a constant, such as `TO_DATE('12-SEP-1999','DD-Mon-YYYY')`. For example, if an existing materialized view is defined as:

```
CREATE MATERIALIZED VIEW sales_on_valentines_day_99_mv
BUILD IMMEDIATE
REFRESH FORCE
ENABLE QUERY REWRITE AS
SELECT prod_id, cust_id, amount_sold
```

```
FROM times t, sales s WHERE s.time_id = t.time_id
AND t.time_id = TO_DATE('14-FEB-1999', 'DD-MON-YYYY');
```

Then the following query can be rewritten:

```
SELECT prod_id, cust_id, amount_sold
FROM sales s, times t WHERE s.time_id = t.time_id
AND t.time_id = TO_DATE('14-FEB-1999', 'DD-MON-YYYY');
```

This query would be rewritten as follows:

```
SELECT * FROM sales_on_valentines_day_99_mv;
```

Handling IN-Lists in Query Rewrite

For example, given the following materialized view definition:

```
CREATE MATERIALIZED VIEW popular_promo_sales_mv
BUILD IMMEDIATE
REFRESH FORCE
ENABLE QUERY REWRITE AS
SELECT p.promo_name, SUM(s.amount_sold) AS sum_amount_sold
FROM  promotions p, sales s
WHERE s.promo_id = p.promo_id
AND promo_name IN ('coupon', 'premium', 'giveaway')
GROUP BY promo_name;
```

The following query can be rewritten:

```
SELECT p.promo_name, SUM(s.amount_sold)
FROM  promotions p, sales s
WHERE s.promo_id = p.promo_id AND promo_name IN ('coupon', 'premium')
GROUP BY promo_name;
```

This query is rewritten as follows:

```
SELECT * FROM popular_promo_sales_mv WHERE promo_name IN ('coupon', 'premium');
```

You can also use expressions in selection predicates. This process resembles the following:

```
expression relational operator constant
```

Where *expression* can be any arbitrary arithmetic expression allowed by the Oracle Database. The expression in the materialized view and the query must match. Oracle attempts to discern expressions that are logically equivalent, such as A+B and B+A, and will always recognize identical expressions as being equivalent.

You can also use queries with an expression on both sides of the operator or user-defined functions as operators. Query rewrite occurs when the complex predicate in the materialized view and the query are logically equivalent. This means that, unlike exact text match, terms could be in a different order and rewrite can still occur, as long as the expressions are equivalent.

In addition, selection predicates can be joined with an AND operator in a query and the query can still be rewritten to use a materialized view as long as every restriction on the data selected by the query is matched by a restriction in the definition of the materialized view. Again, this does not mean an exact text match, but that the restrictions on the data selected must be a logical match. Also, the query may be more restrictive in its selection of data and still be eligible, but it can never be less restrictive than the definition of the materialized view and still be eligible for rewrite. For example, given the preceding materialized view definition, a query such as the following can be rewritten:

```
SELECT p.promo_name, SUM(s.amount_sold)
FROM  promotions p, sales s
WHERE s.promo_id = p.promo_id AND promo_name  = 'coupon'
GROUP BY promo_name
HAVING SUM(s.amount_sold) > 1000;
```

This query is rewritten as follows:

```
SELECT * FROM popular_promo_sales_mv
WHERE promo_name = 'coupon' AND sum_amount_sold > 1000;
```

In this case, the query is more restrictive than the definition of the materialized view, so rewrite can occur. However, if the query had selected `promo_category`, then it could not have been rewritten against the materialized view, because the materialized view definition does not contain that column.

For another example, if the definition of a materialized view restricts a city name column to Boston, then a query that selects Seattle as a value for this column can never be rewritten with that materialized view, but a query that restricts city name to Boston and restricts a column value that is not restricted in the materialized view could be rewritten to use the materialized view.

All the rules noted previously also apply when predicates are combined with an OR operator. The simple predicates, or simple predicates connected by OR operators, are considered separately. Each predicate in the query must be contained in the materialized view if rewrite is to occur. For example, the query could have a restriction such as `city='Boston'` OR `city ='Seattle'` and to be eligible for rewrite, the materialized view that the query might be rewritten against must have

the same restriction. In fact, the materialized view could have additional restrictions, such as `city='Boston' OR city='Seattle' OR city='Cleveland'` and rewrite might still be possible.

Note, however, that the reverse is not true. If the query had the restriction `city = 'Boston' OR city='Seattle' OR city='Cleveland'` and the materialized view only had the restriction `city='Boston' OR city='Seattle'`, then rewrite would not be possible because, with a single materialized view, the query seeks more data than is contained in the restricted subset of data stored in the materialized view.

Checks Made by Query Rewrite

For query rewrite to occur, there are a number of checks that the data must pass. These checks are:

- Join Compatibility Check
- Data Sufficiency Check
- Grouping Compatibility Check
- Aggregate Computability Check

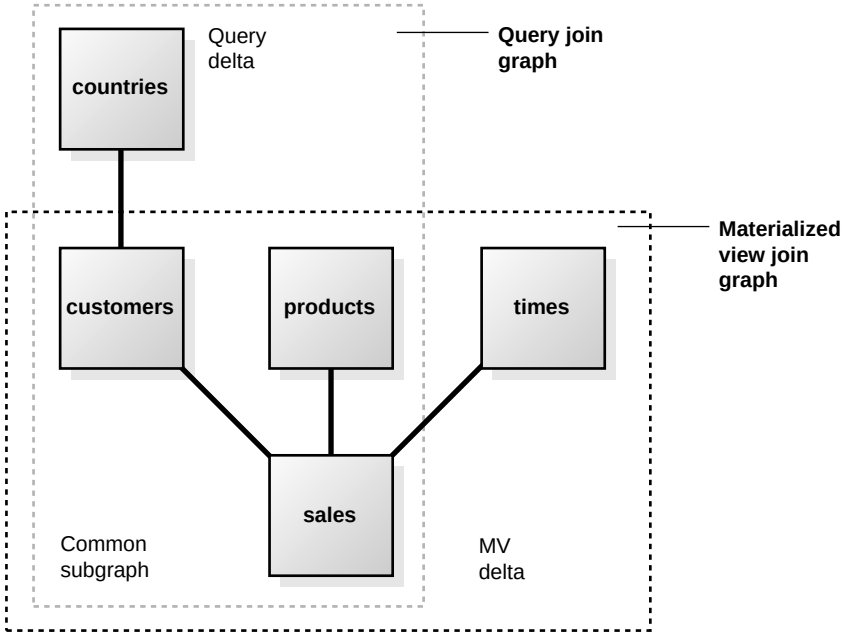
Join Compatibility Check

In this check, the joins in a query are compared against the joins in a materialized view. In general, this comparison results in the classification of joins into three categories:

- Common joins that occur in both the query and the materialized view. These joins form the common subgraph.
- Delta joins that occur in the query but not in the materialized view. These joins form the query delta subgraph.
- Delta joins that occur in the materialized view but not in the query. These joins form the materialized view delta subgraph.

These can be visualized as shown in Figure 18–2.

Figure 18–2 Query Rewrite Subgraphs



Common Joins The common join pairs between the two must be of the same type, or the join in the query must be derivable from the join in the materialized view. For example, if a materialized view contains an outer join of table A with table B, and a query contains an inner join of table A with table B, the result of the inner join can be derived by filtering the antijoin rows from the result of the outer join. For example, consider the following query:

```
SELECT p.prod_name, t.week_ending_day, SUM(amount_sold)
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id AND s.prod_id = p.prod_id
AND t. week_ending_day BETWEEN TO_DATE('01-AUG-1999', 'DD-MON-YYYY')
                           AND TO_DATE('10-AUG-1999', 'DD-MON-YYYY')
GROUP BY prod_name, week_ending_day;
```

The common joins between this query and the materialized view `join_sales_time_product_mv` are:

```
s.time_id = t.time_id AND s.prod_id = p.prod_id
```

They match exactly and the query can be rewritten as follows:

```
SELECT prod_name, week_ending_day, SUM(amount_sold)
FROM   join_sales_time_product_mv
WHERE  week_ending_day BETWEEN TO_DATE('01-AUG-1999','DD-MON-YYYY')
                       AND   TO_DATE('10-AUG-1999','DD-MON-YYYY')
GROUP BY prod_name, week_ending_day;
```

The query could also be answered using the `join_sales_time_product_oj_mv` materialized view where inner joins in the query can be derived from outer joins in the materialized view. The rewritten version will (transparently to the user) filter out the antijoin rows. The rewritten query will have the following structure:

```
SELECT prod_name, week_ending_day, SUM(amount_sold)
FROM   join_sales_time_product_oj_mv
WHERE  week_ending_day BETWEEN TO_DATE('01-AUG-1999','DD-MON-YYYY')
  AND  TO_DATE('10-AUG-1999','DD-MON-YYYY') AND prod_id IS NOT NULL
GROUP BY prod_name, week_ending_day;
```

In general, if you use an outer join in a materialized view containing only joins, you should put in the materialized view either the primary key or the rowid on the right side of the outer join. For example, in the previous example, `join_sales_time_product_oj_mv`, there is a primary key on both `sales` and `products`.

Another example of when a materialized view containing only joins is used is the case of a semijoin rewrites. That is, a query contains either an `EXISTS` or an `IN` subquery with a single table. Consider the following query, which reports the products that had sales greater than $1,000:

```
SELECT DISTINCT prod_name
FROM products p
WHERE EXISTS (SELECT * FROM sales s
              WHERE p.prod_id=s.prod_id AND s.amount_sold > 1000);
```

This query could also be represented as:

```
SELECT DISTINCT prod_name
FROM products p WHERE p.prod_id IN (SELECT s.prod_id FROM sales s
                    WHERE s.amount_sold > 1000);
```

This query contains a semijoin (`s.prod_id = p.prod_id`) between the `products` and the `sales` table.

This query can be rewritten to use either the `join_sales_time_product_mv` materialized view, if foreign key constraints are active or `join_sales_time_product_oj_mv` materialized view, if primary keys are active. Observe that both materialized views contain `s.prod_id=p.prod_id`, which can be used to derive

the semijoin in the query. The query is rewritten with join_sales_time_product_mv as follows:

```
SELECT prod_name
FROM (SELECT DISTINCT prod_name FROM  join_sales_time_product_mv
      WHERE amount_sold > 1000);
```

If the materialized view join_sales_time_product_mv is partitioned by time_id, then this query is likely to be more efficient than the original query because the original join between sales and products has been avoided. The query could be rewritten using join_sales_time_product_oj_mv as follows.

```
SELECT prod_name
FROM (SELECT DISTINCT prod_name FROM  join_sales_time_product_oj_mv
      WHERE amount_sold > 1000 AND prod_id IS NOT NULL);
```

Rewrites with semi-joins are restricted to materialized views with joins only and are not possible for materialized views with joins and aggregates.

Query Delta Joins A **query delta join** is a join that appears in the query but not in the materialized view. Any number and type of delta joins in a query are allowed and they are simply retained when the query is rewritten with a materialized view. In order for the retained join to work, the materialized view must contain the joining key. Upon rewrite, the materialized view is joined to the appropriate tables in the query delta. For example, consider the following query:

```
SELECT p.prod_name, t.week_ending_day, c.cust_city, SUM(s.amount_sold)
FROM   sales s, products p, times t, customers c
WHERE  s.time_id=t.time_id AND s.prod_id = p.prod_id
AND    s.cust_id = c.cust_id
GROUP BY prod_name, week_ending_day, cust_city;
```

Using the materialized view join_sales_time_product_mv, common joins are: s.time_id=t.time_id and s.prod_id=p.prod_id. The delta join in the query is s.cust_id=c.cust_id. The rewritten form will then join the join_sales_time_product_mv materialized view with the customers table as follows:

```
SELECT mv.prod_name, mv.week_ending_day, c.cust_city, SUM(mv.amount_sold)
FROM   join_sales_time_product_mv mv, customers c
WHERE  mv.cust_id = c.cust_id
GROUP BY prod_name, week_ending_day, cust_city;
```

Materialized View Delta Joins A **materialized view delta join** is a join that appears in the materialized view but not the query. All delta joins in a materialized view are required to be lossless with respect to the result of common joins. A lossless join

guarantees that the result of common joins is not restricted. A **lossless** join is one where, if two tables called A and B are joined together, rows in table A will always match with rows in table B and no data will be lost, hence the term lossless join. For example, every row with the foreign key matches a row with a primary key provided no nulls are allowed in the foreign key. Therefore, to guarantee a lossless join, it is necessary to have FOREIGN KEY, PRIMARY KEY, and NOT NULL constraints on appropriate join keys. Alternatively, if the join between tables A and B is an outer join (A being the outer table), it is lossless as it preserves all rows of table A.

All delta joins in a materialized view are required to be non-duplicating with respect to the result of common joins. A non-duplicating join guarantees that the result of common joins is not duplicated. For example, a non-duplicating join is one where, if table A and table B are joined together, rows in table A will match with at most one row in table B and no duplication occurs. To guarantee a non-duplicating join, the key in table B must be constrained to unique values by using a primary key or unique constraint.

Consider the following query that joins `sales` and `times`:

```
SELECT t.week_ending_day, SUM(s.amount_sold)
FROM   sales s, times t
WHERE s.time_id = t.time_id AND t.week_ending_day BETWEEN TO_DATE
 ('01-AUG-1999', 'DD-MON-YYYY') AND TO_DATE('10-AUG-1999', 'DD-MON-YYYY')
GROUP BY week_ending_day;
```

The materialized view `join_sales_time_product_mv` has an additional join (`s.prod_id=p.prod_id`) between `sales` and `products`. This is the delta join in `join_sales_time_product_mv`. You can rewrite the query if this join is lossless and non-duplicating. This is the case if `s.prod_id` is a foreign key to `p.prod_id` and is not null. The query is therefore rewritten as:

```
SELECT week_ending_day, SUM(amount_sold)
FROM   join_sales_time_product_mv
WHERE  week_ending_day BETWEEN TO_DATE('01-AUG-1999', 'DD-MON-YYYY')
                       AND     TO_DATE('10-AUG-1999', 'DD-MON-YYYY')
GROUP BY week_ending_day;
```

The query can also be rewritten with the materialized view `join_sales_time_product_mv_oj` where foreign key constraints are not needed. This view contains an outer join (`s.prod_id=p.prod_id(+)`) between `sales` and `products`. This makes the join lossless. If `p.prod_id` is a primary key, then the non-duplicating condition is satisfied as well and optimizer rewrites the query as follows:

```
SELECT week_ending_day, SUM(amount_sold)
FROM   join_sales_time_product_oj_mv
```

```
WHERE  week_ending_day BETWEEN TO_DATE('01-AUG-1999', 'DD-MON-YYYY')
  AND  TO_DATE('10-AUG-1999', 'DD-MON-YYYY')
GROUP BY week_ending_day;
```

Note that the outer join in the definition of `join_sales_time_product_mv_oj` is not necessary because the parent key - foreign key relationship between `sales` and `products` in the `sh` schema is already lossless. It is used for demonstration purposes only, and would be necessary if `sales.prod_id` were nullable, thus violating the losslessness of the join condition `sales.prod_id = products.prod_id`.

Current limitations restrict most rewrites with outer joins to materialized views with joins only. There is limited support for rewrites with materialized aggregate views with outer joins, so those views should rely on foreign key constraints to assure losslessness of materialized view delta joins.

Join Equivalence Recognition Query rewrite is able to make many transformations based upon the recognition of equivalent joins. Query rewrite recognizes the following construct as being equivalent to a join:

```
WHERE table1.column1 = F(args)   /* sub-expression A */
AND table2.column2 = F(args)     /* sub-expression B */
```

If `F(args)` is a PL/SQL function that is declared to be deterministic and the arguments to both invocations of `F` are the same, then the combination of subexpression `A` with subexpression `B` be can be recognized as a join between `table1.column1` and `table2.column2`. That is, the following expression is equivalent to the previous expression:

```
WHERE table1.column1 = F(args)          /* sub-expression A */
AND table2.column2 = F(args)            /* sub-expression B */
AND table1.column1 = table2.column2     /* join-expression J */
```

Because join-expression `J` can be inferred from sub-expression `A` and subexpression `B`, the inferred join can be used to match a corresponding join of `table1.column1 = table2.column2` in a materialized view.

Data Sufficiency Check

In this check, the optimizer determines if the necessary column data requested by a query can be obtained from a materialized view. For this, the equivalence of one column with another is used. For example, if an inner join between table `A` and table `B` is based on a join predicate `A.X = B.X`, then the data in column `A.X` will equal the data in column `B.X` in the result of the join. This data property is used to match

column A.X in a query with column B.X in a materialized view or vice versa. For example, consider the following query:

```
SELECT p.prod_name, s.time_id, t.week_ending_day, SUM(s.amount_sold)
FROM sales s, products p, times t
WHERE s.time_id=t.time_id AND s.prod_id = p.prod_id
GROUP BY p.prod_name, s.time_id, t.week_ending_day;
```

This query can be answered with join_sales_time_product_mv even though the materialized view does not have s.time_id. Instead, it has t.time_id, which, through a join condition s.time_id=t.time_id, is equivalent to s.time_id. Thus, the optimizer might select the following rewrite:

```
SELECT prod_name, time_id, week_ending_day, SUM(amount_sold)
FROM join_sales_time_product_mv
GROUP BY prod_name, time_id, week_ending_day;
```

Grouping Compatibility Check

This check is required only if both the materialized view and the query contain a GROUP BY clause. The optimizer first determines if the grouping of data requested by a query is exactly the same as the grouping of data stored in a materialized view. In other words, the level of grouping is the same in both the query and the materialized view. If the materialized views groups on all the columns and expressions in the query and also groups on additional columns or expressions, query rewrite can reaggregate the materialized view over the grouping columns and expressions of the query to derive the same result requested by the query.

Aggregate Computability Check

This check is required only if both the query and the materialized view contain aggregates. Here the optimizer determines if the aggregates requested by a query can be derived or computed from one or more aggregates stored in a materialized view. For example, if a query requests AVG(X) and a materialized view contains SUM(X) and COUNT(X), then AVG(X) can be computed as SUM(X)/COUNT(X).

If the grouping compatibility check determined that the rollup of aggregates stored in a materialized view is required, then the aggregate computability check determines if it is possible to roll up each aggregate requested by the query using aggregates in the materialized view.

Other Cases for Query Rewrite

The following discusses some of the other cases when query rewrite is possible:

Query Rewrite Using Partially Stale Materialized Views

When a partition of the detail table is updated, only specific sections of the materialized view are marked stale. The materialized view must have information that can identify the partition of the table corresponding to a particular row or group of the materialized view. The simplest scenario is when the partitioning key of the table is available in the SELECT list of the materialized view because this is the easiest way to map a row to a stale partition. The key points when using partially stale materialized views are:

- Query rewrite can use a materialized view in ENFORCED or TRUSTED mode if the rows from the materialized view used to answer the query are known to be FRESH.
- The fresh rows in the materialized view are identified by adding selection predicates to the materialized view's WHERE clause. Oracle will rewrite a query with this materialized view if its answer is contained within this (restricted) materialized view.

The fact table `sales` is partitioned based on ranges of `time_id` as follows:

```
PARTITION BY RANGE (time_id)
(PARTITION SALES_Q1_1998
           VALUES LESS THAN (TO_DATE('01-APR-1998', 'DD-MON-YYYY')),
 PARTITION SALES_Q2_1998
           VALUES LESS THAN (TO_DATE('01-JUL-1998', 'DD-MON-YYYY')),
 PARTITION SALES_Q3_1998
           VALUES LESS THAN (TO_DATE('01-OCT-1998', 'DD-MON-YYYY')),
...
```

Suppose you have a materialized view grouping by `time_id` as follows:

```
CREATE MATERIALIZED VIEW sum_sales_per_city_mv
ENABLE QUERY REWRITE AS
SELECT s.time_id, p.prod_subcategory, c.cust_city,
       SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id
GROUP BY time_id, prod_subcategory, cust_city;
```

Also suppose new data will be inserted for December 2000, which will be assigned to partition `sales_q4_2000`. For testing purposes, you can apply an arbitrary DML operation on `sales`, changing a different partition than `sales_q1_2000` as the following query requests data in this partition when this materialized view is fresh. For example, the following:

```
INSERT INTO SALES VALUES(17, 10, '01-DEC-2000', 4, 380, 123.45, 54321);
```

Until a refresh is done, the materialized view is generically stale and cannot be used for unlimited rewrite in enforced mode. However, because the table `sales` is partitioned and not all partitions have been modified, Oracle can identify all partitions that have not been touched. The optimizer can identify the fresh rows in the materialized view (the data which is unaffected by updates since the last refresh operation) by implicitly adding selection predicates to the materialized view defining query as follows:

```
SELECT s.time_id, p.prod_subcategory, c.cust_city,
       SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id
AND   s.time_id < TO_DATE('01-OCT-2000','DD-MON-YYYY')
OR s.time_id >= TO_DATE('01-OCT-2001','DD-MON-YYYY'))
GROUP BY time_id, prod_subcategory, cust_city;
```

Note that the freshness of partially stale materialized views is tracked on a per-partition base, and not on a logical base. Because the partitioning strategy of the `sales` fact table is on a quarterly base, changes in December 2000 causes the complete partition `sales_q4_2000` to become stale.

Consider the following query, which asks for sales in quarters 1 and 2 of 2000:

```
SELECT s.time_id, p.prod_subcategory, c.cust_city,
       SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id
AND   s.time_id BETWEEN TO_DATE('01-JAN-2000', 'DD-MON-YYYY')
AND TO_DATE('01-JUL-2000', 'DD-MON-YYYY')
```

```
GROUP BY time_id, prod_subcategory, cust_city;
```

Oracle Database knows that those ranges of rows in the materialized view are fresh and can therefore rewrite the query with the materialized view. The rewritten query looks as follows:

```
SELECT time_id, prod_subcategory, cust_city, sum_amount_sold
FROM sum_sales_per_city_mv
WHERE time_id BETWEEN TO_DATE('01-JAN-2000', 'DD-MON-YYYY')
AND TO_DATE('01-JUL-2000', 'DD-MON-YYYY');
```

Instead of the partitioning key, a partition marker (a function that identifies the partition given a rowid) can be present in the select (and `GROUP BY` list) of the materialized view. You can use the materialized view to rewrite queries that require data from only certain partitions (identifiable by the partition-marker), for instance, queries that have a predicate specifying ranges of the partitioning keys containing entire partitions. See Chapter 9, "Advanced Materialized Views" for details regarding the supplied partition marker function `DBMS_MVIEW.PMARKER`.

The following example illustrates the use of a partition marker in the materialized view instead of directly using the partition key column:

```
CREATE MATERIALIZED VIEW sum_sales_per_city_2_mv
ENABLE QUERY REWRITE AS
SELECT DBMS_MVIEW.PMARKER(s.rowid) AS pmarker,
       t.fiscal_quarter_desc, p.prod_subcategory, c.cust_city,
       SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id
AND   s.time_id = t.time_id
GROUP BY DBMS_MVIEW.PMARKER(s.rowid),
    prod_subcategory, cust_city, fiscal_quarter_desc;
```

Suppose you know that the partition `sales_q1_2000` is fresh and DML changes have taken place for other partitions of the `sales` table. For testing purposes, you can apply an arbitrary DML operation on `sales`, changing a different partition than `sales_q1_2000` when the materialized view is fresh. An example is the following:

```
INSERT INTO SALES VALUES(17, 10, '01-DEC-2000', 4, 380, 123.45, 54321);
```

Although the materialized view `sum_sales_per_city_2_mv` is now considered generically stale, Oracle Database can rewrite the following query using this materialized view. This query restricts the data to the partition `sales_q1_2000`, and selects only certain values of `cust_city`, as shown in the following:

```
SELECT p.prod_subcategory, c.cust_city, SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id AND s.time_id = t.time_id
AND c.cust_city= 'Nuernberg'
AND s.time_id >=TO_DATE('01-JAN-2000','dd-mon-yyyy')
AND s.time_id <  TO_DATE('01-APR-2000','dd-mon-yyyy')
GROUP BY prod_subcategory, cust_city;
```

Note that rewrite with a partially stale materialized view that contains a PMARKER function can only take place when the complete data content of one or more partitions is accessed and the predicate condition is on the partitioned fact table itself, as shown in the earlier example.

The DBMS_MVIEW.PMARKER function gives you exactly one distinct value for each partition. This dramatically reduces the number of rows in a potential materialized view compared to the partitioning key itself, but you are also giving up any detailed information about this key. The only thing you know is the partition number and, therefore, the lower and upper boundary values. This is the trade-off for reducing the cardinality of the range partitioning column and thus the number of rows.

Assuming the value of `p_marker` for partition `sales_q1_2000` is 31070, the previously shown queries can be rewritten against the materialized view as follows:

```
SELECT mv.prod_subcategory, mv.cust_city, SUM(mv.sum_amount_sold)
FROM sum_sales_per_city_2_mv mv
WHERE mv.pmarker = 31070 AND mv.cust_city= 'Nuernberg'
GROUP BY prod_subcategory, cust_city;
```

So the query can be rewritten against the materialized view without accessing stale data.

Query Rewrite Using Nested Materialized Views

Query rewrite attempts to iteratively take advantage of nested materialized views. Oracle Database first tries to rewrite a query with materialized views having aggregates and joins, then with a materialized view containing only joins. If any of the rewrites succeeds, Oracle repeats that process again until no rewrites are found. For example, assume that you had created materialized views `join_sales_time_product_mv` and `sum_sales_time_product_mv` as in the following:

```
CREATE MATERIALIZED VIEW join_sales_time_product_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_id, p.prod_name, t.time_id, t.week_ending_day,
       s.channel_id, s.promo_id, s.cust_id, s.amount_sold
```

```
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id AND s.prod_id = p.prod_id;

CREATE MATERIALIZED VIEW sum_sales_time_product_mv
ENABLE QUERY REWRITE AS
SELECT mv.prod_name, mv.week_ending_day, COUNT(*) cnt_all,
       SUM(mv.amount_sold) sum_amount_sold,
       COUNT(mv.amount_sold) cnt_amount_sold
FROM join_sales_time_product_mv mv
GROUP BY mv.prod_name, mv.week_ending_day;
```

Then consider the following query:

```
SELECT p.prod_name, t.week_ending_day, SUM(s.amount_sold)
FROM sales s, products p, times t
WHERE s.time_id=t.time_id AND s.prod_id=p.prod_id
GROUP BY p.prod_name, t.week_ending_day;
```

Oracle first tries to rewrite it with a materialized aggregate view and finds there is none eligible (note that single-table aggregate materialized view `sum_sales_store_time_mv` cannot yet be used), and then tries a rewrite with a materialized join view and finds that `join_sales_time_product_mv` is eligible for rewrite. The rewritten query has this form:

```
SELECT mv.prod_name, mv.week_ending_day, SUM(mv.amount_sold)
FROM join_sales_time_product_mv mv
GROUP BY mv.prod_name, mv.week_ending_day;
```

Because a rewrite occurred, Oracle tries the process again. This time, the query can be rewritten with single-table aggregate materialized view `sum_sales_store_time` into the following form:

```
SELECT mv.prod_name, mv.week_ending_day, mv.sum_amount_sold
FROM sum_sales_time_product_mv mv;
```

Query Rewrite When Using GROUP BY Extensions

Several extensions to the `GROUP BY` clause in the form of `GROUPING SETS`, `CUBE`, `ROLLUP`, and their concatenation are available. These extensions enable you to selectively specify the groupings of interest in the `GROUP BY` clause of the query. For example, the following is a typical query with grouping sets:

```
SELECT p.prod_subcategory, t.calendar_month_desc, c.cust_city,
  SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, customers c, products p, times t
WHERE s.time_id=t.time_id AND s.prod_id = p.prod_id AND s.cust_id = c.cust_id
```

```
GROUP BY GROUPING SETS ((p.prod_subcategory, t.calendar_month_desc),
   (c.cust_city, p.prod_subcategory));
```

The term **base grouping** for queries with GROUP BY extensions denotes all unique expressions present in the GROUP BY clause. In the previous query, the following grouping (`p.prod_subcategory, t.calendar_month_desc, c.cust_city`) is a base grouping.

The extensions can be present in user queries and in the queries defining materialized views. In both cases, materialized view rewrite applies and you can distinguish rewrite capabilities into the following scenarios:

- Materialized View has Simple GROUP BY and Query has Extended GROUP BY
- Materialized View has Extended GROUP BY and Query has Simple GROUP BY
- Both Materialized View and Query Have Extended GROUP BY

Materialized View has Simple GROUP BY and Query has Extended GROUP BY When a query contains an extended GROUP BY clause, it can be rewritten with a materialized view if its base grouping can be rewritten using the materialized view as listed in the rewrite rules explained in "When Does Oracle Rewrite a Query?" on page 18-4. For example, in the following query:

```
SELECT p.prod_subcategory, t.calendar_month_desc, c.cust_city,
  SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, customers c, products p, times t
WHERE s.time_id=t.time_id AND s.prod_id = p.prod_id AND s.cust_id = c.cust_id
GROUP BY GROUPING SETS
((p.prod_subcategory, t.calendar_month_desc),
   (c.cust_city, p.prod_subcategory));
```

The base grouping is `(p.prod_subcategory, t.calendar_month_desc, c.cust_city, p.prod_subcategory))` and, consequently, Oracle can rewrite the query using `sum_sales_pscat_month_city_mv` as follows:

```
SELECT mv.prod_subcategory, mv.calendar_month_desc, mv.cust_city,
  SUM(mv.sum_amount_sold) AS sum_amount_sold
FROM sum_sales_pscat_month_city_mv mv
GROUP BY GROUPING SETS
((mv.prod_subcategory, mv.calendar_month_desc),
  (mv.cust_city, mv.prod_subcategory));
```

A special situation arises if the query uses the `EXPAND_GSET_TO_UNION` hint. See "Hint for Queries with Extended GROUP BY" on page 18-44 for an example of using `EXPAND_GSET_TO_UNION`.

Materialized View has Extended GROUP BY and Query has Simple GROUP BY In order for a materialized view with an extended GROUP BY to be used for rewrite, it must satisfy two additional conditions:

- It must contain a grouping distinguisher, which is the GROUPING_ID function on all GROUP BY expressions. For example, if the GROUP BY clause of the materialized view is GROUP BY CUBE(a, b), then the SELECT list should contain GROUPING_ID(a, b).
- The GROUP BY clause of the materialized view should not result in any duplicate groupings. For example, GROUP BY GROUPING SETS ((a, b), (a, b)) would disqualify a materialized view from general rewrite.

A materialized view with an extended GROUP BY contains multiple groupings. Oracle finds the grouping with the lowest cost from which the query can be computed and uses that for rewrite. For example, consider the following materialized view:

```
CREATE MATERIALIZED VIEW sum_grouping_set_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_category, p.prod_subcategory, c.cust_state_province, c.cust_city,
  GROUPING_ID(p.prod_category,p.prod_subcategory,
              c.cust_state_province,c.cust_city) AS gid,
  SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c
WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id
GROUP BY GROUPING SETS
((p.prod_category, p.prod_subcategory, c.cust_city),
  (p.prod_category, p.prod_subcategory, c.cust_state_province, c.cust_city),
  (p.prod_category, p.prod_subcategory));
```

In this case, the following query will be rewritten:

```
SELECT p.prod_subcategory, c.cust_city, SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c
WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id
GROUP BY p.prod_subcategory, c.cust_city;
```

This query will be rewritten with the closest matching grouping from the materialized view. That is, the (prodcategory, prod_subcategory, cust_city) grouping:

```
SELECT prod_subcategory, cust_city, SUM(sum_amount_sold) AS sum_amount_sold
FROM sum_grouping_set_mv
WHERE gid = grouping identifier of (prod_category,prod_subcategory, cust_city)
GROUP BY prod_subcategory, cust_city;
```

Both Materialized View and Query Have Extended GROUP BY When both materialized view and the query contain GROUP BY extensions, Oracle uses two strategies for rewrite: grouping match and UNION ALL rewrite. First, Oracle tries grouping match. The groupings in the query are matched against groupings in the materialized view and if all are matched with no rollup, Oracle selects them from the materialized view. For example, consider the following query:

```
SELECT p.prod_category, p.prod_subcategory, c.cust_city,
       SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c
WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id
GROUP BY GROUPING SETS
((p.prod_category, p.prod_subcategory, c.cust_city),
  (p.prod_category, p.prod_subcategory));
```

This query matches two groupings from sum_grouping_set_mv and Oracle rewrites the query as the following:

```
SELECT prod_subcategory, cust_city, sum_amount_sold
FROM sum_grouping_set_mv
WHERE gid = grouping identifier of (prod_category,prod_subcategory, cust_city)
   OR gid = grouping identifier of (prod_category,prod_subcategory)
```

If grouping match fails, Oracle tries a general rewrite mechanism called UNION ALL rewrite. Oracle first represents the query with the extended GROUP BY clause as an equivalent UNION ALL query. Every grouping of the original query is placed in a separate UNION ALL branch. The branch will have a simple GROUP BY clause. For example, consider this query:

```
SELECT p.prod_category, p.prod_subcategory, c.cust_state_province,
  t.calendar_month_desc, SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c, times t
WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id
GROUP BY GROUPING SETS
((p.prod_subcategory, t.calendar_month_desc),
  (t.calendar_month_desc),
  (p.prod_category, p.prod_subcategory, c.cust_state_province),
  (p.prod_category, p.prod_subcategory));
```

This is first represented as UNION ALL with four branches:

```
SELECT null, p.prod_subcategory, null,
    t.calendar_month_desc, SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c, times t
WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id
GROUP BY p.prod_subcategory, t.calendar_month_desc
```

```
UNION ALL
  SELECT null, null, null,
    t.calendar_month_desc, SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c, times t
WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id
GROUP BY t.calendar_month_desc
UNION ALL
SELECT p.prod_category, p.prod_subcategory, c.cust_state_province,
    null, SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c, times t
WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id
GROUP BY p.prod_category, p.prod_subcategory, c.cust_state_province
UNION ALL
  SELECT p.prod_category, p.prod_subcategory, null,
    null, SUM(s.amount_sold) AS sum_amount_sold
  FROM sales s, products p, customers c, times t
  WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id
  GROUP BY p.prod_category, p.prod_subcategory;
```

Each branch is then rewritten separately using the rules from "When Does Oracle Rewrite a Query?" on page 18-4. Using the materialized view `sum_grouping_set_mv`, Oracle can rewrite only branches three (which requires materialized view rollup) and four (which matches the materialized view exactly). The unrewritten branches will be converted back to the extended `GROUP BY` form. Thus, eventually, the query is rewritten as:

```
SELECT null, p.prod_subcategory, null,
    t.calendar_month_desc, SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c, times t
WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id
GROUP BY GROUPING SETS
  ((p.prod_subcategory, t.calendar_month_desc),
    (t.calendar_month_desc),)
UNION ALL
  SELECT prod_category, prod_subcategory, cust_state_province,
    null, SUM(sum_amount_sold) AS sum_amount_sold
  FROM sum_grouping_set_mv
  WHERE gid = <grouping id of (prod_category,prod_subcategory, cust_city)>
  GROUP BY p.prod_category, p.prod_subcategory, c.cust_state_province
UNION ALL
  SELECT prod_category, prod_subcategory, null,
    null, sum_amount_sold
  FROM sum_grouping_set_mv
  WHERE gid = <grouping id of (prod_category,prod_subcategory)>
```

Note that a query with extended GROUP BY is represented as an equivalent UNION ALL and recursively submitted for rewrite optimization. The groupings that cannot be rewritten stay in the last branch of UNION ALL and access the base data instead.

Hint for Queries with Extended GROUP BY

You can use the EXPAND_GSET_TO_UNION hint to force expansion of the query with GROUP BY extensions into the equivalent UNION ALL query. This hint can be used in an environment where materialized views have simple GROUP BY clauses only. In this case, Oracle extends rewrite flexibility as each branch can be independently rewritten by a separate materialized view. See *Oracle Database Performance Tuning Guide* for more information regarding EXPAND_GSET_TO_UNION

Query Rewrite with Inline Views

Oracle Database supports general query rewrite when the user query contains an inline view, or a subquery in the FROM list. Query rewrite matches inline views in the materialized view with inline views in the request query when the text of the two inline views exactly match. In this case, rewrite treats the matching inline view as it would a named view, and general rewrite processing is possible.

The following is an example where the materialized view contains an inline view, and the query has the same inline view, but the aliases for these views are different:

```
CREATE MATERIALIZED VIEW inline_example
ENABLE QUERY REWRITE AS
SELECT t.calendar_month_name, t.calendar_year  p.prod_category,
       SUM(V1.revenue) AS sum_revenue
FROM times t, products p,
     (SELECT time_id, prod_id, amount_sold*0.2 AS revenue FROM sales) V1
WHERE t.time_id = V1.time_id AND p.prod_id = V1.prod_id
GROUP BY calendar_month_name, calendar_year, prod_category;
```

And here is the query that will be rewritten to use the materialized view:

```
SELECT t.calendar_month_name, t.calendar_year, p.prod_category,
       SUM(X1.revenue) AS sum_revenue
FROM times t, products p,
     (SELECT time_id, prod_id, amount_sold*0.2 AS revenue FROM sales) X1
WHERE t.time_id = X1.time_id AND p.prod_id = X1.prod_id
GROUP BY calendar_month_name, calendar_year, prod_category;
```

Query Rewrite with Selfjoins

Oracle accomplishes query rewrite of queries which contain multiple references to the same tables, or self joins by employing two different strategies. Using the first strategy, you need to ensure that the query and the materialized view definitions have the same aliases for the multiple references to a table. If you do not provide a matching alias, Oracle will try the second strategy, where the joins in the query and the materialized view are compared to match the multiple references in the query to the multiple references in the materialized view.

The following is an example of a materialized view and a query. In this example, the query is missing a reference to a column in a table so an exact text match will not work. General query rewrite can occur, however, because the aliases for the table references match.

To demonstrate the self-join rewriting possibility with the sh sample schema, the following addition is assumed to include the actual shipping and payment date in the fact table, referencing the same dimension table times. This is for demonstration purposes only and will not return any results:

```
ALTER TABLE sales ADD (time_id_ship DATE);
ALTER TABLE sales ADD (CONSTRAINT time_id_book_fk FOREIGN key (time_id_ship)
REFERENCES times(time_id) ENABLE NOVALIDATE);
ALTER TABLE sales MODIFY CONSTRAINT time_id_book_fk RELY;
ALTER TABLE sales ADD (time_id_paid DATE);
ALTER TABLE sales ADD (CONSTRAINT time_id_paid_fk FOREIGN KEY (time_id_paid)
REFERENCES times(time_id) ENABLE NOVALIDATE);
ALTER TABLE sales MODIFY CONSTRAINT time_id_paid_fk RELY;
```

Now, you can define a materialized view as follows:

```
CREATE MATERIALIZED VIEW sales_shipping_lag_mv
ENABLE QUERY REWRITE AS
SELECT t1.fiscal_week_number, s.prod_id,
         t2.fiscal_week_number - t1.fiscal_week_number as lag
FROM times t1, sales s, times t2
WHERE t1.time_id = s.time_id AND t2.time_id = s.time_id_ship;
```

The following query fails the exact text match test but is rewritten because the aliases for the table references match:

```
SELECT s.prod_id, t2.fiscal_week_number - t1.fiscal_week_number AS lag
FROM times t1, sales s, times t2
WHERE t1.time_id = s.time_id AND t2.time_id = s.time_id_ship;
```

Note that Oracle performs other checks to ensure the correct match of an instance of a multiply instanced table in the request query with the corresponding table instance in the materialized view. For instance, in the following example, Oracle correctly determines that the matching alias names used for the multiple instances of table `time` does not establish a match between the multiple instances of table `time` in the materialized view.

The following query cannot be rewritten using `sales_shipping_lag_mv`, even though the alias names of the multiply instanced table `time` match because the joins are not compatible between the instances of `time` aliased by `t2`:

```
SELECT s.prod_id, t2.fiscal_week_number - t1.fiscal_week_number AS lag
FROM times t1, sales s, times t2
WHERE t1.time_id = s.time_id AND t2.time_id = s.time_id_paid;
```

This request query joins the instance of the `time` table aliased by `t2` on the `s.time_id_paid` column, while the materialized views joins the instance of the `time` table aliased by `t2` on the `s.time_id_ship` column. Because the join conditions differ, Oracle correctly determines that rewrite cannot occur.

The following query does not have any matching alias in the materialized view, `sales_shipping_lag_mv`, for the table, times. But query rewrite will now compare the joins between the query and the materialized view and correctly match the multiple instances of times.

```
SELECT s.prod_id, x2.fiscal_week_number - x1.fiscal_week_number AS lag
FROM times x1, sales s, times x2
WHERE x1.time_id = s.time_id AND x2.time_id = s.time_id_ship;
```

Query Rewrite and View Constraints

Data warehouse applications recognize multi-dimensional cubes in the database by identifying integrity constraints in the relational schema. Integrity constraints represent primary and foreign key relationships between fact and dimension tables. By querying the data dictionary, applications can recognize integrity constraints and hence the cubes in the database. However, this does not work in an environment where database administrators, for schema complexity or security reasons, define views on fact and dimension tables. In such environments, applications cannot identify the cubes properly. By allowing constraint definitions between views, you can propagate base table constraints to the views, thereby allowing applications to recognize cubes even in a restricted environment.

View constraint definitions are declarative in nature, but operations on views are subject to the integrity constraints defined on the underlying base tables, and constraints on views can be enforced through constraints on base tables. Defining

constraints on base tables is necessary, not only for data correctness and cleanliness, but also for materialized view query rewrite purposes using the original base objects.

Materialized view rewrite extensively uses constraints for query rewrite. They are used for determining lossless joins, which, in turn, determine if joins in the materialized view are compatible with joins in the query and thus if rewrite is possible.

DISABLE NOVALIDATE is the only valid state for a view constraint. However, you can choose RELY or NORELY as the view constraint state to enable more sophisticated query rewrites. For example, a view constraint in the RELY state allows query rewrite to occur when the query integrity level is set to TRUSTED. Table 18–2 illustrates when view constraints are used for determining lossless joins.

Note that view constraints cannot be used for query rewrite integrity level ENFORCED. This level enforces the highest degree of constraint enforcement ENABLE VALIDATE.

Table 18–2 View Constraints and Rewrite Integrity Modes

Constraint States	RELY	NORELY
ENFORCED	No	No
TRUSTED	Yes	No
STALE_TOLERATED	Yes	No

Example 18–10 View Constraints

To demonstrate the rewrite capabilities on views, you need to extend the sh sample schema as follows:

```
CREATE VIEW time_view AS
SELECT time_id, TO_NUMBER(TO_CHAR(time_id, 'ddd')) AS day_in_year FROM times;
```

You can now establish a foreign key / primary key relationship (in RELY ON) mode between the view and the fact table, and thus rewrite will take place as described in Table 18–2, by adding the following constraints. Rewrite will then work for example in TRUSTED mode.

```
ALTER VIEW time_view ADD (CONSTRAINT time_view_pk
   PRIMARY KEY (time_id) DISABLE NOVALIDATE);
ALTER VIEW time_view MODIFY CONSTRAINT time_view_pk RELY;
ALTER TABLE sales ADD (CONSTRAINT time_view_fk FOREIGN KEY (time_id)
   REFERENCES time_view(time_id) DISABLE NOVALIDATE);
```

```
ALTER TABLE sales MODIFY CONSTRAINT time_view_fk RELY;
```

Consider the following materialized view definition:

```
CREATE MATERIALIZED VIEW sales_pcat_cal_day_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_category, t.day_in_year, SUM(s.amount_sold) AS sum_amount_sold
FROM time_view t, sales s, products p
WHERE t.time_id = s.time_id AND p.prod_id = s.prod_id
GROUP BY p.prod_category, t.day_in_year;
```

The following query, omitting the dimension table `products`, will also be rewritten without the primary key / foreign key relationships, because the suppressed join between `sales` and `products` is known to be lossless.

```
SELECT t.day_in_year, SUM(s.amount_sold) AS sum_amount_sold
FROM time_view t, sales s WHERE t.time_id = s.time_id
GROUP BY t.day_in_year;
```

However, if the materialized view `sales_pcat_cal_day_mv` were defined only in terms of the view `time_view`, then you could not rewrite the following query, suppressing then join between `sales` and `time_view`, because there is no basis for losslessness of the delta materialized view join. With the additional constraints as shown previously, this query will also rewrite.

```
SELECT p.prod_category, SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p WHERE p.prod_id = s.prod_id
GROUP BY p.prod_category;
```

To undo the changes you have made to the `sh` schema, issue the following statements:

```
ALTER TABLE sales DROP CONSTRAINT time_view_fk;
DROP VIEW time_view;
```

View Constraints Restrictions If the referential constraint definition involves a view, that is, either the foreign key or the referenced key resides in a view, the constraint can only be in `DISABLE NOVALIDATE` mode.

A `RELY` constraint on a view is allowed only if the referenced `UNIQUE` or `PRIMARY KEY` constraint in `DISABLE NOVALIDATE` mode is also a `RELY` constraint.

The specification of `ON DELETE` actions associated with a referential Integrity constraint, is not allowed (for example, `DELETE` cascade). However, `DELETE`, `UPDATE`, and `INSERT` operations are allowed on views and their base tables as view constraints are in `DISABLE NOVALIDATE` mode.

Query Rewrite and Expression Matching

An expression that appears in a query can be replaced with a simple column in a materialized view provided the materialized view column represents a precomputed expression that matches with the expression in the query. If a query can be rewritten to use a materialized view, it will be faster. This is because materialized views contain precomputed calculations and do not need to perform expression computation.

The expression matching is done by first converting the expressions into canonical forms and then comparing them for equality. Therefore, two different expressions will generally be matched as long as they are equivalent to each other. Further, if the entire expression in a query fails to match with an expression in a materialized view, then subexpressions of it are tried to find a match. The subexpressions are tried in a top-down order to get maximal expression matching.

Consider a query that asks for sum of sales by age brackets (1-10, 11-20, 21-30, and so on).

```
CREATE MATERIALIZED VIEW sales_by_age_bracket_mv
ENABLE QUERY REWRITE AS
SELECT TO_CHAR((2000-c.cust_year_of_birth)/10-0.5,999) AS age_bracket,
       SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, customers c WHERE s.cust_id=c.cust_id
GROUP BY TO_CHAR((2000-c.cust_year_of_birth)/10-0.5,999);
```

The following query rewrites, using expression matching:

```
SELECT TO_CHAR(((2000-c.cust_year_of_birth)/10)-0.5,999), SUM(s.amount_sold)
FROM sales s, customers c WHERE s.cust_id=c.cust_id
GROUP BY TO_CHAR((2000-c.cust_year_of_birth)/10-0.5,999);
```

This query is rewritten in terms of `sum_sales_mv` based on the matching of the canonical forms of the age bracket expressions (that is, 2000 - `c.cust_year_of_birth`)/10-0.5), as follows:

```
SELECT age_bracket, sum_amount_sold FROM sales_by_age_bracket_mv;
```

Date Folding Rewrite

Date folding rewrite is a specific form of expression matching rewrite. In this type of rewrite, a date range in a query is folded into an equivalent date range representing higher date granules. The resulting expressions representing higher date granules in the folded date range are matched with equivalent expressions in a materialized view. The folding of date range into higher date granules such as months, quarters, or years is done when the underlying datatype of the column is

an Oracle DATE. The expression matching is done based on the use of canonical forms for the expressions.

DATE is a built-in datatype which represents ordered time units such as seconds, days, and months, and incorporates a time hierarchy (second -> minute -> hour -> day -> month -> quarter -> year). This hard-coded knowledge about DATE is used in folding date ranges from lower-date granules to higher-date granules. Specifically, folding a date value to the beginning of a month, quarter, year, or to the end of a month, quarter, year is supported. For example, the date value `1-jan-1999` can be folded into the beginning of either year `1999` or quarter `1999-1` or month `1999-01`. And, the date value `30-sep-1999` can be folded into the end of either quarter `1999-03` or month `1999-09`.

Note: Due to the way date folding works, you should be careful when using BETWEEN and date columns. The best way to use BETWEEN and date columns is to increment the later date by 1. In other words, instead of using `date_col BETWEEN '1-jan-1999' AND '30-jun-1999'`, you should use `date_col BETWEEN '1-jan-1999' AND '1-jul-1999'`. You could also use the TRUNC function to get the equivalent result, as in `TRUNC(date_col) BETWEEN '1-jan-1999' AND '30-jun-1999'`. TRUNC will, however, strip time values.

Because date values are ordered, any range predicate specified on date columns can be folded from lower level granules into higher level granules provided the date range represents an integral number of higher level granules. For example, the range predicate `date_col >= '1-jan-1999' AND date_col < '30-jun-1999'` can be folded into either a month range or a quarter range using the TO_CHAR function, which extracts specific date components from a date value.

The advantage of aggregating data by folded date values is the compression of data achieved. Without date folding, the data is aggregated at the lowest granularity level, resulting in increased disk space for storage and increased I/O to scan the materialized view.

Consider a query that asks for the sum of sales by product types for the years 1998.

```
SELECT p.prod_category, SUM(s.amount_sold)
FROM sales s, products p
WHERE s.prod_id=p.prod_id AND s.time_id >= TO_DATE('01-jan-1998', 'dd-mon-yyyy')
  AND s.time_id <   TO_DATE('01-jan-1999', 'dd-mon-yyyy')
GROUP BY p.prod_category;
```

```
CREATE MATERIALIZED VIEW sum_sales_pcat_monthly_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_category, TO_CHAR(s.time_id,'YYYY-MM') AS month,
       SUM(s.amount_sold) AS sum_amount
FROM sales s, products p
WHERE s.prod_id=p.prod_id
GROUP BY p.prod_category, TO_CHAR(s.time_id, 'YYYY-MM');

SELECT p.prod_category, SUM(s.amount_sold)
FROM sales s, products p
WHERE s.prod_id=p.prod_id
AND TO_CHAR(s.time_id, 'YYYY-MM') >= '01-jan-1998'
AND TO_CHAR(s.time_id, 'YYYY-MM') < '01-jan-1999'
GROUP BY p.prod_category;

SELECT mv.prod_category, mv.sum_amount
FROM sum_sales_pcat_monthly_mv mv
WHERE month >= '01-jan-1998' AND month < '01-jan-1999';
```

The range specified in the query represents an integral number of years, quarters, or months. Assume that there is a materialized view mv3 that contains pre-summarized sales by prod_type and is defined as follows:

```
CREATE MATERIALIZED VIEW mv3
ENABLE QUERY REWRITE AS
SELECT prod_type, TO_CHAR(sale_date,'yyyy-mm') AS month, SUM(sales) AS sum_sales
FROM sales, products WHERE sales.prod_id = products.prod_id
GROUP BY prod_type, TO_CHAR(sale_date, 'yyyy-mm');
```

The query can be rewritten by first folding the date range into the month range and then matching the expressions representing the months with the month expression in mv3. This rewrite is shown in two steps (first folding the date range followed by the actual rewrite).

```
SELECT prod_type, SUM(sales) AS sum_sales
FROM sales, products
WHERE sales.prod_id = products.prod_id AND TO_CHAR(sale_date, 'yyyy-mm') >=
      TO_CHAR('01-jan-1998', 'yyyy-mm') AND TO_CHAR('01-jan-1999', 'yyyy-mm')
GROUP BY prod_type;

SELECT prod_type, sum_sales
FROM mv3 WHERE month >=
      TO_CHAR('01-jan-1998', 'yyyy-mm')
  AND month < TO_CHAR('01-jan-1999', 'yyyy-mm');
```

```
GROUP BY prod_type;
```

If mv3 had pre-summarized sales by prod_type and year instead of prod_type and month, the query could still be rewritten by folding the date range into year range and then matching the year expressions.

Partition Change Tracking (PCT) Rewrite

PCT rewrite enables the optimizer to accurately rewrite queries with fresh data using materialized views that are only partially fresh. To do so, Oracle keeps track of which partitions in the detail tables have been updated. Oracle then tracks which rows in the materialized view originate from the affected partitions in the detail tables. The optimizer is then able to use those portions of the materialized view that are known to be fresh.

The optimizer uses PCT rewrite in QUERY_REWRITE_INTEGRITY = ENFORCED and TRUSTED modes. The optimizer does not use PCT rewrite in STALE_TOLERATED mode because data freshness is not considered in that mode.

You can use PCT rewrite with partitioning, but hash partitioning is not supported. The following sections discuss aspects of using PCT:

- PCT Rewrite Based on LIST Partitioned Tables
- PCT and PMARKER
- PCT Rewrite with Materialized Views Based on Range-List Partitioned Tables
- PCT Rewrite Using Rowid as Pmarker

PCT Rewrite Based on LIST Partitioned Tables

If the LIST partitioning key is present in the materialized view's SELECT and GROUP BY, then PCT will be supported by the materialized view. Regardless of the supported partitioning type, if the partition marker or rowid of the detail table is present in the materialized view then PCT is supported by the materialized view on that specific detail table.

```
CREATE TABLE sales_par_list
(calendar_year, calendar_month_number, day_number_in_month,
 country_name, prod_id, quantity_sold, amount_sold)
 PARTITION BY LIST (country_name)
 (PARTITION America
      VALUES ('United States of America', 'Argentina'),
  PARTITION Asia
      VALUES ('Japan', 'India'),
```

```
  PARTITION Europe
      VALUES ('France', 'Spain', 'Ireland'))
  AS SELECT t.calendar_year, t.calendar_month_number,
         t.day_number_in_month, c1.country_name, s.prod_id,
         s.quantity_sold, s.amount_sold
  FROM times t, countries c1, sales s, customers c2
  WHERE s.time_id = t.time_id and s.cust_id = c2.cust_id and
        c2.country_id = c1.country_id and
        c1.country_name IN ('United States of America', 'Argentina',
       'Japan', 'India', 'France', 'Spain', 'Ireland');
```

If a materialized view is created on the table sales_par_list, which has a list partitioning key, PCT rewrite will use that materialized view for potential rewrites.

To illustrate this feature, the following example creates a materialized view that has the total amounts sold of every product in each country for each year. The view depends on detail tables sales_par_list and product.

```
CREATE MATERIALIZED VIEW sales_per_country_mv
BUILD IMMEDIATE
REFRESH FORCE ON DEMAND
ENABLE QUERY REWRITE AS
SELECT s.calendar_year calendar_year, s.country_name country_name,
 p.prod_name prod_name, SUM(s.amount_sold) AS sum_sales, COUNT(*) AS cnt
FROM sales_par_list s, products p
WHERE s.prod_id = p.prod_id AND s.calendar_year <= 2000
GROUP BY s.calendar_year, s.country_name, prod_name;
```

sales_per_country_mv supports PCT against sales_par_list as its list partition key country_name is in its SELECT and GROUP BY list. Table products is not partitioned, so sales_per_country_mv does not support PCT against this table.

A query could be rewritten (in ENFORCED or TRUSTED modes) in terms of sales_per_country_mv even if sales_per_country_mv is stale if the incoming query accesses only fresh parts of the materialized view. You can determine which parts of the materialized view are FRESH only if the updated tables are PCT enabled in the materialized view; if non-PCT enabled tables have been updated then the rewrite is not possible with fresh data from that specific materialized view as you cannot identify the FRESH portions of the materialized view.

sales_per_country_mv supports PCT on sales_par_list and does not support PCT on table product. If table product is updated, then PCT rewrite is not possible with sales_per_country_mv as you cannot tell which portions of the materialized view are FRESH.

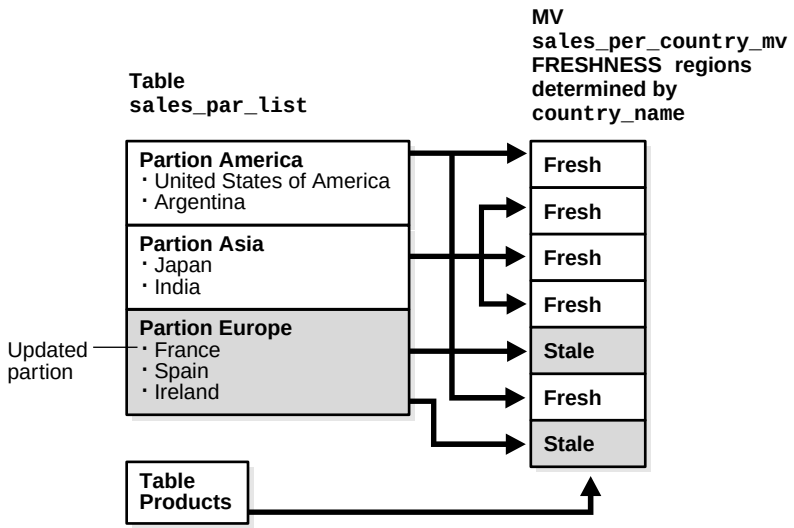
The following updates sales_par_list as follows:

```
INSERT INTO sales_par_list VALUES (2000, 10, 22, 'France', 900, 20, 200.99);
```

This statement inserted a row into partition Europe in table sales_par_list. Now sales_per_country_mv is stale, but PCT rewrite (in ENFORCED and TRUSTED modes) is possible as this materialized view supports PCT against table sales_par_list. The fresh and stale areas of the materialized view are identified based on the partitioned detail table sales_par_list.

Figure 18–3 illustrates what is fresh and what is stale in this example.

Figure 18–3 PCT Rewrite and List Partitioning



Consider the following query:

```
SELECT s.country_name, p.prod_name, SUM(s.amount_sold) AS sum_sales,
   COUNT(*) AS cnt
FROM sales_par_list s, products p
WHERE s.prod_id = p.prod_id AND s.calendar_year = 2000 AND
   s.country_name IN ('United States of America', 'Japan')
GROUP BY s.country_name, p.prod_name;
```

This query accesses partitions America and Asia in sales_par_list; these partition have not been updated so rewrite is possible with stale materialized view

sales_per_country_mv as this query will access only FRESH portions of the materialized view.

The query is rewritten in terms of sales_per_country_mv as follows:

```
SELECT country_name, prod_name, SUM(sum_sales) AS sum_slaes, SUM(cnt) AS cnt
FROM sales_per_country_mv WHERE calendar_year = 2000
  AND country_name IN ('United States of America', 'Japan')
GROUP BY country_name, prod_name;
```

Now consider the following query:

```
SELECT s.country_name,  p.prod_name,
 SUM(s.amount_sold) AS sum_sales, COUNT(*) AS cnt
FROM sales_par_list s, products p
WHERE  s.prod_id = p.prod_id AND s.calendar_year = 1999 AND
       s.country_name IN ('Japan', 'India', 'Spain')
GROUP BY s.country_name, p.prod_name;
```

This query accesses partitions Europe and Asia in sales_par_list; partition Europe has been updated so this query cannot be rewritten in terms of sales_per_country_mv as the required data from the materialized view is stale.

You will be able to rewrite after any kinds of updates to sales_par_list, that is DMLs, direct loads and Partition Maintenance Operations (PMOPs) if the incoming query accesses FRESH parts of the materialized view.

PCT and PMARKER

When a partition marker is provided, the query rewrite capabilities are limited to rewrite queries that access whole detail table partitions as all rows from a specific partition have the same pmarker value. That is, if a query accesses a portion of a detail table partition, it is not rewritten even if that data corresponds to a FRESH portion of the materialized view. Now FRESH portions of the materialized view are determined by the pmarker value. To determine which rows of the materialized view are fresh, you associate freshness with the marker value, so all rows in the materialized view with a specific pmarker value are FRESH or are STALE.

The following creates a materialized view has the total amounts sold of every product in each detail table partition of sales_par_list for each year. This materialized view will also depend on detail table products as shown in the following:

```
CREATE MATERIALIZED VIEW sales_per_dt_partition_mv
BUILD IMMEDIATE
REFRESH FORCE ON DEMAND
```

```
ENABLE QUERY REWRITE AS
SELECT s.calendar_year calendar_year, p.prod_name prod_name,
       DBMS_MVIEW.PMARKER(s.rowid) pmarker,
       SUM(s.amount_sold) AS sum_sales, COUNT(*) AS cnt
FROM sales_par_list s, products p
WHERE s.prod_id = p.prod_id AND s.calendar_year > 2000
GROUP BY s.calendar_year, DBMS_MVIEW.PMARKER(s.rowid), p.prod_name;
```

The materialized view sales_per_dt_partition_mv provides the sum of sales for each detail table partition. This materialized view supports PCT rewrite against table sales_par_list because the partition marker is in its SELECT and GROUP BY clauses. Table 18–3 lists the partition names and their pmarkers for this example.

Table 18–3 Partition Names and Their Pmarkers

Partition Name	Pmarker
America	1000
Asia	1001
Europe	1002

Then update the table sales_par_list as follows:

```
DELETE FROM sales_par_list WHERE country_name = 'India';
```

You have deleted rows from partition Asia in table sales_par_list. Now sales_per_dt_partition_mv is stale, but PCT rewrite (in ENFORCED and TRUSTED modes) is possible as this materialized view supports PCT (pmarker based) against table sales_par_list.

Now consider the following query:

```
SELECT p.prod_name, SUM(s.amount_sold) AS sum_sales, COUNT(*) AS cnt
FROM sales_par_list s, products p
WHERE  s.prod_id = p.prod_id AND s.calendar_year = 2001 AND
       s.country_name IN ('United States of America', 'Argentina')
GROUP BY p.prod_name;
```

This query can be rewritten in terms of sales_per_dt_partition_mv as all the data corresponding to a detail table partition is accessed, and the materialized view is FRESH with respect to this data. This query accesses all data in partition America, which has not been updated.

The query is rewritten in terms of sales_per_dt_partition_mv as follows:

```
SELECT prod_name, SUM(sum_sales) AS sum_sales, SUM(cnt) AS cnt
FROM sales_per_dt_partition_mv
WHERE calendar_year = 2001 AND pmarker = 1000
GROUP BY prod_name;
```

PCT Rewrite with Materialized Views Based on Range-List Partitioned Tables

If the detail table is range-list partitioned, a materialized view that depends on this detail table can support PCT at both the partitioning and subpartitioning levels. If both the partition and subpartition keys are present in the materialized view, PCT can be done at a finer granularity; materialized view refreshes can be done to smaller portions of the materialized view and more queries could be rewritten with a stale materialized view. Alternatively, if only the partition key is present in the materialized view, PCT can be done with courser granularity.

Consider the following range-list partitioned table:

```
CREATE TABLE sales_par_range_list
 (calendar_year, calendar_month_number, day_number_in_month,
  country_name, prod_id, prod_name, quantity_sold, amount_sold)
PARTITION BY RANGE (calendar_month_number)
SUBPARTITION BY LIST (country_name)
 (PARTITION q1 VALUES LESS THAN (4)
 (SUBPARTITION q1_America VALUES
 ('United States of America', 'Argentina'),
   SUBPARTITION q1_Asia VALUES ('Japan', 'India'),
   SUBPARTITION q1_Europe VALUES ('France', 'Spain', 'Ireland')),
   PARTITION q2 VALUES LESS THAN (7)
  (SUBPARTITION q2_America VALUES
   ('United States of America', 'Argentina'),
   SUBPARTITION q2_Asia VALUES ('Japan', 'India'),
   SUBPARTITION q2_Europe VALUES ('France', 'Spain', 'Ireland')),
     PARTITION q3 VALUES LESS THAN (10)
  (SUBPARTITION q3_America VALUES
   ('United States of America', 'Argentina'),
   SUBPARTITION q3_Asia VALUES ('Japan', 'India'),
   SUBPARTITION q3_Europe VALUES ('France', 'Spain', 'Ireland')),
     PARTITION q4 VALUES LESS THAN (13)
  (SUBPARTITION q4_America VALUES
   ('United States of America', 'Argentina'),
   SUBPARTITION q4_Asia VALUES ('Japan', 'India'),
   SUBPARTITION q4_Europe VALUES ('France', 'Spain', 'Ireland')))
  AS SELECT t.calendar_year, t.calendar_month_number,
     t.day_number_in_month, c1.country_name, s.prod_id,
     p.prod_name, s.quantity_sold, s.amount_sold
```

```
  FROM times t, countries c1, products p, sales s, customers c2
  WHERE s.time_id = t.time_id AND s.prod_id = p.prod_id AND
        s.cust_id = c2.cust_id AND c2.country_id = c1.country_id AND
        c1.country_name IN ('United States of America', 'Argentina',
          'Japan', 'India', 'France', 'Spain', 'Ireland');
```

Let us consider the following materialized view `sum_sales_per_year_month_mv`, which has the total amount of products sold each month of each year:

```
CREATE MATERIALIZED VIEW  sum_sales_per_year_month_mv
BUILD IMMEDIATE
REFRESH FORCE ON DEMAND
ENABLE QUERY REWRITE AS
SELECT s.calendar_year, s.calendar_month_number,
        SUM(s.amount_sold) AS sum_sales, COUNT(*) AS cnt
FROM sales_par_range_list s WHERE s.calendar_year > 1990
GROUP BY s.calendar_year, s.calendar_month_number;
```

`sales_per_country_mv` supports PCT against `sales_par_range_list` at the range partitioning level as its range partition key `calendar_month_number` is in its `SELECT` and `GROUP BY` list:

```
INSERT INTO sales_par_range_list
   VALUES (2001, 3, 25, 'Spain', 20, 'PROD20',  300,  20.50);
```

This statement inserts a row with `calendar_month_number = 3` and `country_name = 'Spain'`. This row is inserted into partition `q1` subpartition `Europe`. After this `INSERT` statement, `sum_sales_per_year_month_mv` is stale with respect to partition q1 of `sales_par_range_list`. So any incoming query that accesses data from this partition in `sales_par_range_list` cannot be rewritten, for example, the following statement:

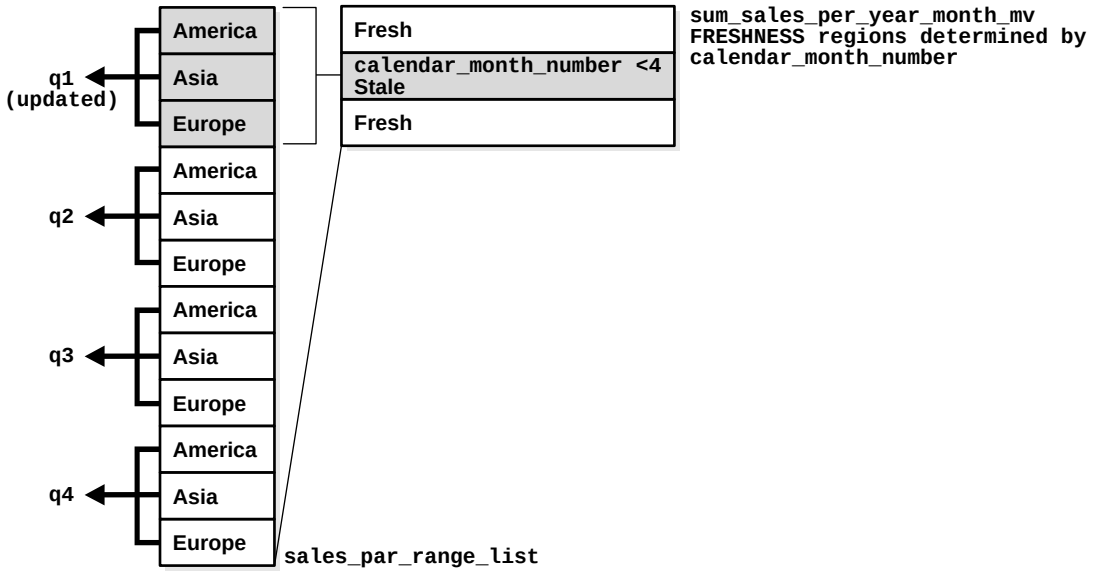
```
SELECT s.calendar_year, SUM(s.amount_sold) AS sum_sales, COUNT(*) AS cnt
FROM sales_par_range_list s
WHERE s.calendar_year = 2000 AND s.calendar_month_number BETWEEN 5 AND 9
GROUP BY s.calendar_year;
```

An example of a statement that does rewrite after the INSERT statement is the following, because it accesses fresh material:

```
SELECT s.calendar_year, SUM(s.amount_sold) AS sum_sales, COUNT(*) AS cnt
FROM sales_par_range_list s
WHERE s.calendar_year = 2000 AND s.calendar_month_number BETWEEN 2 AND 6
GROUP BY s.calendar_year;
```

Figure 18–4 offers a graphical illustration of what is stale and what is fresh.

Figure 18–4 PCT Rewrite and Range-List Partitioning



PCT Rewrite Using Rowid as Pmarker

A materialized view supports PCT rewrite provided a partition key or a partition marker is provided in its SELECT and GROUP BY clause, if there is a GROUP BY clause. In Oracle Database 10*g*, you can use the rowids of the partitioned table instead of the pmarker or the partition key. Note that Oracle converts the rowids into pmarkers internally. Consider the following table:

```
CREATE TABLE product_par_list
(prod_id, prod_name, prod_category,
 prod_subcategory, prod_list_price)
 PARTITION BY LIST (prod_category)
 (PARTITION prod_cat1
      VALUES ('Boys', 'Men'),
  PARTITION prod_cat2
      VALUES ('Girls', 'Women'))
 AS
   SELECT prod_id, prod_name, prod_category, prod_subcategory, prod_list_price
   FROM products;
```

Let us create the following materialized view on tables, `sales_par_list` and `product_par_list`:

```
CREATE MATERIALIZED VIEW  sum_sales_per_category_mv
BUILD IMMEDIATE
REFRESH FORCE ON DEMAND
ENABLE QUERY REWRITE AS
SELECT p.rowid prid, p.prod_category,
       SUM (s.amount_sold) sum_sales, COUNT(*) cnt
FROM sales_par_list s, product_par_list p
WHERE s.prod_id = p.prod_id and s.calendar_year <= 2000
GROUP BY p.rowid, p.prod_category;
```

All the limitations that apply to pmarker rewrite will apply here as well. The incoming query should access a whole partition for the query to be rewritten. The following pmarker table used in this case:

```
product_par_list       pmarker value
---------------        -------------
prod_cat1                     1000
prod_cat2                     1001
prod_cat3                     1002
```

Update table `product_par_list` as follows:

```
DELETE FROM product_par_list WHERE prod_name = 'MEN';
```

So `sum_sales_per_category_mv` is stale with respect to partition `prod_list1` from `product_par_list`.

Now consider the following query:

```
SELECT p.prod_category, SUM(s.amount_sold) AS sum_sales, COUNT(*) AS cnt
FROM sales_par_list s, product_par_list p
WHERE  s.prod_id = p.prod_id AND p.prod_category IN
       ('Girls', 'Women') AND s.calendar_year <= 2000
GROUP BY p.prod_category;
```

This query can be rewritten in terms of `sum_sales_per_category_mv` as all the data corresponding to a detail table partition is accessed, and the materialized view is FRESH with respect to this data. This query accesses all data in partition `prod_cat2`, which has not been updated. Following is the rewritten query in terms of `sum_sales_per_category_mv`:

```
SELECT prod_category, sum_sales, cnt
FROM sum_sales_per_category_mv WHERE dbms_mview.pmarker(srid) IN (1000)
GROUP BY prod_category;
```

Query Rewrite and Bind Variables

Query rewrite is supported when the query contains user bind variables as long as the actual bind values are not required during query rewrite. If the actual values of the bind variables are required during query rewrite, then we say that query rewrite is dependent on the bind values. Because the user bind variables are not available during query rewrite time, if query rewrite is dependent on the bind values, it is not possible to rewrite the query.

For example, consider the following materialized view, `customer_mv`, which has the predicate, (`customer_id >= 1000`), in the `WHERE` clause:

```
CREATE MATERIALIZED VIEW UPPER_CUSTOMER_MV
ENABLE QUERY REWRITE AS
SELECT CUST_ID, PROD_ID,  SUM(AMOUNT_SOLD) AS TOTAL_AMOUNT
FROM SALES WHERE CUST_ID >= 1000
GROUP BY CUST_ID, PROD_ID;
```

Consider the following query, which has a user bind variable, `:user_id`, in its `WHERE` clause:

```
SELECT CUST_ID, PROD_ID, SUM(AMOUNT_SOLD) AS SUM_AMOUNT
FROM SALES WHERE CUST_ID > :user_id
GROUP BY CUST_ID, PROD_ID;
```

Because the materialized view, `customer_mv`, has a selection in its `WHERE` clause, query rewrite is dependent on the actual value of the user bind variable, `user_id`, to compute the containment. Because `user_id` is not available during query rewrite time and query rewrite is dependent on the bind value of `user_id`, this query cannot be rewritten.

Even though the preceding example has a user bind variable in the `WHERE` clause, the same is true regardless of where the user bind variable appears in the query. In other words, irrespective of where a user bind variable appears in a query, if query rewrite is dependent on its value, then the query cannot be rewritten.

Now consider the following query which has a user bind variable, `:user_id`, in its `SELECT` list:

```
SELECT CUST_ID + :user_id, PROD_ID, SUM(AMOUNT_SOLD) AS TOTAL_AMOUNT
FROM SALES WHERE CUST_ID >= 2000
GROUP BY CUST_ID, PROD_ID;
```

Because the value of the user bind variable, `user_id`, is not required during query rewrite time, the preceding query will rewrite.

Query Rewrite Using Set Operator Materialized Views

You can use query rewrite with materialized views that contain set operators. In this case, the query and materialized view do not have to match textually for rewrite to occur. As an example, consider the following materialized view, which gets the postal codes for male customers from San Francisco or Los Angeles:

```
CREATE MATERIALIZED VIEW cust_male_postal_mv
ENABLE QUERY REWRITE AS
SELECT c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_gender = 'M' AND c.cust_city = 'San Francisco'
UNION ALL
SELECT c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_gender = 'M' AND c.cust_city = 'Los Angeles';
```

If you have the following query, which displays the postal codes for male customers from San Francisco or Los Angeles:

```
SELECT c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_city = 'Los Angeles' AND c.cust_gender = 'M'
UNION ALL
SELECT c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_city = 'San Francisco' AND c.cust_gender = '';
```

The rewritten query will be the following:

```
SELECT mv.cust_city, mv.cust_postal_code
FROM cust_male_postal_mv mv;
```

The rewritten query has dropped the UNION ALL and replaced it with the materialized view. Normally, query rewrite has to use the existing set of general eligibility rules to determine if the SELECT subselections under the UNION ALL are equivalent in the query and the materialized view.

If, for example, you have a query that retrieves the postal codes for male customers from San Francisco, Palmdale, or Los Angeles, the same rewrite can occur as in the previous example but query rewrite must keep the UNION ALL with the base tables, as in the following:

```
SELECT c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_city= 'Palmdale' AND c.cust_gender ='M'
```

```
UNION ALL
SELECT c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_city = 'Los Angeles' AND c.cust_gender = 'M'
UNION ALL
SELECT c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_city = 'San Francisco' AND c.cust_gender = 'M';
```

The rewritten query will be:

```
SELECT mv.cust_city, mv.cust_postal_code
FROM cust_male_postal_mv mv
UNION ALL
SELECT c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_city = 'Palmdale' AND c.cust_gender = 'M';
```

So query rewrite will detect the case where a subset of the UNION ALL can be rewritten using the materialized view cust_male_postal_mv.

UNION, UNION ALL, and INTERSECT are commutative, so query rewrite can rewrite regardless of the order the subselects are found in the query or materialized view. However, MINUS is not commutative. A MINUS B is not equivalent to B MINUS A. Therefore, the order in which the subselects appear under the MINUS operator in the query and the materialized view must be in the same order for rewrite to happen. As an example, consider the case where there exists an old version of the customer table called customer_old and you want to find the difference between the old one and the current customer table only for male customers who live in London. That is, you want to find those customers in the current one that were not in the old one. The following example shows how this is done using a MINUS:

```
SELECT c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_city= 'Los Angeles' AND c.cust_gender = 'M'
MINUS
SELECT c.cust_city, c.cust_postal_code
FROM customers_old c
WHERE c.cust_city = 'Los Angeles' AND c.cust_gender = 'M';
```

Switching the subselects would yield a different answer. This illustrates that MINUS is not commutative.

UNION ALL Marker

If a materialized view contains one or more UNION ALL operators, it can also include a UNION ALL marker. The UNION ALL marker is used to identify from which UNION ALL subselect each row in the materialized view originates. Query rewrite can use the marker to distinguish what rows coming from the materialized view belong to a certain UNION ALL subselect. This is useful if the query needs only a subset of the data from the materialized view or if the subselects of the query do not textually match with the subselects of the materialized view. As an example, the following query retrieves the postal codes for male customers from San Francisco and female customers from Los Angeles:

```
SELECT c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_gender = 'M' and c.cust_city = 'San Francisco'
UNION ALL
SELECT c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_gender = 'F' and c.cust_city = 'Los Angeles';
```

The query can be answered using the following materialized view:

```
CREATE MATERIALIZED VIEW cust_postal_mv
ENABLE QUERY REWRITE AS
SELECT 1 AS marker, c.cust_gender, c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_city = 'Los Angeles'
UNION ALL
SELECT 2 AS marker, c.cust_gender, c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_city = 'San Francisco';
```

The rewritten query is as follows:

```
SELECT mv.cust_city, mv.cust_postal_code
FROM cust_postal_mv mv
WHERE mv.marker = 2 AND mv.cust_gender = 'M'
UNION ALL
SELECT mv.cust_city, mv.cust_postal_code
FROM cust_postal_mv mv
WHERE mv.marker = 1 AND mv.cust_gender = 'F';
```

The WHERE clause of the first subselect includes `mv.marker = 2` and `mv.cust_gender = 'M'`, which selects only the rows that represent male customers in the second subselect of the UNION ALL. The WHERE clause of the second subselect includes `mv.marker = 1` and `mv.cust_gender = 'F'`, which selects only those

rows that represent female customers in the first subselect of the UNION ALL. Note that query rewrite cannot take advantage of set operators that drop duplicate or distinct rows. For example, UNION drops duplicates so query rewrite cannot tell what rows have been dropped.

The rules for using a marker are that it must:

- Be a constant number or string and be the same datatype for all UNION ALL subselects.
- Yield a constant, distinct value for each UNION ALL subselect. You cannot reuse the same value in more than one subselect.
- Be in the same ordinal position for all subselects.

Did Query Rewrite Occur?

Because query rewrite occurs transparently, special steps have to be taken to verify that a query has been rewritten. Of course, if the query runs faster, this should indicate that rewrite has occurred, but that is not proof. Therefore, to confirm that query rewrite does occur, use the EXPLAIN PLAN statement or the DBMS_MVIEW.EXPLAIN_REWRITE procedure.

Explain Plan

The EXPLAIN PLAN facility is used as described in *Oracle Database SQL Reference*. For query rewrite, all you need to check is that the object_name column in PLAN_TABLE contains the materialized view name. If it does, then query rewrite has occurred when this query is executed. An example is the following, which creates the materialized view cal_month_sales_mv:

```
CREATE MATERIALIZED VIEW cal_month_sales_mv
ENABLE QUERY REWRITE AS
SELECT  t.calendar_month_desc, SUM(s.amount_sold) AS dollars
FROM sales s, times t WHERE s.time_id = t.time_id
GROUP BY t.calendar_month_desc;
```

If EXPLAIN PLAN is used on the following SQL statement, the results are placed in the default table PLAN_TABLE. However, PLAN_TABLE must first be created using the utlxplan.sql script. Note that EXPLAIN PLAN does not actually execute the query.

```
EXPLAIN PLAN FOR
SELECT  t.calendar_month_desc, SUM(s.amount_sold)
```

```
FROM  sales s, times t WHERE s.time_id = t.time_id
GROUP BY t.calendar_month_desc;
```

For the purposes of query rewrite, the only information of interest from PLAN_TABLE is the OBJECT_NAME, which identifies the objects that will be used to execute this query. Therefore, you would expect to see the object name calendar_month_sales_mv in the output as illustrated in the following:

```
SELECT OPERATION, OBJECT_NAME FROM PLAN_TABLE;

OPERATION                 OBJECT_NAME
--------------------      -----------
SELECT STATEMENT
MAT_VIEW REWRITE ACCESS   CALENDAR_MONTH_SALES_MV
```

DBMS_MVIEW.EXPLAIN_REWRITE Procedure

It can be difficult to understand why a query did not rewrite. The rules governing query rewrite eligibility are quite complex, involving various factors such as constraints, dimensions, query rewrite integrity modes, freshness of the materialized views, and the types of queries themselves. In addition, you may want to know why query rewrite chose a particular materialized view instead of another. To help with this matter, Oracle provides the DBMS_MVIEW.EXPLAIN_REWRITE procedure to advise you when a query can be rewritten and, if not, why not. Using the results from DBMS_MVIEW.EXPLAIN_REWRITE, you can take the appropriate action needed to make a query rewrite if at all possible.

Note that the query specified in the EXPLAIN_REWRITE statement is never actually executed.

DBMS_MVIEW.EXPLAIN_REWRITE Syntax

You can obtain the output from DBMS_MVIEW.EXPLAIN_REWRITE in two ways. The first is to use a table, while the second is to create a varray. The following shows the basic syntax for using an output table:

```
DBMS_MVIEW.EXPLAIN_REWRITE (
    query           VARCHAR2,
    mv              VARCHAR2(30),
    statement_id    VARCHAR2(30));
```

You can create an output table called REWRITE_TABLE by executing the utlxrw.sql script.

The query parameter is a text string representing the SQL query. The parameter, mv, is a fully qualified materialized view name in the form of schema.mv. This is an optional parameter. When it is not specified, EXPLAIN_REWRITE returns any relevant messages regarding all the materialized views considered for rewriting the given query. When schema is omitted and only mv is specified, EXPLAIN_REWRITE looks for the materialized view in the current schema.

Therefore, to call the EXPLAIN_REWRITE procedure using an output table is as follows:

```
DBMS_MVIEW.EXPLAIN_REWRITE (
     query            [VARCHAR2 | CLOB],
     mv                VARCHAR2(30),
     statement_id      VARCHAR2(30));
```

If you want to direct the output of EXPLAIN_REWRITE to a varray instead of a table, you should call the procedure as follows:

```
DBMS_MVIEW.EXPLAIN_REWRITE (
    query           [VARCHAR2 | CLOB],
    mv               VARCHAR2(30),
    output_array     SYS.RewriteArrayType);
```

Note that if the query is less than 256 characters long, EXPLAIN_REWRITE can be easily invoked with the EXECUTE command from SQL*Plus. Otherwise, the recommended method is to use a PL/SQL BEGIN... END block, as shown in the examples in /rdbms/demo/smxrw*.

Using REWRITE_TABLE

Output of EXPLAIN_REWRITE can be directed to a table named REWRITE_TABLE. You can create this output table by running the utlxrw.sql script. This script can be found in the admin directory. The format of REWRITE_TABLE is as follows.

```
CREATE TABLE REWRITE_TABLE(
  statement_id          VARCHAR2(30),    -- ID for the query
  mv_owner              VARCHAR2(30),    -- MV's schema
  mv_name               VARCHAR2(30),    -- Name of the MV
  sequence              INTEGER,         -- Seq # of error msg
  query                 VARCHAR2(2000),  -- user query
  message               VARCHAR2(512),   -- EXPLAIN_REWRITE error msg
  pass                  VARCHAR2(3),     -- Query Rewrite pass no
  mv_in_msg             VARCHAR2(30),    -- MV in current message
  measure_in_msg        VARCHAR2(30),    -- Measure in current message
  join_back_tbl         VARCHAR2(30),    -- Join back table in current msg
```

```
  join_back_col         VARCHAR2(30),    -- Join back column in current msg
  original_cost         INTEGER,         -- Cost of original query
  rewritten_cost        INTEGER,         -- Cost of rewritten query
  flags                 INTEGER,         -- Associated flags
  reserved1             INTEGER,         -- For future use
  reserved2             VARCHAR2(10)     -- For future use
);
```

Example 18–11 EXPLAIN_REWRITE Using REWRITE_TABLE

An example PL/SQL invocation is:

```
EXECUTE DBMS_MVIEW.EXPLAIN_REWRITE \
('SELECT p.prod_name, SUM(amount_sold) ' ||\
'FROM sales s, products p ' ||\
'WHERE s.prod_id = p.prod_id ' ||\
' AND prod_name > ''B%'' ' ||\
' AND prod_name < ''C%'' ' ||\
'GROUP BY prod_name', \
'TestXRW.PRODUCT_SALES_MV', \
'SH');

SELECT message FROM rewrite_table ORDER BY sequence;
MESSAGE
--------------------------------------------------------------------------------
QSM-01033: query rewritten with materialized view, PRODUCT_SALES_MV
1 row selected.
```

The following is another example where you can see a more detailed explanation of why some materialized views were not considered and eventually the materialized view `sales_mv` was chosen as the best one.

```
DECLARE
  qrytext VARCHAR2(500)  :='SELECT cust_first_name, cust_last_name,
 SUM(amount_sold) AS dollar_sales FROM sales s, customers c WHERE s.cust_id=
 c.cust_id GROUP BY cust_first_name, cust_last_name';
    idno    VARCHAR2(30) :='ID1';
BEGIN
  DBMS_MVIEW.EXPLAIN_REWRITE(qrytext, '', idno);
END;
/
SELECT message FROM rewrite_table ORDER BY sequence;


SQL> MESSAGE
--------------------------------------------------------------------------------
```

```
QSM-01082: Joining materialized view, CAL_MONTH_SALES_MV, with table, SALES, not possible
QSM-01022: a more optimal materialized view than PRODUCT_SALES_MV was used to rewrite
QSM-01022: a more optimal materialized view than FWEEK_PSCAT_SALES_MV was used to rewrite
QSM-01033: query rewritten with materialized view, SALES_MV
```

Using a Varray

You can save the output of EXPLAIN_REWRITE in a PL/SQL varray. The elements of this array are of the type RewriteMessage, which is predefined in the SYS schema as shown in the following:

```
TYPE RewriteMessage IS record(
  mv_owner               VARCHAR2(30),  -- MV's schema
  mv_name                VARCHAR2(30),  -- Name of the MV
  sequence               INTEGER,       -- Seq # of error msg
  query_text             VARCHAR2(2000),-- user query
  message                VARCHAR2(512), -- EXPLAIN_REWRITE error msg
  pass                   VARCHAR2(3),   -- Query Rewrite pass no
  mv_in_msg              VARCHAR2(30),  -- MV in current message
  measure_in_msg         VARCHAR2(30),  -- Measure in current message
  join_back_tbl          VARCHAR2(30),  -- Join back table in current msg
  join_back_col          VARCHAR2(30),  -- Join back column in current msg
  original_cost          NUMBER(10),    -- Cost of original query
  rewritten_cost         NUMBER(10),    -- Cost of rewritten query
  flags                  NUMBER,        -- Associated flags
  reserved1              NUMBER,        -- For future use
  reserved2              VARCHAR2(10)   -- For future use
);
```

The array type, RewriteArrayType, which is a varray of RewriteMessage objects, is predefined in the SYS schema as follows:

- `TYPE RewriteArrayType AS VARRAY(256) OF RewriteMessage;`
- Using this array type, now you can declare an array variable and specify it in the EXPLAIN_REWRITE statement.
- Each RewriteMessage record provides a message concerning rewrite processing.
- The parameters are the same as for REWRITE_TABLE, except for `statement_id`, which is not used when using a varray as output.
- The `mv_owner` field defines the owner of materialized view that is relevant to the message.

- The mv_name field defines the name of a materialized view that is relevant to the message.
- The sequence field defines the sequence in which messages should be ordered.
- The query_text field contains the first 2000 characters of the query text under analysis.
- The message field contains the text of message relevant to rewrite processing of query.
- The flags, reserved1, and reserved2 fields are reserved for future use.

Example 18–12 EXPLAIN_REWRITE Using a VARRAY

Consider the following query:

```
SELECT c.cust_state_province, AVG(s.amount_sold)
FROM sales s, customers c WHERE s.cust_id = c.cust_id
GROUP BY c.cust_state_province;
```

If this statement is used with the following materialized view:

```
CREATE MATERIALIZED VIEW avg_sales_city_state_mv
ENABLE QUERY REWRITE AS
SELECT c.cust_city, c.cust_state_province, AVG(s.amount_sold)
FROM sales s, customers c WHERE s.cust_id = c.cust_id
GROUP BY c.cust_city, c.cust_state_province;
```

The query will not rewrite with this materialized view. This can be quite confusing to a novice user as it seems like all information required for rewrite is present in the materialized view. You can find out from DBMS_MVIEW.EXPLAIN_REWRITE that AVG cannot be computed from the given materialized view. The problem is that a ROLLUP is required here and AVG requires a COUNT or a SUM to do ROLLUP.

An example PL/SQL block for the previous query, using a varray as its output, is as follows:

```
SET SERVEROUTPUT ON
DECLARE
  Rewrite_Array SYS.RewriteArrayType := SYS.RewriteArrayType();
  querytxt VARCHAR2(1500) := 'SELECT c.cust_state_province,
  AVG(s.amount_sold)
   FROM sales s, customers c WHERE s.cust_id = c.cust_id
   GROUP BY c.cust_state_province';
  i NUMBER;
```

```
BEGIN
  DBMS_MVIEW.EXPLAIN_REWRITE(querytxt, 'AVG_SALES_CITY_STATE_MV',
  Rewrite_Array);
  FOR i IN 1..Rewrite_Array.count
  LOOP
    DBMS_OUTPUT.PUT_LINE(Rewrite_Array(i).message);
  END LOOP;
END;
/
```

The following is the output of this EXPLAIN_REWRITE statement:

```
QSM-01065: materialized view, AVG_SALES_CITY_STATE_MV, cannot compute
  measure, AVG, in the query
QSM-01101: rollup(s) took place on mv, AVG_SALES_CITY_STATE_MV
QSM-01053: NORELY referential integrity constraint on table, CUSTOMERS,
  in TRUSTED/STALE TOLERATED integrity mode
PL/SQL procedure successfully completed.
```

EXPLAIN_REWRITE Benefit Statistics

The output of EXPLAIN_REWRITE contains two columns, original_cost and rewritten_cost, that can help you estimate query cost. original_cost gives the optimizer's estimation for the query cost when query rewrite was disabled. rewritten_cost gives the optimizer's estimation for the query cost when query was rewritten using a materialized view. These cost values can be used to find out what benefit a particular query will receive from rewrite.

Support for Query Text Larger than 32KB in EXPLAIN_REWRITE

In this release, the EXPLAIN_REWRITE procedure has been enhanced to support large queries. The input query text can now be defined using a CLOB datatype instead of a VARCHAR datatype. This allows EXPLAIN_REWRITE to accept queries up to 4 GB.

The syntax for using EXPLAIN_REWRITE using CLOB to obtain the output into a table is shown as follows:

```
DBMS_MVIEW.EXPLAIN_REWRITE(
   query          IN CLOB,
   mv             IN VARCHAR2,
   statement_id   IN VARCHAR2);
```

The second argument, mv, and the third argument, statement_id, can be NULL. Similarly, the syntax for using EXPLAIN_REWRITE using CLOB to obtain the output into a varray is shown as follows:

```
DBMS_MVIEW.EXPLAIN_REWRITE(
   query           IN CLOB,
   mv              IN VARCHAR2,
   msg_array       IN OUT SYS.RewriteArrayType);
```

As before, the second argument, mv, can be NULL. Note that long query texts in CLOB can be generated using the procedures provided in the DBMS_LOB package.

Design Considerations for Improving Query Rewrite Capabilities

This section discusses design considerations that will help in obtaining the maximum benefit from query rewrite. They are not mandatory for using query rewrite and rewrite is not guaranteed if you follow them. They are general rules to consider, and are the following:

- Query Rewrite Considerations: Constraints
- Query Rewrite Considerations: Dimensions
- Query Rewrite Considerations: Outer Joins
- Query Rewrite Considerations: Text Match
- Query Rewrite Considerations: Aggregates
- Query Rewrite Considerations: Grouping Conditions
- Query Rewrite Considerations: Expression Matching
- Query Rewrite Considerations: Date Folding
- Query Rewrite Considerations: Statistics

Query Rewrite Considerations: Constraints

Make sure all inner joins referred to in a materialized view have referential integrity (foreign key / primary key constraints) with additional NOT NULL constraints on the foreign key columns. Since constraints tend to impose a large overhead, you could make them NO VALIDATE and RELY and set the parameter QUERY_REWRITE_INTEGRITY to STALE_TOLERATED or TRUSTED. However, if you set QUERY_REWRITE_INTEGRITY to ENFORCED, all constraints must be enabled, enforced, and validated to get maximum rewritability.

You should avoid using the ON DELETE clause as it can lead to unexpected results.

Query Rewrite Considerations: Dimensions

You can express the hierarchical relationships and functional dependencies in normalized or denormalized dimension tables using the HIERARCHY and DETERMINES clauses of a dimension. Dimensions can express intra-table relationships which cannot be expressed by constraints. Set the parameter QUERY_REWRITE_INTEGRITY to TRUSTED or STALE_TOLERATED for query rewrite to take advantage of the relationships declared in dimensions.

Query Rewrite Considerations: Outer Joins

Another way of avoiding constraints is to use outer joins in the materialized view. Query rewrite will be able to derive an inner join in the query, such as (A.a=B.b), from an outer join in the materialized view (A.a = B.b(+)), as long as the rowid of B or column B.b is available in the materialized view. Most of the support for rewrites with outer joins is provided for materialized views with joins only. To exploit it, a materialized view with outer joins should store the rowid or primary key of the inner table of an outer join. For example, the materialized view join_sales_time_product_mv_oj stores the primary keys prod_id and time_id of the inner tables of outer joins.

Query Rewrite Considerations: Text Match

If you need to speed up an extremely complex, long-running query, you could create a materialized view with the exact text of the query. Then the materialized view would contain the query results, thus eliminating the time required to perform any complex joins and search through all the data for that which is required.

Query Rewrite Considerations: Aggregates

To get the maximum benefit from query rewrite, make sure that all aggregates which are needed to compute ones in the targeted set of queries are present in the materialized view. The conditions on aggregates are quite similar to those for incremental refresh. For instance, if AVG(x) is in the query, then you should store COUNT(x) and AVG(x) or store SUM(x) and COUNT(x) in the materialized view. See "General Restrictions on Fast Refresh" on page 8-27 for fast refresh requirements.

Query Rewrite Considerations: Grouping Conditions

Aggregating data at lower levels in the hierarchy is better than aggregating at higher levels because lower levels can be used to rewrite more queries. Note, however, that doing so will also take up more space. For example, instead of grouping on state, group on city (unless space constraints prohibit it).

Instead of creating multiple materialized views with overlapping or hierarchically related GROUP BY columns, create a single materialized view with all those GROUP BY columns. For example, instead of using a materialized view that groups by city and another materialized view that groups by month, use a single materialized view that groups by city and month.

Use GROUP BY on columns that correspond to levels in a dimension but not on columns that are functionally dependent, because query rewrite will be able to use the functional dependencies automatically based on the DETERMINES clause in a dimension. For example, instead of grouping on `prod_name`, group on `prod_id` (as long as there is a dimension which indicates that the attribute `prod_id` determines `prod_name`, you will enable the rewrite of a query involving `prod_name`).

Query Rewrite Considerations: Expression Matching

If several queries share the same common subselect, it is advantageous to create a materialized view with the common subselect as one of its SELECT columns. This way, the performance benefit due to precomputation of the common subselect can be obtained across several queries.

Query Rewrite Considerations: Date Folding

When creating a materialized view which aggregates data by folded date granules such as months or quarters or years, always use the year component as the prefix but not as the suffix. For example, `TO_CHAR(date_col, 'yyyy-q')` folds the date into quarters, which collate in year order, whereas `TO_CHAR(date_col, 'q-yyyy')` folds the date into quarters, which collate in quarter order. The former preserves the ordering while the latter does not. For this reason, any materialized view created without a year prefix will not be eligible for date folding rewrite.

Query Rewrite Considerations: Statistics

Optimization with materialized views is based on cost and the optimizer needs statistics of both the materialized view and the tables in the query to make a

cost-based choice. Materialized views should thus have statistics collected using the DBMS_STATS package.

Advanced Rewrite Using Equivalences

There is a special type of query rewrite that is possible where a declaration is made that two SQL statements are functionally equivalent. This capability enables you to place inside application knowledge into the database so the database can exploit this knowledge for improved query performance. You do this by declaring two SELECT statements to be functionally equivalent (returning the same rows and columns) and indicating that one of the SELECT statements is more favorable for performance.

This advanced rewrite capability can generally be applied to a variety of query performance problems and opportunities. Any application can use this capability to affect rewrites against complex user queries that can be answered with much simpler and more performant queries that have been specifically created, usually by someone with inside application knowledge.

There are many scenarios where you can have inside application knowledge that would allow SQL statement transformation and tuning for significantly improved performance. The types of optimizations you may wish to affect can be very simple or as sophisticated as significant restructuring of the query. However, the incoming SQL queries are often generated by applications and you have no control over the form and structure of the application-generated queries.

To gain access to this capability, you need to connect as SYSDBA and explicitly grant execute access to the desired database administrators who will be declaring rewrite equivalences. See *PL/SQL Packages and Types Reference* for more information.

To illustrate advanced rewrite, some examples using the OLAP Server environment are offered in this section. OLAP Server provides users with efficient access to multi-dimensional data. To optimize resource usage, OLAP Server may employ complicated SQL, custom C code or table functions to retrieve the data from the database. This complexity is irrelevant as far as end users are concerned. Users would still want to obtain their answers using typical queries with SELECT ... GROUP BY.

In the following example, OLAP Server declares to Oracle that a given user query must be executed using a specified alternative query. Oracle would recognize this relationship and every time the user asked the query, it would transparently rewrite it using the alternative. Thus, the user is saved from the trouble of understanding and writing SQL for complicated aggregate computations.

Example 18–13 Rewrite Using Equivalence

There are two base tables `sales_fact` and `geog_dim`. You can compute the total sales for each city, state and region with a rollup, by issuing the following statement:

```
SELECT g.region, g.state, g.city,
GROUPING_ID(g.city, g.state, g.region), SUM(sales)
FROM sales_fact f, geog_dim g WHERE f.geog_key = g.geog_key
GROUP BY ROLLUP(g.region, g.state, g.city);
```

OLAP Server would want to materialize this query for quick results. Unfortunately, the resulting materialized view occupies too much disk space. However, if you have a dimension rolling up city to state to region, you can easily compress the three grouping columns into one column using a decode statement. (This is also known as an embedded total):

```
DECODE (gid, 0, city, 1, state, 3, region, 7, "grand_total")
```

What this does is use the lowest level of the hierarchy to represent the entire information. For example, saying `Boston` means `Boston, MA, New England Region` and saying `CA` to mean `CA, Western Region`. OLAP Server stores these embedded total results into a table, say, `embedded_total_sales`.

However, when returning the result back to the user, you would want to have all the data columns (city, state, region). In order to return the results efficiently and quickly, OLAP Server may use a custom table function (`et_function`) to retrieve the data back from the `embedded_total_sales` table in the expanded form as follows:

```
SELECT * FROM TABLE (et_function);
```

In other words, this feature allows OLAP Server to declare the equivalence of the user's preceding query to the alternative query OLAP Server uses to compute it, as in the following:

```
DBMS_ADVANCED_REWRITE.DECLARE_REWRITE_EQUIVALENCE (
   'OLAPI_EMBEDDED_TOTAL',
   'SELECT g.region, g.state, g.city,
    GROUPING_ID(g.city, g.state, g.region), SUM(sales)
    FROM sales_fact f, geog_dim g
    WHERE f.geog_key = g.geog_key
    GROUP BY ROLLUP(g.region, g.state, g.city)',
    'SELECT * FROM TABLE(et_function)');
```

This invocation of DECLARE_REWRITE_EQUIVALENCE creates an equivalence declaration named OLAPI_EMBEDDED_TOTAL stating that the specified SOURCE_STMT and the specified DESTINATION_STMT are functionally equivalent, and that the specified DESTINATION_STMT is preferable for performance. After the DBA creates such a declaration, the user need have no knowledge of the space optimization being performed underneath the covers.

This capability also allows OLAPI to perform specialized partial materializations of a SQL query. For instance, it could perform a rollup using a UNION ALL of three relations as shown in Example 18–14.

Example 18–14 Rewrite Using Equivalence (UNION ALL)

```
CREATE MATERIALIZED VIEW T1
AS SELECT g.region, g.state, g,city, 0 as gid, SUM(sales) sales
FROM sales_fact f, geog_dim g WHERE f.geog_key = g.geog_key
GROUP BY g.region, g.state, g.city;
CREATE MATERIALIZED VIEW T2 AS
SELECT region, state, SUM(sales) sales
FROM T1 GROUP BY region, state;
CREATE VIEW T3 AS
SELECT region, SUM(sales) sales
FROM T2 GROUP BY region;
```

The ROLLUP(region, state, city) query is then equivalent to:

```
SELECT * FROM T1 UNION ALL
SELECT region, state, NULL, 1 AS gid, sales FROM T2 UNION ALL
SELECT region, NULL, NULL, 3 AS gid, sales FROM T3 UNION ALL
SELECT NULL, NULL, NULL, 7 AS gid, SUM(sales) FROM T3;
```

By specifying this equivalence, Oracle would use the more efficient second form of the query to compute the ROLLUP query asked by the user.

```
DBMS_ADVANCED_REWRITE.DECLARE_REWRITE_EQUIVALENCE (
   'OLAPI_ROLLUP',
   'SELECT g.region, g.state, g.city,
   GROUPING_ID(g.city, g.state, g.region), SUM(sales)
   FROM sales_fact f, geog_dim g
   WHERE f.geog_key = g.geog_key
   GROUP BY ROLLUP(g.region, g.state, g.city ',
   ' SELECT * FROM T1
   UNION ALL
   SELECT region, state, NULL, 1 as gid, sales FROM T2
   UNION ALL
```

```
   SELECT region, NULL, NULL, 3 as gid, sales FROM T3
   UNION ALL
   SELECT NULL, NULL, NULL, 7 as gid, SUM(sales) FROM T3');
```

Another application of this feature is to provide users special aggregate computations that may be conceptually simple but extremely complex to express in SQL. In this case, OLAP Server asks the user to use a specified custom aggregate function and internally compute it using complex SQL.

Example 18–15 Rewrite Using Equivalence (Using a Custom Aggregate)

Suppose the application users want to see the sales for each city, state and region and also additional sales information for specific seasons. For example, the New England user wants additional sales information for cities in New England for the winter months. OLAP Server would provide you a special aggregate `Seasonal_Agg` that computes the earlier aggregate. You would ask a classic summary query but use `Seasonal_Agg(sales, region)` rather than `SUM(sales)`.

```
SELECT g.region, t,monthname, Seasonal_Agg(sales, region) AS sales
FROM sales_fact f, geog_dim g, time t
WHERE f.geog_key = g.geog_key AND f.time_key = t.time_key
GROUP BY g.region, t.monthname;
```

Instead of asking the user to write SQL that does the extra computation, OLAP Server does it for them by using this feature. In this example, `Seasonal_Agg` is computed using the spreadsheet functionality (see Chapter 22, "SQL for Modeling"). Note that even though `Seasonal_Agg` is a user-defined aggregate, the required behavior is to add extra rows to the query's answer, which cannot be easily done with simple PL/SQL functions.

```
DBMS_ADVANCED_REWRITE.DECLARE_REWRITE_EQUIVALENCE (
   'OLAPI_SEASONAL_AGG',
   SELECT g.region, t,monthname, Seasonal_Agg(sales, region) AS sales
   FROM sales_fact f, geog_dim g, time t
   WHERE f.geog_key = g.geog_key and f.time_key = t.time_key
   GROUP BY g.region, t.monthname',
   'SELECT g,region, t.monthname, SUM(sales) AS sales
   FROM sales_fact f, geog_dim g
   WHERE f.geog_key = g.geog_key and t.time_key = f.time_key
   GROUP BY g.region, g.state, g.city, t.monthname
   DIMENSION BY g.region, t.monthname
   (sales ['New England', 'Winter'] = AVG(sales) OVER monthname IN
    ('Dec', 'Jan', 'Feb', 'Mar'),
   sales ['Western', 'Summer' ] = AVG(sales) OVER monthname IN
   ('May', 'Jun', 'July', 'Aug'), .);
```

19

Schema Modeling Techniques

The following topics provide information about schemas in a data warehouse:

- Schemas in Data Warehouses
- Third Normal Form
- Star Schemas
- Optimizing Star Queries

Schemas in Data Warehouses

A **schema** is a collection of database objects, including tables, views, indexes, and synonyms.

There is a variety of ways of arranging schema objects in the schema models designed for data warehousing. One data warehouse schema model is a star schema. The sh sample schema (the basis for most of the examples in this book) uses a star schema. However, there are other schema models that are commonly used for data warehouses. The most prevalent of these schema models is the **third normal form (3NF)** schema. Additionally, some data warehouse schemas are neither star schemas nor 3NF schemas, but instead share characteristics of both schemas; these are referred to as hybrid schema models.

The Oracle Database is designed to support all data warehouse schemas. Some features may be specific to one schema model (such as the star transformation feature, described in "Using Star Transformation" on page 19-6, which is specific to star schemas). However, the vast majority of Oracle's data warehousing features are equally applicable to star schemas, 3NF schemas, and hybrid schemas. Key data warehousing capabilities such as partitioning (including the rolling window load technique), parallelism, materialized views, and analytic SQL are implemented in all schema models.

The determination of which schema model should be used for a data warehouse should be based upon the requirements and preferences of the data warehouse project team. Comparing the merits of the alternative schema models is outside of the scope of this book; instead, this chapter will briefly introduce each schema model and suggest how Oracle can be optimized for those environments.

Third Normal Form

Although this guide primarily uses star schemas in its examples, you can also use the third normal form for your data warehouse implementation.

Third normal form modeling is a classical relational-database modeling technique that minimizes data redundancy through normalization. When compared to a star schema, a 3NF schema typically has a larger number of tables due to this normalization process. For example, in Figure 19–1, `orders` and `order items` tables contain similar information as `sales` table in the star schema in Figure 19–2.

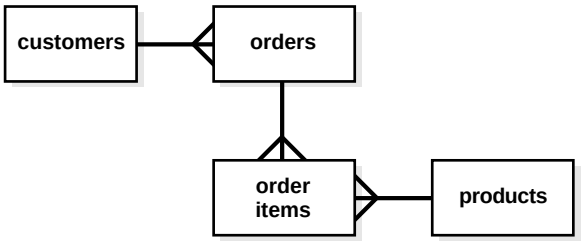
3NF schemas are typically chosen for large data warehouses, especially environments with significant data-loading requirements that are used to feed data marts and execute long-running queries.

The main advantages of 3NF schemas are that they:

- Provide a neutral schema design, independent of any application or data-usage considerations
- May require less data-transformation than more normalized schemas such as star schemas

Figure 19–1 presents a graphical representation of a third normal form schema.

Figure 19–1 Third Normal Form Schema



Optimizing Third Normal Form Queries

Queries on 3NF schemas are often very complex and involve a large number of tables. The performance of joins between large tables is thus a primary consideration when using 3NF schemas.

One particularly important feature for 3NF schemas is partition-wise joins. The largest tables in a 3NF schema should be partitioned to enable partition-wise joins. The most common partitioning technique in these environments is composite range-hash partitioning for the largest tables, with the most-common join key chosen as the hash-partitioning key.

Parallelism is often heavily utilized in 3NF environments, and parallelism should typically be enabled in these environments.

Star Schemas

The **star schema** is perhaps the simplest data warehouse schema. It is called a star schema because the entity-relationship diagram of this schema resembles a star, with points radiating from a central table. The center of the star consists of a large fact table and the points of the star are the dimension tables.

A **star query** is a join between a fact table and a number of dimension tables. Each dimension table is joined to the fact table using a primary key to foreign key join, but the dimension tables are not joined to each other. The optimizer recognizes star queries and generates efficient execution plans for them.

A typical fact table contains keys and measures. For example, in the `sh` sample schema, the fact table, `sales`, contain the measures `quantity_sold`, `amount`, and `cost`, and the keys `cust_id`, `time_id`, `prod_id`, `channel_id`, and `promo_id`. The dimension tables are `customers`, `times`, `products`, `channels`, and `promotions`. The `products` dimension table, for example, contains information about each product number that appears in the fact table.

A star join is a primary key to foreign key join of the dimension tables to a fact table.

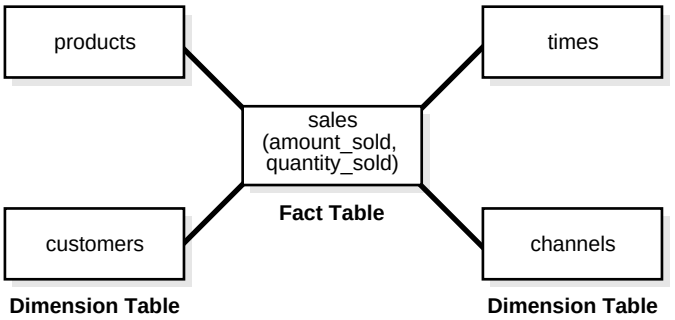
The main advantages of star schemas are that they:

- Provide a direct and intuitive mapping between the business entities being analyzed by end users and the schema design.
- Provide highly optimized performance for typical star queries.
- Are widely supported by a large number of business intelligence tools, which may anticipate or even require that the data warehouse schema contain dimension tables.

Star schemas are used for both simple data marts and very large data warehouses.

Figure 19–2 presents a graphical representation of a star schema.

Figure 19–2 Star Schema

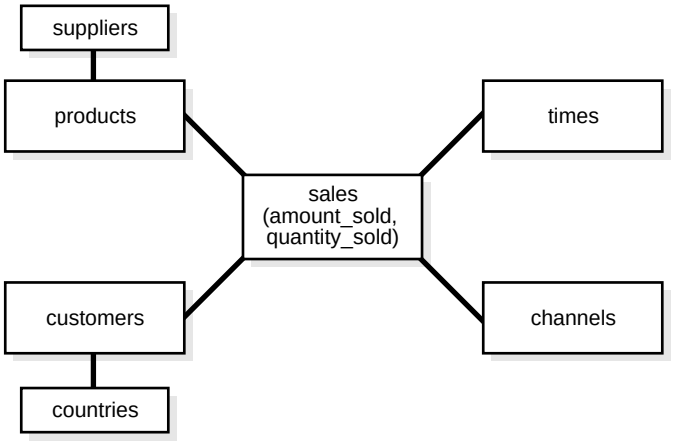


Snowflake Schemas

The snowflake schema is a more complex data warehouse model than a star schema, and is a type of star schema. It is called a snowflake schema because the diagram of the schema resembles a snowflake.

Snowflake schemas normalize dimensions to eliminate redundancy. That is, the dimension data has been grouped into multiple tables instead of one large table. For example, a product dimension table in a star schema might be normalized into a products table, a `product_category` table, and a `product_manufacturer` table in a snowflake schema. While this saves space, it increases the number of dimension tables and requires more foreign key joins. The result is more complex queries and reduced query performance. Figure 19–3 presents a graphical representation of a snowflake schema.

Figure 19–3 Snowflake Schema



Note: Oracle Corporation recommends you choose a star schema over a snowflake schema unless you have a clear reason not to.

Optimizing Star Queries

You should consider the following when using star queries:

- Tuning Star Queries

Tuning Star Queries

To get the best possible performance for star queries, it is important to follow some basic guidelines:

- A bitmap index should be built on each of the foreign key columns of the fact table or tables.
- The initialization parameter STAR_TRANSFORMATION_ENABLED should be set to TRUE. This enables an important optimizer feature for star-queries. It is set to FALSE by default for backward-compatibility.

When a data warehouse satisfies these conditions, the majority of the star queries running in the data warehouse will use a query execution strategy known as the star transformation. The star transformation provides very efficient query performance for star queries.

Using Star Transformation

The star transformation is a powerful optimization technique that relies upon implicitly rewriting (or transforming) the SQL of the original star query. The end user never needs to know any of the details about the star transformation. Oracle's query optimizer automatically chooses the star transformation where appropriate.

The star transformation is a query transformation aimed at executing star queries efficiently. Oracle processes a star query using two basic phases. The first phase retrieves exactly the necessary rows from the fact table (the result set). Because this retrieval utilizes bitmap indexes, it is very efficient. The second phase joins this result set to the dimension tables. An example of an end user query is: "What were the sales and profits for the grocery department of stores in the west and southwest sales districts over the last three quarters?" This is a simple star query.

Note: Bitmap indexes are available only if you have purchased the Oracle Database Enterprise Edition. In Oracle Database Standard Edition, bitmap indexes and star transformation are not available.

Star Transformation with a Bitmap Index

A prerequisite of the star transformation is that there be a single-column bitmap index on every join column of the fact table. These join columns include all foreign key columns.

For example, the sales table of the sh sample schema has bitmap indexes on the time_id, channel_id, cust_id, prod_id, and promo_id columns.

Consider the following star query:

```
SELECT ch.channel_class, c.cust_city, t.calendar_quarter_desc,
   SUM(s.amount_sold) sales_amount
FROM sales s, times t, customers c, channels ch
WHERE s.time_id = t.time_id
AND   s.cust_id = c.cust_id
AND   s.channel_id = ch.channel_id
AND   c.cust_state_province = 'CA'
AND   ch.channel_desc in ('Internet','Catalog')
AND   t.calendar_quarter_desc IN ('1999-Q1','1999-Q2')
GROUP BY ch.channel_class, c.cust_city, t.calendar_quarter_desc;
```

This query is processed in two phases. In the first phase, Oracle Database uses the bitmap indexes on the foreign key columns of the fact table to identify and retrieve only the necessary rows from the fact table. That is, Oracle Database will retrieve the result set from the fact table using essentially the following query:

```
SELECT ... FROM sales
WHERE time_id IN
  (SELECT time_id FROM times
   WHERE calendar_quarter_desc IN('1999-Q1','1999-Q2'))
   AND cust_id IN
  (SELECT cust_id FROM customers WHERE cust_state_province='CA')
   AND channel_id IN
  (SELECT channel_id FROM channels WHERE channel_desc
IN('Internet','Catalog'));
```

This is the transformation step of the algorithm, because the original star query has been transformed into this subquery representation. This method of accessing the fact table leverages the strengths of bitmap indexes. Intuitively, bitmap indexes provide a set-based processing scheme within a relational database. Oracle has implemented very fast methods for doing set operations such as AND (an intersection in standard set-based terminology), OR (a set-based union), MINUS, and COUNT.

In this star query, a bitmap index on time_id is used to identify the set of all rows in the fact table corresponding to sales in 1999-Q1. This set is represented as a bitmap (a string of 1's and 0's that indicates which rows of the fact table are members of the set).

A similar bitmap is retrieved for the fact table rows corresponding to the sale from 1999-Q2. The bitmap OR operation is used to combine this set of Q1 sales with the set of Q2 sales.

Additional set operations will be done for the customer dimension and the product dimension. At this point in the star query processing, there are three bitmaps. Each bitmap corresponds to a separate dimension table, and each bitmap represents the set of rows of the fact table that satisfy that individual dimension's constraints.

These three bitmaps are combined into a single bitmap using the bitmap AND operation. This final bitmap represents the set of rows in the fact table that satisfy all of the constraints on the dimension table. This is the result set, the exact set of rows from the fact table needed to evaluate the query. Note that none of the actual data in the fact table has been accessed. All of these operations rely solely on the bitmap indexes and the dimension tables. Because of the bitmap indexes' compressed data representations, the bitmap set-based operations are extremely efficient.

Once the result set is identified, the bitmap is used to access the actual data from the sales table. Only those rows that are required for the end user's query are retrieved from the fact table. At this point, Oracle has effectively joined all of the dimension tables to the fact table using bitmap indexes. This technique provides excellent performance because Oracle is joining all of the dimension tables to the fact table with one logical join operation, rather than joining each dimension table to the fact table independently.

The second phase of this query is to join these rows from the fact table (the result set) to the dimension tables. Oracle will use the most efficient method for accessing and joining the dimension tables. Many dimension are very small, and table scans are typically the most efficient access method for these dimension tables. For large dimension tables, table scans may not be the most efficient access method. In the previous example, a bitmap index on product.department can be used to quickly identify all of those products in the grocery department. Oracle's optimizer automatically determines which access method is most appropriate for a given dimension table, based upon the optimizer's knowledge about the sizes and data distributions of each dimension table.

The specific join method (as well as indexing method) for each dimension table will likewise be intelligently determined by the optimizer. A hash join is often the most efficient algorithm for joining the dimension tables. The final answer is returned to the user once all of the dimension tables have been joined. The query technique of retrieving only the matching rows from one table and then joining to another table is commonly known as a semijoin.

Execution Plan for a Star Transformation with a Bitmap Index

The following typical execution plan might result from "Star Transformation with a Bitmap Index" on page 19-6:

```
SELECT STATEMENT
 SORT GROUP BY
  HASH JOIN
   TABLE ACCESS FULL                          CHANNELS
   HASH JOIN
    TABLE ACCESS FULL                         CUSTOMERS
    HASH JOIN
     TABLE ACCESS FULL                        TIMES
     PARTITION RANGE ITERATOR
      TABLE ACCESS BY LOCAL INDEX ROWID       SALES
       BITMAP CONVERSION TO ROWIDS
        BITMAP AND
         BITMAP MERGE
          BITMAP KEY ITERATION
           BUFFER SORT
            TABLE ACCESS FULL                 CUSTOMERS
           BITMAP INDEX RANGE SCAN            SALES_CUST_BIX
         BITMAP MERGE
          BITMAP KEY ITERATION
           BUFFER SORT
            TABLE ACCESS FULL                 CHANNELS
           BITMAP INDEX RANGE SCAN            SALES_CHANNEL_BIX
         BITMAP MERGE
          BITMAP KEY ITERATION
           BUFFER SORT
            TABLE ACCESS FULL                 TIMES
           BITMAP INDEX RANGE SCAN            SALES_TIME_BIX
```

In this plan, the fact table is accessed through a bitmap access path based on a bitmap AND, of three merged bitmaps. The three bitmaps are generated by the BITMAP MERGE row source being fed bitmaps from row source trees underneath it. Each such row source tree consists of a BITMAP KEY ITERATION row source which fetches values from the subquery row source tree, which in this example is a full table access. For each such value, the BITMAP KEY ITERATION row source retrieves the bitmap from the bitmap index. After the relevant fact table rows have been retrieved using this access path, they are joined with the dimension tables and temporary tables to produce the answer to the query.

Star Transformation with a Bitmap Join Index

In addition to bitmap indexes, you can use a bitmap join index during star transformations. Assume you have the following additional index structure:

```
CREATE BITMAP INDEX sales_c_state_bjix
ON sales(customers.cust_state_province)
FROM sales, customers
WHERE sales.cust_id = customers.cust_id
LOCAL NOLOGGING COMPUTE STATISTICS;
```

The processing of the same star query using the bitmap join index is similar to the previous example. The only difference is that Oracle will utilize the join index, instead of a single-table bitmap index, to access the customer data in the first phase of the star query.

Execution Plan for a Star Transformation with a Bitmap Join Index

The following typical execution plan might result from "Execution Plan for a Star Transformation with a Bitmap Join Index" on page 19-10:

```
SELECT STATEMENT
 SORT GROUP BY
  HASH JOIN
   TABLE ACCESS FULL                          CHANNELS
   HASH JOIN
    TABLE ACCESS FULL                         CUSTOMERS
    HASH JOIN
     TABLE ACCESS FULL                        TIMES
     PARTITION RANGE ALL
      TABLE ACCESS BY LOCAL INDEX ROWID       SALES
       BITMAP CONVERSION TO ROWIDS
        BITMAP AND
         BITMAP INDEX SINGLE VALUE            SALES_C_STATE_BJIX
         BITMAP MERGE
          BITMAP KEY ITERATION
           BUFFER SORT
            TABLE ACCESS FULL                 CHANNELS
           BITMAP INDEX RANGE SCAN            SALES_CHANNEL_BIX
         BITMAP MERGE
          BITMAP KEY ITERATION
           BUFFER SORT
            TABLE ACCESS FULL                 TIMES
           BITMAP INDEX RANGE SCAN            SALES_TIME_BIX
```

The difference between this plan as compared to the previous one is that the inner part of the bitmap index scan for the `customer` dimension has no subselect. This is because the join predicate information on `customer.cust_state_province` can be satisfied with the bitmap join index `sales_c_state_bjix`.

How Oracle Chooses to Use Star Transformation

The optimizer generates and saves the best plan it can produce without the transformation. If the transformation is enabled, the optimizer then tries to apply it to the query and, if applicable, generates the best plan using the transformed query. Based on a comparison of the cost estimates between the best plans for the two versions of the query, the optimizer will then decide whether to use the best plan for the transformed or untransformed version.

If the query requires accessing a large percentage of the rows in the fact table, it might be better to use a full table scan and not use the transformations. However, if the constraining predicates on the dimension tables are sufficiently selective that only a small portion of the fact table needs to be retrieved, the plan based on the transformation will probably be superior.

Note that the optimizer generates a subquery for a dimension table only if it decides that it is reasonable to do so based on a number of criteria. There is no guarantee that subqueries will be generated for all dimension tables. The optimizer may also decide, based on the properties of the tables and the query, that the transformation does not merit being applied to a particular query. In this case the best regular plan will be used.

Star Transformation Restrictions

Star transformation is not supported for tables with any of the following characteristics:

- Queries with a table hint that is incompatible with a bitmap access path
- Queries that contain bind variables
- Tables with too few bitmap indexes. There must be a bitmap index on a fact table column for the optimizer to generate a subquery for it.
- Remote fact tables. However, remote dimension tables are allowed in the subqueries that are generated.
- Anti-joined tables
- Tables that are already used as a dimension table in a subquery

- Tables that are really unmerged views, which are not view partitions

The star transformation may not be chosen by the optimizer for the following cases:

- Tables that have a good single-table access path
- Tables that are too small for the transformation to be worthwhile

In addition, temporary tables will not be used by star transformation under the following conditions:

- The database is in read-only mode
- The star query is part of a transaction that is in serializable mode

20

SQL for Aggregation in Data Warehouses

This chapter discusses aggregation of SQL, a basic aspect of data warehousing. It contains these topics:

Overview of SQL for Aggregation in Data Warehouses

Aggregation is a fundamental part of data warehousing. To improve aggregation performance in your warehouse, Oracle Database provides the following extensions to the GROUP BY clause:

- CUBE and ROLLUP extensions to the GROUP BY clause
- Three GROUPING functions
- GROUPING SETS expression

The CUBE, ROLLUP, and GROUPING SETS extensions to SQL make querying and reporting easier and faster. CUBE, ROLLUP, and grouping sets produce a single result set that is equivalent to a UNION ALL of differently grouped rows. ROLLUP calculates aggregations such as SUM, COUNT, MAX, MIN, and AVG at increasing levels of aggregation, from the most detailed up to a grand total. CUBE is an extension similar to ROLLUP, enabling a single statement to calculate all possible combinations of aggregations. The CUBE, ROLLUP, and the GROUPING SETS extension lets you specify just the groupings needed in the GROUP BY clause. This allows efficient analysis across multiple dimensions without performing a CUBE operation. Computing a CUBE creates a heavy processing load, so replacing cubes with grouping sets can significantly increase performance.

To enhance performance, CUBE, ROLLUP, and GROUPING SETS can be parallelized: multiple processes can simultaneously execute all of these statements. These capabilities make aggregate calculations more efficient, thereby enhancing database performance, and scalability.

The three GROUPING functions help you identify the group each row belongs to and enable sorting subtotal rows and filtering results.

Analyzing Across Multiple Dimensions

One of the key concepts in decision support systems is multidimensional analysis: examining the enterprise from all necessary combinations of dimensions. We use the term **dimension** to mean any category used in specifying questions. Among the most commonly specified dimensions are time, geography, product, department, and distribution channel, but the potential dimensions are as endless as the varieties of enterprise activity. The events or entities associated with a particular set of dimension values are usually referred to as facts. The facts might be sales in units or local currency, profits, customer counts, production volumes, or anything else worth tracking.

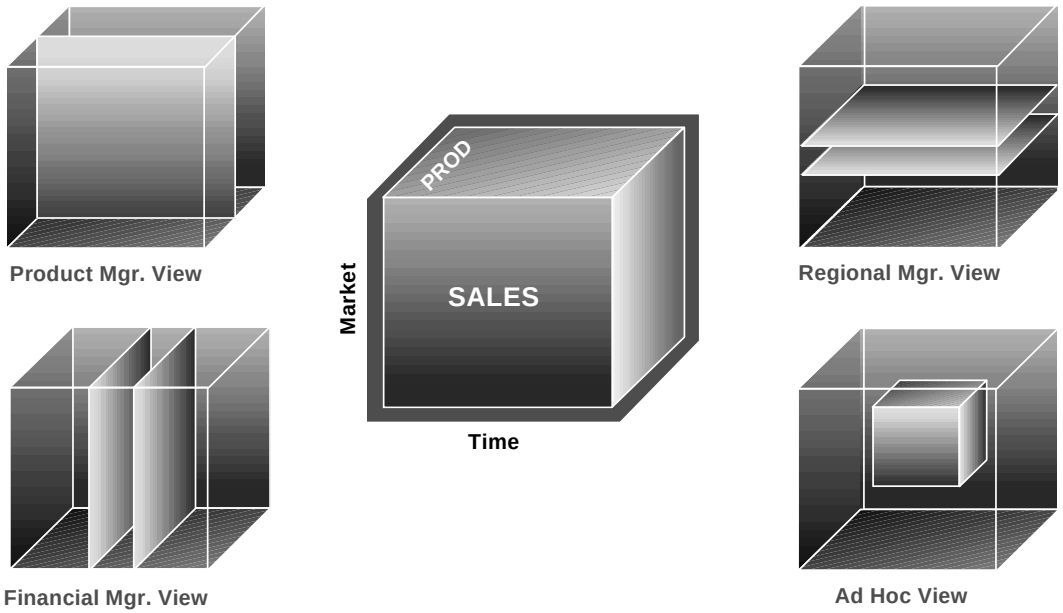
Here are some examples of multidimensional requests:

- Show total sales across all products at increasing aggregation levels for a geography dimension, from state to country to region, for 1999 and 2000.
- Create a cross-tabular analysis of our operations showing expenses by territory in South America for 1999 and 2000. Include all possible subtotals.
- List the top 10 sales representatives in Asia according to 2000 sales revenue for automotive products, and rank their commissions.

All these requests involve multiple dimensions. Many multidimensional questions require aggregated data and comparisons of data sets, often across time, geography or budgets.

To visualize data that has many dimensions, analysts commonly use the analogy of a data cube, that is, a space where facts are stored at the intersection of n dimensions. Figure 20–1 shows a data cube and how it can be used differently by various groups. The cube stores sales data organized by the dimensions of product, market, sales, and time. Note that this is only a metaphor: the actual data is physically stored in normal tables. The cube data consists of both detail and aggregated data.

Figure 20–1 Logical Cubes and Views by Different Users



You can retrieve slices of data from the cube. These correspond to cross-tabular reports such as the one shown in Table 20–1. Regional managers might study the data by comparing slices of the cube applicable to different markets. In contrast, product managers might compare slices that apply to different products. An ad hoc user might work with a wide variety of constraints, working in a subset cube.

Answering multidimensional questions often involves accessing and querying huge quantities of data, sometimes in millions of rows. Because the flood of detailed data generated by large organizations cannot be interpreted at the lowest level, aggregated views of the information are essential. Aggregations, such as sums and counts, across many dimensions are vital to multidimensional analyses. Therefore, analytical tasks require convenient and efficient data aggregation.

Optimized Performance

Not only multidimensional issues, but all types of processing can benefit from enhanced aggregation facilities. Transaction processing, financial and manufacturing systems—all of these generate large numbers of production reports needing substantial system resources. Improved efficiency when creating these reports will reduce system load. In fact, any computer process that aggregates data from details to higher levels will benefit from optimized aggregation performance.

These extensions provide aggregation features and bring many benefits, including:

- Simplified programming requiring less SQL code for many tasks.
- Quicker and more efficient query processing.
- Reduced client processing loads and network traffic because aggregation work is shifted to servers.
- Opportunities for caching aggregations because similar queries can leverage existing work.

An Aggregate Scenario

To illustrate the use of the `GROUP BY` extension, this chapter uses the sh data of the sample schema. All the examples refer to data from this scenario. The hypothetical company has sales across the world and tracks sales by both dollars and quantities information. Because there are many rows of data, the queries shown here typically have tight constraints on their `WHERE` clauses to limit the results to a small number of rows.

Example 20–1 Simple Cross-Tabular Report With Subtotals

Table 20–1 is a sample cross-tabular report showing the total sales by `country_id` and `channel_desc` for the US and France through the Internet and direct sales in September 2000.

Table 20–1 Simple Cross-Tabular Report With Subtotals

Channel	Country		
	France	US	Total
Internet	9,597	124,224	133,821
Direct Sales	61,202	638,201	699,403
Total	70,799	762,425	833,224

Consider that even a simple report such as this, with just nine values in its grid, generates four subtotals and a grand total. Half of the values needed for this report would not be calculated with a query that requested `SUM(amount_sold)` and did a `GROUP BY(channel_desc, country_id)`. To get the higher-level aggregates would require additional queries. Database commands that offer improved calculation of subtotals bring major benefits to querying, reporting, and analytical operations.

```
SELECT channels.channel_desc, countries.country_iso_code,
  TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels, countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND
  sales.channel_id= channels.channel_id AND channels.channel_desc IN
  ('Direct Sales', 'Internet') AND times.calendar_month_desc='2000-09'
  AND customers.country_id=countries.country_id
  AND countries.country_iso_code IN ('US','FR')
GROUP BY CUBE(channels.channel_desc, countries.country_iso_code);

CHANNEL_DESC         CO SALES$
-------------------- -- --------------
                              833,224
                     FR        70,799
                     US       762,425
Internet                      133,821
Internet             FR         9,597
Internet             US       124,224
Direct Sales                  699,403
Direct Sales         FR        61,202
Direct Sales         US       638,201
```

Interpreting NULLs in Examples

NULLs returned by the GROUP BY extensions are not always the traditional null meaning value unknown. Instead, a NULL may indicate that its row is a subtotal. To avoid introducing another non-value in the database system, these subtotal values are not given a special tag. See "GROUPING Functions" on page 20-12 for details on how the NULLs representing subtotals are distinguished from NULLs stored in the data.

ROLLUP Extension to GROUP BY

ROLLUP enables a SELECT statement to calculate multiple levels of subtotals across a specified group of dimensions. It also calculates a grand total. ROLLUP is a simple extension to the GROUP BY clause, so its syntax is extremely easy to use. The ROLLUP extension is highly efficient, adding minimal overhead to a query.

The action of ROLLUP is straightforward: it creates subtotals that roll up from the most detailed level to a grand total, following a grouping list specified in the ROLLUP clause. ROLLUP takes as its argument an ordered list of grouping columns. First, it calculates the standard aggregate values specified in the GROUP BY clause. Then, it creates progressively higher-level subtotals, moving from right to left through the list of grouping columns. Finally, it creates a grand total.

ROLLUP creates subtotals at n+1 levels, where n is the number of grouping columns. For instance, if a query specifies ROLLUP on grouping columns of `time`, `region`, and `department` (n=3), the result set will include rows at four aggregation levels.

You might want to compress your data when using ROLLUP. This is particularly useful when there are few updates to older partitions.

When to Use ROLLUP

Use the ROLLUP extension in tasks involving subtotals.

- It is very helpful for subtotaling along a hierarchical dimension such as time or geography. For instance, a query could specify a `ROLLUP(y, m, day)` or `ROLLUP(country, state, city)`.
- For data warehouse administrators using summary tables, ROLLUP can simplify and speed up the maintenance of summary tables.

ROLLUP Syntax

ROLLUP appears in the GROUP BY clause in a SELECT statement. Its form is:

```
SELECT … GROUP BY ROLLUP(grouping_column_reference_list)
```

Example 20–2 ROLLUP

This example uses the data in the sh sample schema data, the same data as was used in Figure 20–1. The ROLLUP is across three dimensions.

```
SELECT channels.channel_desc, calendar_month_desc, countries.country_iso_code,
   TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels, countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND
  sales.channel_id= channels.channel_id AND channels.channel_desc IN
  ('Direct Sales', 'Internet') AND times.calendar_month_desc IN
  ('2000-09', '2000-10') AND countries.country_iso_code IN ('GB', 'US')
GROUP BY ROLLUP(channels.channel_desc, calendar_month_desc,
countries.country_iso_code);

CHANNEL_DESC         CALENDAR CO SALES$
-------------------- -------- -- --------------
Internet             2000-09  GB        228,241
Internet             2000-09  US        228,241
Internet             2000-09            456,482
Internet             2000-10  GB        239,236
Internet             2000-10  US        239,236
Internet             2000-10            478,473
Internet                                934,955
Direct Sales         2000-09  GB      1,217,808
Direct Sales         2000-09  US      1,217,808
Direct Sales         2000-09          2,435,616
Direct Sales         2000-10  GB      1,225,584
Direct Sales         2000-10  US      1,225,584
Direct Sales         2000-10          2,451,169
Direct Sales                          4,886,784
                                      5,821,739
```

Note that results do not always add due to rounding.

This query returns the following sets of rows:

- Regular aggregation rows that would be produced by GROUP BY without using ROLLUP.
- First-level subtotals aggregating across country_id for each combination of channel_desc and calendar_month.

- Second-level subtotals aggregating across `calendar_month_desc` and `country_id` for each `channel_desc` value.
- A grand total row.

Partial Rollup

You can also roll up so that only some of the sub-totals will be included. This partial rollup uses the following syntax:

```
GROUP BY expr1, ROLLUP(expr2, expr3);
```

In this case, the GROUP BY clause creates subtotals at (2+1=3) aggregation levels. That is, at level (`expr1`, `expr2`, `expr3`), (`expr1`, `expr2`), and (`expr1`).

Example 20–3 Partial ROLLUP

```
SELECT channel_desc, calendar_month_desc, countries.country_iso_code,
   TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels, countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND
   sales.channel_id= channels.channel_id AND channels.channel_desc IN
  ('Direct Sales', 'Internet') AND times.calendar_month_desc IN
  ('2000-09', '2000-10') AND countries.country_iso_code IN ('GB', 'US')
GROUP BY channel_desc, ROLLUP(calendar_month_desc, countries.country_iso_code);

CHANNEL_DESC         CALENDAR CO SALES$
-------------------- -------- -- --------------
Internet             2000-09  GB         228,241
Internet             2000-09  US         228,241
Internet             2000-09             456,482
Internet             2000-10  GB         239,236
Internet             2000-10  US         239,236
Internet             2000-10             478,473
Internet                                 934,955
Direct Sales         2000-09  GB       1,217,808
Direct Sales         2000-09  US       1,217,808
Direct Sales         2000-09           2,435,616
Direct Sales         2000-10  GB       1,225,584
Direct Sales         2000-10  US       1,225,584
Direct Sales         2000-10           2,451,169
Direct Sales                           4,886,784
```

This query returns the following sets of rows:

- Regular aggregation rows that would be produced by GROUP BY without using ROLLUP.
- First-level subtotals aggregating across country_id for each combination of channel_desc and calendar_month_desc.
- Second-level subtotals aggregating across calendar_month_desc and country_id for each channel_desc value.
- It does not produce a grand total row.

CUBE Extension to GROUP BY

CUBE takes a specified set of grouping columns and creates subtotals for all of their possible combinations. In terms of multidimensional analysis, CUBE generates all the subtotals that could be calculated for a data cube with the specified dimensions. If you have specified CUBE(time, region, department), the result set will include all the values that would be included in an equivalent ROLLUP statement plus additional combinations. For instance, in Figure 20–1, the departmental totals across regions (279,000 and 319,000) would not be calculated by a ROLLUP(time, region, department) clause, but they would be calculated by a CUBE(time, region, department) clause. If *n* columns are specified for a CUBE, there will be 2 to the *n* combinations of subtotals returned. Example 20–4 on page 20-10 gives an example of a three-dimension cube. See *Oracle Database SQL Reference* for syntax and restrictions.

When to Use CUBE

Consider Using CUBE in any situation requiring cross-tabular reports. The data needed for cross-tabular reports can be generated with a single SELECT using CUBE. Like ROLLUP, CUBE can be helpful in generating summary tables. Note that population of summary tables is even faster if the CUBE query executes in parallel.

CUBE is typically most suitable in queries that use columns from multiple dimensions rather than columns representing different levels of a single dimension. For instance, a commonly requested cross-tabulation might need subtotals for all the combinations of month, state, and product. These are three independent dimensions, and analysis of all possible subtotal combinations is commonplace. In contrast, a cross-tabulation showing all possible combinations of year, month, and day would have several values of limited interest, because there is a natural hierarchy in the time dimension. Subtotals such as profit by day of month summed across year would be unnecessary in most analyses. Relatively few users need to ask "What were the total sales for the 16th of each month across the year?" See

"Hierarchy Handling in ROLLUP and CUBE" on page 20-26 for an example of handling rollup calculations efficiently.

CUBE Syntax

CUBE appears in the GROUP BY clause in a SELECT statement. Its form is:

```
SELECT …  GROUP BY CUBE (grouping_column_reference_list)
```

Example 20–4 CUBE

```
SELECT channel_desc, calendar_month_desc, countries.country_iso_code,
       TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels, countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND
   sales.channel_id= channels.channel_id AND channels.channel_desc IN
  ('Direct Sales', 'Internet') AND times.calendar_month_desc IN
  ('2000-09', '2000-10') AND countries.country_iso_code IN ('GB', 'US')
GROUP BY CUBE(channel_desc, calendar_month_desc, countries.country_iso_code);

CHANNEL_DESC         CALENDAR CO SALES$
-------------------- -------- -- --------------
                                     5,821,739
                              GB     2,910,870
                              US     2,910,870
                     2000-09         2,892,098
                     2000-09  GB     1,446,049
                     2000-09  US     1,446,049
                     2000-10         2,929,641
                     2000-10  GB     1,464,821
                     2000-10  US     1,464,821
Internet                               934,955
Internet                      GB       467,478
Internet                      US       467,478
Internet             2000-09           456,482
Internet             2000-09  GB       228,241
Internet             2000-09  US       228,241
Internet             2000-10           478,473
Internet             2000-10  GB       239,236
Internet             2000-10  US       239,236
Direct Sales                         4,886,784
Direct Sales                  GB     2,443,392
Direct Sales                  US     2,443,392
Direct Sales         2000-09         2,435,616
Direct Sales         2000-09  GB     1,217,808
```

```
Direct Sales         2000-09  US      1,217,808
Direct Sales         2000-10          2,451,169
Direct Sales         2000-10  GB      1,225,584
Direct Sales         2000-10  US      1,225,584
```

This query illustrates CUBE aggregation across three dimensions.

Partial CUBE

Partial CUBE resembles partial ROLLUP in that you can limit it to certain dimensions and precede it with columns outside the CUBE operator. In this case, subtotals of all possible combinations are limited to the dimensions within the cube list (in parentheses), and they are combined with the preceding items in the GROUP BY list.

The syntax for partial CUBE is as follows:

```
GROUP BY expr1, CUBE(expr2, expr3)
```

This syntax example calculates 2*2, or 4, subtotals. That is:

- (expr1, expr2, expr3)
- (expr1, expr2)
- (expr1, expr3)
- (expr1)

Example 20–5 Partial CUBE

Using the sales database, you can issue the following statement:

```
SELECT channel_desc, calendar_month_desc, countries.country_iso_code,
       TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels, countries
WHERE sales.time_id = times.time_id AND sales.cust_id = customers.cust_id AND
  sales.channel_id = channels.channel_id AND channels.channel_desc IN
 ('Direct Sales', 'Internet') AND times.calendar_month_desc IN
 ('2000-09', '2000-10') AND countries.country_iso_code IN ('GB', 'US')
GROUP BY channel_desc, CUBE(calendar_month_desc, countries.country_iso_code);

CHANNEL_DESC         CALENDAR CO SALES$
-------------------- -------- -- --------------
Internet                                934,955
Internet                        GB      467,478
Internet                        US      467,478
Internet             2000-09            456,482
```

```
Internet             2000-09  GB        228,241
Internet             2000-09  US        228,241
Internet             2000-10            478,473
Internet             2000-10  GB        239,236
Internet             2000-10  US        239,236
Direct Sales                            4,886,784
Direct Sales                  GB        2,443,392
Direct Sales                  US        2,443,392
Direct Sales         2000-09            2,435,616
Direct Sales         2000-09  GB        1,217,808
Direct Sales         2000-09  US        1,217,808
Direct Sales         2000-10            2,451,169
Direct Sales         2000-10  GB        1,225,584
Direct Sales         2000-10  US        1,225,584
```

Calculating Subtotals Without CUBE

Just as for ROLLUP, multiple SELECT statements combined with UNION ALL statements could provide the same information gathered through CUBE. However, this might require many SELECT statements. For an n-dimensional cube, 2 to the *n* SELECT statements are needed. In the three-dimension example, this would mean issuing SELECT statements linked with UNION ALL. So many SELECT statements yield inefficient processing and very lengthy SQL.

Consider the impact of adding just one more dimension when calculating all possible combinations: the number of SELECT statements would double to 16. The more columns used in a CUBE clause, the greater the savings compared to the UNION ALL approach.

GROUPING Functions

Two challenges arise with the use of ROLLUP and CUBE. First, how can you programmatically determine which result set rows are subtotals, and how do you find the exact level of aggregation for a given subtotal? You often need to use subtotals in calculations such as percent-of-totals, so you need an easy way to determine which rows are the subtotals. Second, what happens if query results contain both stored NULL values and "NULL" values created by a ROLLUP or CUBE? How can you differentiate between the two? See *Oracle Database SQL Reference* for syntax and restrictions.

GROUPING Function

GROUPING handles these problems. Using a single column as its argument, GROUPING returns 1 when it encounters a NULL value created by a ROLLUP or CUBE operation. That is, if the NULL indicates the row is a subtotal, GROUPING returns a 1. Any other type of value, including a stored NULL, returns a 0.

GROUPING appears in the selection list portion of a SELECT statement. Its form is:

```
SELECT …  [GROUPING(dimension_column)…]  …
  GROUP BY …    {CUBE | ROLLUP| GROUPING SETS}  (dimension_column)
```

Example 20–6 GROUPING to Mask Columns

This example uses GROUPING to create a set of mask columns for the result set shown in Example 20–3 on page 20-8. The mask columns are easy to analyze programmatically.

```
SELECT channel_desc, calendar_month_desc, country_iso_code,
TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$, GROUPING(channel_desc) AS Ch,
   GROUPING(calendar_month_desc) AS Mo, GROUPING(country_iso_code) AS Co
FROM sales, customers, times, channels, countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND
  sales.channel_id= channels.channel_id AND channels.channel_desc IN
 ('Direct Sales', 'Internet') AND times.calendar_month_desc IN
 ('2000-09', '2000-10') AND countries.country_iso_code IN ('GB', 'US')
GROUP BY ROLLUP(channel_desc, calendar_month_desc, countries.country_iso_code);
```

```
CHANNEL_DESC         CALENDAR CO SALES$                CH         MO         CO
-------------------- -------- -- -------------- ---------- ---------- ----------
Internet             2000-09  GB         228,241          0          0          0
Internet             2000-09  US         228,241          0          0          0
Internet             2000-09             456,482          0          0          1
Internet             2000-10  GB         239,236          0          0          0
Internet             2000-10  US         239,236          0          0          0
Internet             2000-10             478,473          0          0          1
Internet                                 934,955          0          1          1
Direct Sales         2000-09  GB       1,217,808          0          0          0
Direct Sales         2000-09  US       1,217,808          0          0          0
Direct Sales         2000-09             2,435,616          0          0          1
Direct Sales         2000-10  GB       1,225,584          0          0          0
Direct Sales         2000-10  US       1,225,584          0          0          0
Direct Sales         2000-10             2,451,169          0          0          1
Direct Sales                             4,886,784          0          1          1
                                         5,821,739          1          1          1
```

A program can easily identify the detail rows by a mask of "0 0 0" on the T, R, and D columns. The first level subtotal rows have a mask of "0 0 1", the second level subtotal rows have a mask of "0 1 1", and the overall total row has a mask of "1 1 1".

You can improve the readability of result sets by using the GROUPING and DECODE functions as shown in Example 20–7.

Example 20–7 GROUPING For Readability

```
SELECT DECODE(GROUPING(channel_desc), 1, 'Multi-channel sum', channel_desc) AS
 Channel, DECODE (GROUPING (country_iso_code), 1, 'Multi-country sum',
 country_iso_code) AS Country, TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels, countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id
  AND sales.channel_id= channels.channel_id AND channels.channel_desc
IN ('Direct Sales', 'Internet') AND times.calendar_month_desc= '2000-09'
  AND country_iso_code IN ('GB', 'US')
GROUP BY CUBE(channel_desc, country_iso_code);

CHANNEL              COUNTRY           SALES$
-------------------- ---------------- -------------
Multi-channel sum    Multi-country sum      2,892,098
Multi-channel sum    GB                     1,446,049
Multi-channel sum    US                     1,446,049
Internet             Multi-country sum        456,482
Internet             GB                       228,241
Internet             US                       228,241
Direct Sales         Multi-country sum      2,435,616
Direct Sales         GB                     1,217,808
Direct Sales         US                     1,217,808
```

To understand the previous statement, note its first column specification, which handles the channel_desc column. Consider the first line of the previous statement:

```
SELECT DECODE(GROUPING(channel_desc), 1, 'All Channels', channel_desc)AS Channel
```

In this, the channel_desc value is determined with a DECODE function that contains a GROUPING function. The GROUPING function returns a 1 if a row value is an aggregate created by ROLLUP or CUBE, otherwise it returns a 0. The DECODE function then operates on the GROUPING function's results. It returns the text "All Channels" if it receives a 1 and the channel_desc value from the database if it receives a 0. Values from the database will be either a real value such as "Internet" or a stored NULL. The second column specification, displaying country_id, works the same way.

When to Use GROUPING

The GROUPING function is not only useful for identifying NULLs, it also enables sorting subtotal rows and filtering results. In Example 20–8, you retrieve a subset of the subtotals created by a CUBE and none of the base-level aggregations. The HAVING clause constrains columns that use GROUPING functions.

Example 20–8 GROUPING Combined with HAVING

```
SELECT channel_desc, calendar_month_desc, country_iso_code, TO_CHAR(
SUM(amount_sold), '9,999,999,999') SALES$, GROUPING(channel_desc) CH, GROUPING
  (calendar_month_desc)  MO, GROUPING(country_iso_code) CO
FROM sales, customers, times, channels, countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND
  sales.channel_id= channels.channel_id AND channels.channel_desc IN
 ('Direct Sales', 'Internet') AND times.calendar_month_desc IN
 ('2000-09', '2000-10') AND country_iso_code IN ('GB', 'US')
GROUP BY CUBE(channel_desc, calendar_month_desc, country_iso_code)
HAVING (GROUPING(channel_desc)=1 AND GROUPING(calendar_month_desc)= 1 AND
  GROUPING(country_iso_code)=1) OR (GROUPING(channel_desc)=1 AND
  GROUPING (calendar_month_desc)= 1) OR (GROUPING(country_iso_code)=1
  AND GROUPING(calendar_month_desc)= 1);

CHANNEL_DESC         C CO SALES$                CH         MO         CO
-------------------- - -- -------------- ---------- ---------- ----------
                       GB      2,910,870          1          1          0
                       US      2,910,870          1          1          0
Direct Sales                   4,886,784          0          1          1
Internet                         934,955          0          1          1
                               5,821,739          1          1          1
```

Compare the result set of Example 20–8 with that in Example 20–3 on page 20-8 to see how Example 20–8 is a precisely specified group: it contains only the yearly totals, regional totals aggregated over time and department, and the grand total.

GROUPING_ID Function

To find the GROUP BY level of a particular row, a query must return GROUPING function information for each of the GROUP BY columns. If we do this using the GROUPING function, every GROUP BY column requires another column using the GROUPING function. For instance, a four-column GROUP BY clause needs to be analyzed with four GROUPING functions. This is inconvenient to write in SQL and increases the number of columns required in the query. When you want to store the

query result sets in tables, as with materialized views, the extra columns waste storage space.

To address these problems, you can use the GROUPING_ID function. GROUPING_ID returns a single number that enables you to determine the exact GROUP BY level. For each row, GROUPING_ID takes the set of 1's and 0's that would be generated if you used the appropriate GROUPING functions and concatenates them, forming a bit vector. The bit vector is treated as a binary number, and the number's base-10 value is returned by the GROUPING_ID function. For instance, if you group with the expression CUBE(a, b) the possible values are as shown in Table 20–2.

Table 20–2 GROUPING_ID Example for CUBE(a, b)

Aggregation Level	Bit Vector	GROUPING_ID
a, b	0 0	0
a	0 1	1
b	1 0	2
Grand Total	1 1	3

GROUPING_ID clearly distinguishes groupings created by grouping set specification, and it is very useful during refresh and rewrite of materialized views.

GROUP_ID Function

While the extensions to GROUP BY offer power and flexibility, they also allow complex result sets that can include duplicate groupings. The GROUP_ID function lets you distinguish among duplicate groupings. If there are multiple sets of rows calculated for a given level, GROUP_ID assigns the value of 0 to all the rows in the first set. All other sets of duplicate rows for a particular grouping are assigned higher values, starting with 1. For example, consider the following query, which generates a duplicate grouping:

Example 20–9 GROUP_ID

```
SELECT country_iso_code, SUBSTR(cust_state_province,1,12), SUM(amount_sold),
  GROUPING_ID(country_iso_code, cust_state_province) GROUPING_ID, GROUP_ID()
FROM sales, customers, times, countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id
  AND customers.country_id=countries.country_id AND times.time_id= '30-OCT-00'
  AND country_iso_code IN ('FR', 'ES')
GROUP BY GROUPING SETS (country_iso_code,
```

```
ROLLUP(country_iso_code, cust_state_province));

CO SUBSTR(CUST_ SUM(AMOUNT_SOLD) GROUPING_ID GROUP_ID()
-- ------------ --------------- ----------- ----------
ES Alicante              135.32           0          0
ES Valencia             4133.56           0          0
ES Barcelona              24.22           0          0
FR Centre                  74.3           0          0
FR Aquitaine             231.97           0          0
FR Rhtne-Alpes          1624.69           0          0
FR Ile-de-Franc         1860.59           0          0
FR Languedoc-Ro          4287.4           0          0
                        12372.05           3          0
ES                        4293.1           1          0
FR                       8078.95           1          0
ES                        4293.1           1          1
FR                       8078.95           1          1
```

This query generates the following groupings: (`country_id`, `cust_state_province`), (`country_id`), (`country_id`), and (). Note that the grouping (`country_id`) is repeated twice. The syntax for `GROUPING SETS` is explained in "GROUPING SETS Expression" on page 20-17.

This function helps you filter out duplicate groupings from the result. For example, you can filter out duplicate (`region`) groupings from the previous example by adding a `HAVING` clause condition `GROUP_ID()=0` to the query.

GROUPING SETS Expression

You can selectively specify the set of groups that you want to create using a `GROUPING SETS` expression within a `GROUP BY` clause. This allows precise specification across multiple dimensions without computing the whole `CUBE`. For example, you can say:

```
SELECT channel_desc, calendar_month_desc, country_iso_code,
       TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels, countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND
  sales.channel_id= channels.channel_id AND channels.channel_desc IN
 ('Direct Sales', 'Internet') AND times.calendar_month_desc IN
 ('2000-09', '2000-10') AND country_iso_code IN ('GB', 'US')
GROUP BY GROUPING SETS((channel_desc, calendar_month_desc, country_iso_code),
    (channel_desc, country_iso_code), (calendar_month_desc, country_iso_code));
```

Note that this statement uses composite columns, described in "Composite Columns" on page 20-20. This statement calculates aggregates over three groupings:

- (channel_desc, calendar_month_desc, country_iso_code)
- (channel_desc, country_iso_code)
- (calendar_month_desc, country_iso_code)

Compare the previous statement with the following alternative, which uses the CUBE operation and the GROUPING_ID function to return the desired rows:

```
SELECT channel_desc, calendar_month_desc, country_iso_code,
       TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$,
       GROUPING_ID(channel_desc, calendar_month_desc, country_iso_code) gid
FROM sales, customers, times, channels, countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND
  sales.channel_id= channels.channel_id AND channels.channel_desc IN
 ('Direct Sales', 'Internet') AND times.calendar_month_desc IN
 ('2000-09', '2000-10') AND country_iso_code IN ('GB', 'US')
GROUP BY CUBE(channel_desc, calendar_month_desc, country_iso_code)
HAVING GROUPING_ID(channel_desc, calendar_month_desc, country_iso_code)=0
  OR GROUPING_ID(channel_desc, calendar_month_desc, country_iso_code)=2
  OR GROUPING_ID(channel_desc, calendar_month_desc, country_iso_code)=4;
```

This statement computes all the 8 (2 *2 *2) groupings, though only the previous 3 groups are of interest to you.

Another alternative is the following statement, which is lengthy due to several unions. This statement requires three scans of the base table, making it inefficient. CUBE and ROLLUP can be thought of as grouping sets with very specific semantics. For example, consider the following statement:

```
CUBE(a, b, c)
```

This statement is equivalent to:

```
GROUPING SETS ((a, b, c), (a, b), (a, c), (b, c), (a), (b), (c), ())
ROLLUP(a, b, c)
```

And this statement is equivalent to:

```
GROUPING SETS ((a, b, c), (a, b), ())
```

GROUPING SETS Syntax

GROUPING SETS syntax lets you define multiple groupings in the same query. GROUP BY computes all the groupings specified and combines them with UNION ALL. For example, consider the following statement:

```
GROUP BY GROUPING sets (channel_desc, calendar_month_desc, country_id )
```

This statement is equivalent to:

```
GROUP BY channel_desc UNION ALL
GROUP BY calendar_month_desc UNION ALL GROUP BY country_id
```

Table 20–3 shows grouping sets specification and equivalent GROUP BY specification. Note that some examples use composite columns.

Table 20–3 GROUPING SETS Statements and Equivalent GROUP BY

GROUPING SETS Statement	Equivalent GROUP BY Statement
`GROUP BY GROUPING SETS(a, b, c)`	`GROUP BY a UNION ALL GROUP BY b UNION ALL GROUP BY c`
`GROUP BY GROUPING SETS(a, b, (b, c))`	`GROUP BY a UNION ALL GROUP BY b UNION ALL GROUP BY b, c`
`GROUP BY GROUPING SETS((a, b, c))`	`GROUP BY a, b, c`
`GROUP BY GROUPING SETS(a, (b), ())`	`GROUP BY a UNION ALL GROUP BY b UNION ALL GROUP BY ()`
`GROUP BY GROUPING SETS(a, ROLLUP(b, c))`	`GROUP BY a UNION ALL GROUP BY ROLLUP(b, c)`

In the absence of an optimizer that looks across query blocks to generate the execution plan, a query based on UNION would need multiple scans of the base table, sales. This could be very inefficient as fact tables will normally be huge. Using GROUPING SETS statements, all the groupings of interest are available in the same query block.

Composite Columns

A composite column is a collection of columns that are treated as a unit during the computation of groupings. You specify the columns in parentheses as in the following statement:

```
ROLLUP (year, (quarter, month), day)
```

In this statement, the data is not rolled up across year and quarter, but is instead equivalent to the following groupings of a UNION ALL:

- (year, quarter, month, day),
- (year, quarter, month),
- (year)
- ()

Here, (quarter, month) form a composite column and are treated as a unit. In general, composite columns are useful in ROLLUP, CUBE, GROUPING SETS, and concatenated groupings. For example, in CUBE or ROLLUP, composite columns would mean skipping aggregation across certain levels. That is, the following statement:

```
GROUP BY ROLLUP(a, (b, c))
```

This is equivalent to:

```
GROUP BY a, b, c UNION ALL
GROUP BY a UNION ALL
GROUP BY ()
```

Here, (b, c) are treated as a unit and rollup will not be applied across (b, c). It is as if you have an alias, for example z, for (b, c) and the GROUP BY expression reduces to GROUP BY ROLLUP(a, z). Compare this with the normal rollup as in the following:

```
GROUP BY ROLLUP(a, b, c)
```

This would be the following:

```
GROUP BY a, b, c UNION ALL
GROUP BY a, b UNION ALL
GROUP BY a UNION ALL
GROUP BY ().
```

Similarly, the following statement is equivalent to the four GROUP BYs:

```
GROUP BY CUBE((a, b), c)

GROUP BY a, b, c UNION ALL
GROUP BY a, b UNION ALL
GROUP BY c UNION ALL
GROUP By ()
```

In GROUPING SETS, a composite column is used to denote a particular level of GROUP BY. See Table 20–3 for more examples of composite columns.

Example 20–10 Composite Columns

You do not have full control over what aggregation levels you want with CUBE and ROLLUP. For example, the following statement:

```
SELECT channel_desc, calendar_month_desc, country_iso_code,
 TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels, countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND
 sales.channel_id= channels.channel_id AND channels.channel_desc IN ('Direct
Sales', 'Internet') AND times.calendar_month_desc IN ('2000-09', '2000-10')
AND country_iso_code IN ('GB', 'US')
GROUP BY ROLLUP(channel_desc, calendar_month_desc, country_iso_code);
```

This statement results in Oracle computing the following groupings:

- (channel_desc, calendar_month_desc, country_iso_code)
- (channel_desc, calendar_month_desc)
- (channel_desc)
- ()

If you are just interested in grouping of lines (1), (3) and (4) in this example, you cannot limit the calculation to those groupings without using composite columns. With composite columns, this is possible by treating month and country as a single unit while rolling up. Columns enclosed in parentheses are treated as a unit while computing CUBE and ROLLUP. Thus, you would say:

```
SELECT channel_desc, calendar_month_desc, country_iso_code,
    TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels, countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND
      sales.channel_id= channels.channel_id  AND channels.channel_desc IN
 ('Direct Sales', 'Internet') AND times.calendar_month_desc IN
```

```
 ('2000-09', '2000-10') AND country_iso_code IN ('GB', 'US')
GROUP BY ROLLUP(channel_desc, (calendar_month_desc, country_iso_code));

CHANNEL_DESC         CALENDAR CO SALES$
-------------------- -------- -- --------------
Internet             2000-09  GB        228,241
Internet             2000-09  US        228,241
Internet             2000-10  GB        239,236
Internet             2000-10  US        239,236
Internet                                934,955
Direct Sales         2000-09  GB      1,217,808
Direct Sales         2000-09  US      1,217,808
Direct Sales         2000-10  GB      1,225,584
Direct Sales         2000-10  US      1,225,584
Direct Sales                         4,886,784
                                     5,821,739
```

Concatenated Groupings

Concatenated groupings offer a concise way to generate useful combinations of groupings. Groupings specified with concatenated groupings yield the cross-product of groupings from each grouping set. The cross-product operation enables even a small number of concatenated groupings to generate a large number of final groups. The concatenated groupings are specified simply by listing multiple grouping sets, cubes, and rollups, and separating them with commas. Here is an example of concatenated grouping sets:

```
GROUP BY GROUPING SETS(a, b), GROUPING SETS(c, d)
```

This SQL defines the following groupings:

```
(a, c), (a, d), (b, c), (b, d)
```

Concatenation of grouping sets is very helpful for these reasons:

- Ease of query development

  You need not enumerate all groupings manually.

- Use by applications

  SQL generated by OLAP applications often involves concatenation of grouping sets, with each grouping set defining groupings needed for a dimension.

Example 20–11 Concatenated Groupings

You can also specify more than one grouping in the GROUP BY clause. For example, if you want aggregated sales values for each product rolled up across all levels in the time dimension (year, month and day), and across all levels in the geography dimension (region), you can issue the following statement:

```
SELECT channel_desc, calendar_year, calendar_quarter_desc, country_iso_code,
  cust_state_province, TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels, countries
WHERE sales.time_id = times.time_id AND sales.cust_id = customers.cust_id
 AND sales.channel_id = channels.channel_id AND countries.country_id =
    customers.country_id AND channels.channel_desc IN
   ('Direct Sales', 'Internet') AND times.calendar_month_desc IN ('2000-09',
 '2000-10') AND countries.country_iso_code IN ('GB', 'FR')
GROUP BY channel_desc, GROUPING SETS (ROLLUP(calendar_year,
   calendar_quarter_desc),
ROLLUP(country_iso_code, cust_state_province));
```

This results in the following groupings:

- (channel_desc, calendar_year, calendar_quarter_desc)
- (channel_desc, calendar_year)
- (channel_desc)
- (channel_desc, country_iso_code, cust_state_province)
- (channel_desc, country_iso_code)
- (channel_desc)

This is the cross-product of the following:

- The expression, channel_desc
- ROLLUP(calendar_year, calendar_quarter_desc), which is equivalent to ((calendar_year, calendar_quarter_desc), (calendar_year), ())
- ROLLUP(country_iso_code, cust_state_province), which is equivalent to ((country_iso_code, cust_state_province), (country_iso_code), ())

Note that the output contains two occurrences of (channel_desc) group. To filter out the extra (channel_desc) group, the query could use a GROUP_ID function.

Another concatenated join example is the following, showing the cross product of two grouping sets:

Example 20–12 Concatenated Groupings (Cross-Product of Two Grouping Sets)

```
SELECT country_iso_code, cust_state_province, calendar_year, calendar_quarter_
desc, TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels, countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND
 countries.country_id=customers.country_id  AND
  sales.channel_id= channels.channel_id AND channels.channel_desc IN
 ('Direct Sales', 'Internet') AND times.calendar_month_desc IN
 ('2000-09', '2000-10') AND country_iso_code IN ('GB', 'FR')
GROUP BY GROUPING SETS (country_iso_code, cust_state_province),
         GROUPING SETS (calendar_year, calendar_quarter_desc);
```

This statement results in the computation of groupings:

- (country_iso_code, year), (country_iso_code, calendar_quarter_desc), (cust_state_province, year) and (cust_state_province, calendar_quarter_desc)

Concatenated Groupings and Hierarchical Data Cubes

One of the most important uses for concatenated groupings is to generate the aggregates needed for a hierarchical cube of data. A hierarchical cube is a data set where the data is aggregated along the rollup hierarchy of each of its dimensions and these aggregations are combined across dimensions. It includes the typical set of aggregations needed for business intelligence queries. By using concatenated groupings, you can generate all the aggregations needed by a hierarchical cube with just *n* ROLLUPs (where *n* is the number of dimensions), and avoid generating unwanted aggregations.

Consider just three of the dimensions in the sh sample schema data set, each of which has a multilevel hierarchy:

- time: year, quarter, month, day (week is in a separate hierarchy)
- product: category, subcategory, prod_name
- geography: region, subregion, country, state, city

This data is represented using a column for each level of the hierarchies, creating a total of twelve columns for dimensions, plus the columns holding sales figures.

For our business intelligence needs, we would like to calculate and store certain aggregates of the various combinations of dimensions. In Example 20–13 on page 20-25, we create the aggregates for all levels, except for "day", which would create too many rows. In particular, we want to use ROLLUP within each dimension

to generate useful aggregates. Once we have the ROLLUP-based aggregates within each dimension, we want to combine them with the other dimensions. This will generate our hierarchical cube. Note that this is not at all the same as a CUBE using all twelve of the dimension columns: that would create 2 to the 12th power (4,096) aggregation groups, of which we need only a small fraction. Concatenated grouping sets make it easy to generate exactly the aggregations we need. Example 20–13 shows where a GROUP BY clause is needed.

Example 20–13 Concatenated Groupings and Hierarchical Cubes

```
SELECT calendar_year, calendar_quarter_desc, calendar_month_desc,
  country_region, country_subregion, countries.country_iso_code, cust_state_
province,
  cust_city, prod_category_desc, prod_subcategory_desc, prod_name, TO_CHAR(SUM
 (amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels, countries, products
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND
  sales.channel_id= channels.channel_id AND sales.prod_id=products.prod_id AND
  customers.country_id=countries.country_id AND channels.channel_desc IN
 ('Direct Sales', 'Internet') AND times.calendar_month_desc IN
 ('2000-09', '2000-10') AND prod_name IN ('Envoy Ambassador',
 'Mouse Pad') AND countries.country_iso_code IN ('GB', 'US')
GROUP BY ROLLUP(calendar_year, calendar_quarter_desc, calendar_month_desc),
  ROLLUP(country_region, country_subregion, countries.country_iso_code,
         cust_state_province, cust_city),
  ROLLUP(prod_category_desc, prod_subcategory_desc, prod_name);
```

The ROLLUPs in the GROUP BY specification generate the following groups, four for each dimension.

Table 20–4 Hierarchical CUBE Example

ROLLUP By Time	ROLLUP By Product	ROLLUP By Geography
year, quarter, month	category, subcategory, name	region, subregion, country, state, city
		region, subregion, country, state
		region, subregion, country
year, quarter	category, subcategory	region, subregion
year	category	region
all times	all products	all geographies

The concatenated grouping sets specified in the previous SQL will take the ROLLUP aggregations listed in the table and perform a cross-product on them. The cross-product will create the 96 (4x4x6) aggregate groups needed for a hierarchical cube of the data. There are major advantages in using three ROLLUP expressions to replace what would otherwise require 96 grouping set expressions: the concise SQL is far less error-prone to develop and far easier to maintain, and it enables much better query optimization. You can picture how a cube with more dimensions and more levels would make the use of concatenated groupings even more advantageous.

See "Working with Hierarchical Cubes in SQL" on page 20-29 for more information regarding hierarchical cubes.

Considerations when Using Aggregation

This section discusses the following topics.

Hierarchy Handling in ROLLUP and CUBE

The ROLLUP and CUBE extensions work independently of any hierarchy metadata in your system. Their calculations are based entirely on the columns specified in the SELECT statement in which they appear. This approach enables CUBE and ROLLUP to be used whether or not hierarchy metadata is available. The simplest way to handle levels in hierarchical dimensions is by using the ROLLUP extension and indicating levels explicitly through separate columns. The following code shows a simple example of this with months rolled up to quarters and quarters rolled up to years.

Example 20–14 ROLLUP and CUBE Hierarchy Handling

```
SELECT calendar_year, calendar_quarter_number,
    calendar_month_number, SUM(amount_sold)
FROM sales, times, products, customers, countries
WHERE sales.time_id=times.time_id AND sales.prod_id=products.prod_id AND
  sales.cust_id=customers.cust_id AND prod_name IN ('Envoy Ambassador',
```

```
 'Mouse Pad') AND country_iso_code = 'GB' AND calendar_year=1999
GROUP BY ROLLUP(calendar_year, calendar_quarter_number, calendar_month_number);

CALENDAR_YEAR CALENDAR_QUARTER_NUMBER CALENDAR_MONTH_NUMBER SUM(AMOUNT_SOLD)
------------- ---------------------- --------------------- ---------------
         1999                      1                     1       168419.74
         1999                      1                     2       332348.02
         1999                      1                     3       169511.52
         1999                      1                             670279.28
         1999                      2                     4       247291.88
         1999                      2                     5       182338.86
         1999                      2                     6       264493.91
         1999                      2                             694124.65
         1999                      3                     7       192268.11
         1999                      3                     8       182550.88
         1999                      3                     9       270309.26
         1999                      3                             645128.25
         1999                      4                    10       180400.32
         1999                      4                    11       232830.87
         1999                      4                    12       168008.07
         1999                      4                             581239.26
         1999                                                   2590771.44
```

Column Capacity in ROLLUP and CUBE

CUBE, ROLLUP, and GROUPING SETS do not restrict the GROUP BY clause column capacity. The GROUP BY clause, with or without the extensions, can work with up to 255 columns. However, the combinatorial explosion of CUBE makes it unwise to specify a large number of columns with the CUBE extension. Consider that a 20-column list for CUBE would create 2 to the 20 combinations in the result set. A very large CUBE list could strain system resources, so any such query needs to be tested carefully for performance and the load it places on the system.

HAVING Clause Used with GROUP BY Extensions

The HAVING clause of SELECT statements is unaffected by the use of GROUP BY. Note that the conditions specified in the HAVING clause apply to both the subtotal and non-subtotal rows of the result set. In some cases a query may need to exclude the subtotal rows or the non-subtotal rows from the HAVING clause. This can be achieved by using a GROUPING or GROUPING_ID function together with the HAVING clause. See Example 20–8 on page 20-15 and its associated SQL statement for an example.

ORDER BY Clause Used with GROUP BY Extensions

In many cases, a query needs to order the rows in a certain way, and this is done with the ORDER BY clause. The ORDER BY clause of a SELECT statement is unaffected by the use of GROUP BY, since the ORDER BY clause is applied after the GROUP BY calculations are complete.

Note that the ORDER BY specification makes no distinction between aggregate and non-aggregate rows of the result set. For instance, you might wish to list sales figures in declining order, but still have the subtotals at the end of each group. Simply ordering sales figures in descending sequence will not be sufficient, since that will place the subtotals (the largest values) at the start of each group. Therefore, it is essential that the columns in the ORDER BY clause include columns that differentiate aggregate from non-aggregate columns. This requirement means that queries using ORDER BY along with aggregation extensions to GROUP BY will generally need to use one or more of the GROUPING functions.

Using Other Aggregate Functions with ROLLUP and CUBE

The examples in this chapter show ROLLUP and CUBE used with the SUM function. While this is the most common type of aggregation, these extensions can also be used with all other functions available to the GROUP BY clause, for example, COUNT, AVG, MIN, MAX, STDDEV, and VARIANCE. COUNT, which is often needed in cross-tabular analyses, is likely to be the second most commonly used function.

Computation Using the WITH Clause

The WITH clause (formally known as subquery_factoring_clause) enables you to reuse the same query block in a SELECT statement when it occurs more than once within a complex query. WITH is a part of the SQL-99 standard. This is particularly useful when a query has multiple references to the same query block and there are joins and aggregations. Using the WITH clause, Oracle retrieves the results of a query block and stores them in the user's temporary tablespace. Note that Oracle Database does not support recursive use of the WITH clause.

The following query is an example of where you can improve performance and write SQL more simply by using the WITH clause. The query calculates the sum of sales for each channel and holds it under the name channel_summary. Then it checks each channel's sales total to see if any channel's sales are greater than one third of the total sales. By using the WITH clause, the channel_summary data is calculated just once, avoiding an extra scan through the large sales table.

Example 20–15 WITH Clause

```
WITH channel_summary AS (SELECT channels.channel_desc, SUM(amount_sold)
AS channel_total FROM sales, channels
WHERE sales.channel_id = channels.channel_id GROUP BY channels.channel_desc)
SELECT channel_desc, channel_total
FROM channel_summary WHERE channel_total > (SELECT SUM(channel_total) * 1/3
FROM channel_summary);

CHANNEL_DESC         CHANNEL_TOTAL
-------------------- -------------
Direct Sales            57875260.6
```

Note that this example could also be performed efficiently using the reporting aggregate functions described in Chapter 21, "SQL for Analysis and Reporting".

Working with Hierarchical Cubes in SQL

This section illustrates examples of working with hierarchical cubes.

Specifying Hierarchical Cubes in SQL

Oracle Database can specify hierarchical cubes in a simple and efficient SQL query. These hierarchical cubes represent the logical cubes referred to in many OLAP products. To specify data in the form of hierarchical cubes, you can use one of the extensions to the GROUP BY clause, concatenated grouping sets, to generate the aggregates needed for a hierarchical cube of data. By using concatenated rollup (rolling up along the hierarchy of each dimension and then concatenate them across multiple dimensions), you can generate all the aggregations needed by a hierarchical cube.

Example 20–16 Concatenated ROLLUP

The following shows the GROUP BY clause needed to create a hierarchical cube for a 2-dimensional example similar to 2013. The following simple syntax performs a concatenated rollup:

```
GROUP BY ROLLUP(year, quarter, month), ROLLUP(Division, brand, item)
```

This concatenated rollup takes the ROLLUP aggregations similar to those listed in Table 20–4, "Hierarchical CUBE Example" in the prior section and performs a cross-product on them. The cross-product will create the 16 (4x4) aggregate groups needed for a hierarchical cube of the data.

Querying Hierarchical Cubes in SQL

Analytic applications treat data as cubes, but they want only certain slices and regions of the cube. Concatenated rollup (hierarchical cube) enables relational data to be treated as cubes. To handle complex analytic queries, the fundamental technique is to enclose a hierarchical cube query in an outer query that specifies the exact slice needed from the cube. Oracle Database optimizes the processing of hierarchical cubes nested inside slicing queries. By applying many powerful algorithms, these queries can be processed at unprecedented speed and scale. This enables OLAP tools and analytic applications to use a consistent style of queries to handle the most complex questions.

Example 20–17 Hierarchical Cube Query

Consider the following analytic query. It consists of a hierarchical cube query nested in a slicing query.

```
SELECT month, division, sum_sales FROM
  (SELECT year, quarter, month, division, brand, item, SUM(sales) sum_sales,
      GROUPING_ID(grouping-columns) gid
   FROM sales, products, time
   WHERE join-condition
   GROUP BY ROLLUP(year, quarter, month),
            ROLLUP(division, brand, item))
WHERE division = 25 AND month = 200201 AND gid = gid-for-Division-Month;
```

The inner hierarchical cube specified defines a simple cube, with two dimensions and four levels in each dimension. It would generate 16 groups (4 Time levels * 4 Product levels). The `GROUPING_ID` function in the query identifies the specific group each row belongs to, based on the aggregation level of the *grouping-columns* in its argument.

The outer query applies the constraints needed for our specific query, limiting Division to a value of 25 and Month to a value of 200201 (representing January 2002 in this case). In conceptual terms, it slices a small chunk of data from the cube. The outer query's constraint on the `GID` column, indicated in the query by *gid-for-division-month* would be the value of a key indicating that the data is grouped as a combination of `division` and `month`. The `GID` constraint selects only those rows that are aggregated at the level of a `GROUP BY` month, division clause.

Oracle Database removes unneeded aggregation groups from query processing based on the outer query conditions. The outer conditions of the previous query limit the result set to a single group aggregating `division` and `month`. Any other

groups involving year, month, brand, and item are unnecessary here. The group pruning optimization recognizes this and transforms the query into:

```
SELECT month, division, sum_sales
FROM (SELECT  null, null,  month, division, null, null, SUM(sales) sum_sales,
      GROUPING_ID(grouping-columns) gid
      FROM sales, products, time WHERE join-condition
   GROUP BY month, division)
WHERE division = 25 AND month = 200201 AND gid = gid-for-Division-Month;
```

The bold items highlight the changed SQL. The inner query now has a simple GROUP BY clause of month, division. The columns year, quarter, brand and item have been converted to null to match the simplified GROUP BY clause. Because the query now requests just one group, fifteen out of sixteen groups are removed from the processing, greatly reducing the work. For a cube with more dimensions and more levels, the savings possible through group pruning can be far greater. Note that the group pruning transformation works with all the GROUP BY extensions: ROLLUP, CUBE, and GROUPING SETS.

While the optimizer has simplified the previous query to a simple GROUP BY, faster response times can be achieved if the group is precomputed and stored in a materialized view. Because OLAP queries can ask for any slice of the cube many groups may need to be precomputed and stored in a materialized view. This is discussed in the next section.

SQL for Creating Materialized Views to Store Hierarchical Cubes

OLAP requires fast response times for multiple users, and this in turn demands that significant parts of an OLAP cube be precomputed and held in materialized views. The Oracle Database enables great flexibility in the use of materialized views for OLAP.

Data warehouse designers can choose exactly how much data to materialize. A data warehouse can have the full hierarchical cube materialized. While this will take the most storage space, it ensures quick response for any query within the cube. On the other hand, a data warehouse could have just partial materialization, saving storage space, but allowing only a subset of possible queries to be answered at highest speed. If an OLAP environment's queries cover the full range of aggregate groupings possible in its data set, it may be best to materialize the whole hierarchical cube.

This means that each dimension's aggregation hierarchy is precomputed in combination with each of the other dimensions. Naturally, precomputing a full hierarchical cube requires more disk space and higher creation and refresh times

than a small set of aggregate groups. The trade-off in processing time and disk space versus query performance needs to be considered before deciding to create it. An additional possibility you could consider is to use data compression to lessen your disk space requirements.

See *Oracle Database SQL Reference* for compression syntax and restrictions and "Storage And Table Compression" on page 8-22 for details regarding compression.

Examples of Hierarchical Cube Materialized Views

This section shows complete and partial hierarchical cube materialized views. Many of the examples are meant to illustrate capabilities, and do not actually run.

In a data warehouse where rolling window scenario is very common, it is recommended that you store the hierarchical cube in multiple materialized views - one for each level of time you are interested in. Hence, a complete hierarchical cube will be stored in four materialized views: `sales_hierarchical_mon_cube_mv`, `sales_hierarchical_qtr_cube_mv`, `sales_hierarchical_yr_cube_mv`, and `sales_hierarchical_all_cube_mv`.

The following statements create a complete hierarchical cube stored in a set of three composite partitioned and one list partitioned materialized view.

Example 20–18 Complete Hierarchical Cube Materialized View

```
CREATE MATERIALIZED VIEW sales_hierarchical_mon_cube_mv
PARTITION BY RANGE (mon)
SUBPARTITION BY LIST (gid)
REFRESH FAST ON DEMAND
ENABLE QUERY REWRITE AS
SELECT calendar_year yr, calendar_quarter_desc qtr, calendar_month_desc mon,
    country_id, cust_state_province, cust_city,
    prod_category, prod_subcategory, prod_name,
    GROUPING_ID(calendar_year, calendar_quarter_desc, calendar_month_desc,
                country_id, cust_state_province, cust_city,
                prod_category, prod_subcategory, prod_name) gid,
    SUM(amount_sold) s_sales, COUNT(amount_sold) c_sales,
    COUNT(*) c_star
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id AND s.time_id = t.time_id
GROUP BY calendar_year, calendar_quarter_desc, calendar_month_desc,
  ROLLUP(country_id, cust_state_province, cust_city),
  ROLLUP(prod_category, prod_subcategory, prod_name),
...;
```

```
CREATE MATERIALIZED VIEW sales_hierarchical_qtr_cube_mv
REFRESH FAST ON DEMAND
ENABLE QUERY REWRITE AS
SELECT calendar_year yr, calendar_quarter_desc qtr,
    country_id, cust_state_province, cust_city,
    prod_category, prod_subcategory, prod_name,
    GROUPING_ID(calendar_year, calendar_quarter_desc,
                country_id, cust_state_province, cust_city,
                prod_category, prod_subcategory, prod_name) gid,
    SUM(amount_sold) s_sales, COUNT(amount_sold) c_sales,
    COUNT(*) c_star
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id
      AND s.time_id = t.time_id
GROUP BY calendar_year, calendar_quarter_desc,
  ROLLUP(country_id, cust_state_province, cust_city),
  ROLLUP(prod_category, prod_subcategory, prod_name),
PARTITION BY RANGE (qtr)
 SUBPARTITION BY LIST (gid)
...;

CREATE MATERIALIZED VIEW sales_hierarchical_yr_cube_mv
PARTITION BY RANGE (year)
SUBPARTITION BY LIST (gid)
REFRESH FAST ON DEMAND
ENABLE QUERY REWRITE AS
SELECT calendar_year yr, country_id, cust_state_province, cust_city,
    prod_category, prod_subcategory, prod_name,
    GROUPING_ID(calendar_year, country_id, cust_state_province, cust_city,
                prod_category, prod_subcategory, prod_name) gid,
    SUM(amount_sold) s_sales, COUNT(amount_sold) c_sales, COUNT(*) c_star
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id AND s.time_id = t.time_id
GROUP BY calendar_year,
  ROLLUP(country_id, cust_state_province, cust_city),
  ROLLUP(prod_category, prod_subcategory, prod_name),
...;

CREATE MATERIALIZED VIEW sales_hierarchical_all_cube_mv
REFRESH FAST ON DEMAND
ENABLE QUERY REWRITE AS
SELECT country_id, cust_state_province, cust_city,
    prod_category, prod_subcategory, prod_name,
    GROUPING_ID(country_id, cust_state_province, cust_city,
                prod_category, prod_subcategory, prod_name) gid,
```

```
    SUM(amount_sold) s_sales, COUNT(amount_sold) c_sales, COUNT(*) c_star
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id AND s.time_id = t.time_id
GROUP BY ROLLUP(country_id, cust_state_province, cust_city),
         ROLLUP(prod_category, prod_subcategory, prod_name),
PARTITION BY LIST (gid)
...;
```

This allows use of PCT refresh on the materialized views `sales_hierarchical_mon_cube_mv`, `sales_hierarchical_qtr_cube_mv`, and `sales_hierarchical_yr_cube_mv` on partition maintenance operations to sales table. PCT refresh can also be used when there have been significant changes to the base table and log based fast refresh is estimated to be slower than PCT refresh. You can just specify the method as force (method => '?') in to refresh sub-programs in the `DBMS_MVIEW` package and Oracle Database will pick the best method of refresh. See "Partition Change Tracking (PCT) Refresh" on page 15-15 for more information regarding PCT refresh.

Because `sales_hierarchical_qtr_cube_mv` does not contain any column from `times` table, PCT refresh is not enabled on it. But, you can still call refresh sub-programs in the `DBMS_MVIEW` package with method as force (method => '?') and Oracle Database will pick the best method of refresh.

If you are interested in a partial cube (that is, a subset of groupings from the complete cube), then Oracle Corporation recommends storing the cube as a "federated cube". A federated cube stores each grouping of interest in a separate materialized view.

```
calendar_year yr, calendar_quarter_desc qtr, calendar_month_desc mon,
country_id, cust_state_province, cust_city, prod_category,
prod_subcategory, prod_name,

CREATE MATERIALIZED VIEW sales_mon_city_prod_mv
PARTITION BY RANGE (mon)
...
BUILD DEFERRED
REFRESH FAST ON DEMAND
  USING TRUSTED CONSTRAINTS
ENABLE QUERY REWRITE AS
SELECT calendar_month_desc mon, cust_city, prod_name, SUM(amount_sold) s_sales,
       COUNT(amount_sold) c_sales, COUNT(*) c_star
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id
AND s.time_id = t.time_id
GROUP BY calendar_month_desc, cust_city, prod_name;
```

```
CREATE MATERIALIZED VIEW sales_qtr_city_prod_mv
PARTITION BY RANGE (qtr)
...
BUILD DEFERRED
REFRESH FAST ON DEMAND
  USING TRUSTED CONSTRAINTS
ENABLE QUERY REWRITE AS
SELECT calendar_quarter_desc qtr, cust_city, prod_name,SUM(amount_sold) s_sales,
COUNT(amount_sold) c_sales, COUNT(*) c_star
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id =p.prod_id AND s.time_id = t.time_id
GROUP BY calendar_quarter_desc, cust_city, prod_name;

CREATE MATERIALIZED VIEW sales_yr_city_prod_mv
PARTITION BY RANGE (yr)
...
BUILD DEFERRED
REFRESH FAST ON DEMAND
USING TRUSTED CONSTRAINTS
ENABLE QUERY REWRITE AS
SELECT calendar_year yr, cust_city, prod_name, SUM(amount_sold) s_sales,
       COUNT(amount_sold) c_sales, COUNT(*) c_star
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id =p.prod_id AND s.time_id = t.time_id
GROUP BY calendar_year, cust_city, prod_name;

CREATE MATERIALIZED VIEW sales_mon_city_scat_mv
PARTITION BY RANGE (mon)
...
BUILD DEFERRED
REFRESH FAST ON DEMAND
  USING TRUSTED CONSTRAINTS
ENABLE QUERY REWRITE AS
SELECT calendar_month_desc mon, cust_city, prod_subcategory,
       SUM(amount_sold) s_sales, COUNT(amount_sold) c_sales, COUNT(*) c_star
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id =p.prod_id AND s.time_id =t.time_id
GROUP BY calendar_month_desc, cust_city, prod_subcategory;

CREATE MATERIALIZED VIEW sales_qtr_city_cat_mv
PARTITION BY RANGE (qtr)
...
BUILD DEFERRED
REFRESH FAST ON DEMAND
```

```
  USING TRUSTED CONSTRAINTS
ENABLE QUERY REWRITE AS
SELECT calendar_quarter_desc qtr, cust_city, prod_category cat,
       SUM(amount_sold) s_sales, COUNT(amount_sold) c_sales, COUNT(*) c_star
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id =p.prod_id AND s.time_id =t.time_id
GROUP BY calendar_quarter_desc, cust_city, prod_category;

CREATE MATERIALIZED VIEW sales_yr_city_all_mv
PARTITION BY RANGE (yr)
...
BUILD DEFERRED
REFRESH FAST ON DEMAND
  USING TRUSTED CONSTRAINTS
ENABLE QUERY REWRITE AS
SELECT calendar_year yr, cust_city, SUM(amount_sold) s_sales,
       COUNT(amount_sold) c_sales, COUNT(*) c_star
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id AND s.time_id = t.time_id
GROUP BY calendar_year, cust_city;
```

These materialized views can be created as BUILD DEFERRED and then, you can execute DBMS_MVIEW.REFRESH_DEPENDENT(number_of_failures, 'SALES', 'C' ...) so that the complete refresh of each of the materialized views defined on the detail table SALES is scheduled in the most efficient order. Please refer to section on "Scheduling Refresh" on page 15-22.

Because each of these materialized views is partitioned on the time level (month, quarter, or year) present in the SELECT list, PCT is enabled on SALES table for each one of them, thus providing an opportunity to apply PCT refresh method in addition to FAST and COMPLETE refresh methods.

21

SQL for Analysis and Reporting

The following topics provide information about how to improve analytical SQL queries in a data warehouse:

Overview of SQL for Analysis and Reporting

Oracle has enhanced SQL's analytical processing capabilities by introducing a new family of analytic SQL functions. These analytic functions enable you to calculate:

- Rankings and percentiles
- Moving window calculations
- Lag/lead analysis
- First/last analysis
- Linear regression statistics

Ranking functions include cumulative distributions, percent rank, and N-tiles. Moving window calculations allow you to find moving and cumulative aggregations, such as sums and averages. Lag/lead analysis enables direct inter-row references so you can calculate period-to-period changes. First/last analysis enables you to find the first or last value in an ordered group.

Other enhancements to SQL include the CASE expression. CASE expressions provide if-then logic useful in many situations.

In Oracle Database 10*g*, the SQL reporting capability was further enhanced by the introduction of partitioned outer join. Partitioned outer join is an extension to ANSI outer join syntax that allows users to selectively densify certain dimensions while keeping others sparse. This allows reporting tools to selectively densify dimensions, for example, the ones that appear in their cross-tabular reports while keeping others sparse.

To enhance performance, analytic functions can be parallelized: multiple processes can simultaneously execute all of these statements. These capabilities make calculations easier and more efficient, thereby enhancing database performance, scalability, and simplicity.

Analytic functions are classified as described in Table 21–1.

Table 21–1 Analytic Functions and Their Uses

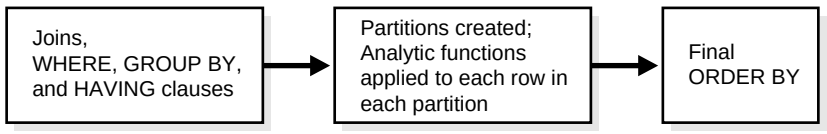
Type	Used For
Ranking	Calculating ranks, percentiles, and n-tiles of the values in a result set.
Windowing	Calculating cumulative and moving aggregates. Works with these functions: SUM, AVG, MIN, MAX, COUNT, VARIANCE, STDDEV, FIRST_VALUE, LAST_VALUE, and new statistical functions
Reporting	Calculating shares, for example, market share. Works with these functions: SUM, AVG, MIN, MAX, COUNT (with/without DISTINCT), VARIANCE, STDDEV, RATIO_TO_REPORT, and new statistical functions

Table 21–1 (Cont.) Analytic Functions and Their Uses

Type	Used For
LAG/LEAD	Finding a value in a row a specified number of rows from a current row.
FIRST/LAST	First or last value in an ordered group.
Linear Regression	Calculating linear regression and other statistics (slope, intercept, and so on).
Inverse Percentile	The value in a data set that corresponds to a specified percentile.
Hypothetical Rank and Distribution	The rank or percentile that a row would have if inserted into a specified data set.

To perform these operations, the analytic functions add several new elements to SQL processing. These elements build on existing SQL to allow flexible and powerful calculation expressions. With just a few exceptions, the analytic functions have these new elements. The processing flow is represented in Figure 21–1.

Figure 21–1 Processing Order



The essential concepts used in analytic functions are:

- Processing order

  Query processing using analytic functions takes place in three stages. First, all joins, `WHERE`, `GROUP BY` and `HAVING` clauses are performed. Second, the result set is made available to the analytic functions, and all their calculations take place. Third, if the query has an `ORDER BY` clause at its end, the `ORDER BY` is processed to allow for precise output ordering. The processing order is shown in Figure 21–1.

- Result set partitions

  The analytic functions allow users to divide query result sets into groups of rows called partitions. Note that the term **partitions** used with analytic functions is unrelated to the table partitions feature. Throughout this chapter, the term partitions refers to only the meaning related to analytic functions. Partitions are created after the groups defined with `GROUP BY` clauses, so they are available to any aggregate results such as sums and averages. Partition divisions may be based upon any desired columns or expressions. A query

result set may be partitioned into just one partition holding all the rows, a few large partitions, or many small partitions holding just a few rows each.

- Window

  For each row in a partition, you can define a sliding window of data. This window determines the range of rows used to perform the calculations for the current row. Window sizes can be based on either a physical number of rows or a logical interval such as time. The window has a starting row and an ending row. Depending on its definition, the window may move at one or both ends. For instance, a window defined for a cumulative sum function would have its starting row fixed at the first row of its partition, and its ending row would slide from the starting point all the way to the last row of the partition. In contrast, a window defined for a moving average would have both its starting and end points slide so that they maintain a constant physical or logical range.

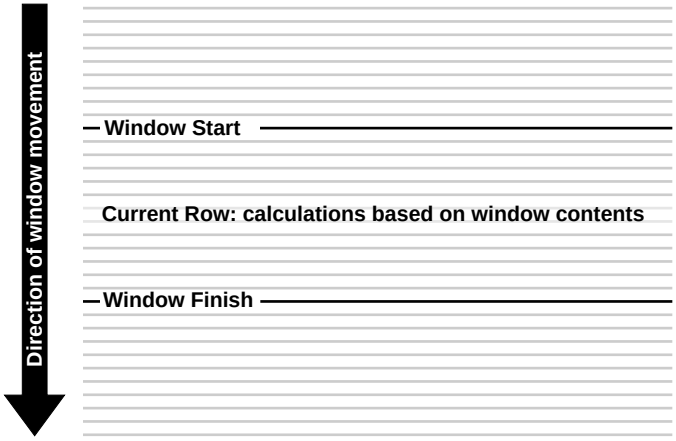
  A window can be set as large as all the rows in a partition or just a sliding window of one row within a partition. When a window is near a border, the function returns results for only the available rows, rather than warning you that the results are not what you want.

  When using window functions, the current row is included during calculations, so you should only specify (*n*-1) when you are dealing with *n* items.

- Current row

  Each calculation performed with an analytic function is based on a current row within a partition. The current row serves as the reference point determining the start and end of the window. For instance, a centered moving average calculation could be defined with a window that holds the current row, the six preceding rows, and the following six rows. This would create a sliding window of 13 rows, as shown in Figure 21–2.

Figure 21–2 Sliding Window Example



Ranking Functions

A ranking function computes the rank of a record compared to other records in the data set based on the values of a set of measures. The types of ranking function are:

- RANK and DENSE_RANK Functions
- CUME_DIST Function
- PERCENT_RANK Function
- NTILE Function
- ROW_NUMBER Function

RANK and DENSE_RANK Functions

The RANK and DENSE_RANK functions allow you to rank items in a group, for example, finding the top three products sold in California last year. There are two functions that perform ranking, as shown by the following syntax:

```
RANK ( ) OVER ( [query_partition_clause] order_by_clause )
DENSE_RANK ( ) OVER ( [query_partition_clause] order_by_clause )
```

The difference between RANK and DENSE_RANK is that DENSE_RANK leaves no gaps in ranking sequence when there are ties. That is, if you were ranking a competition using DENSE_RANK and had three people tie for second place, you would say that

all three were in second place and that the next person came in third. The RANK function would also give three people in second place, but the next person would be in fifth place.

The following are some relevant points about RANK:

- Ascending is the default sort order, which you may want to change to descending.
- The expressions in the optional PARTITION BY clause divide the query result set into groups within which the RANK function operates. That is, RANK gets reset whenever the group changes. In effect, the value expressions of the PARTITION BY clause define the reset boundaries.
- If the PARTITION BY clause is missing, then ranks are computed over the entire query result set.
- The ORDER BY clause specifies the measures (<value expression>) on which ranking is done and defines the order in which rows are sorted in each group (or partition). Once the data is sorted within each partition, ranks are given to each row starting from 1.
- The NULLS FIRST | NULLS LAST clause indicates the position of NULLs in the ordered sequence, either first or last in the sequence. The order of the sequence would make NULLs compare either high or low with respect to non-NULL values. If the sequence were in ascending order, then NULLS FIRST implies that NULLs are smaller than all other non-NULL values and NULLS LAST implies they are larger than non-NULL values. It is the opposite for descending order. See the example in "Treatment of NULLs" on page 21-10.
- If the NULLS FIRST | NULLS LAST clause is omitted, then the ordering of the null values depends on the ASC or DESC arguments. Null values are considered larger than any other values. If the ordering sequence is ASC, then nulls will appear last; nulls will appear first otherwise. Nulls are considered equal to other nulls and, therefore, the order in which nulls are presented is non-deterministic.

Ranking Order

The following example shows how the [ASC | DESC] option changes the ranking order.

Example 21–1 Ranking Order

```
SELECT channel_desc, TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$,
   RANK() OVER (ORDER BY SUM(amount_sold)) AS default_rank,
```

```
   RANK() OVER (ORDER BY SUM(amount_sold) DESC NULLS LAST) AS custom_rank
FROM sales, products, customers, times, channels, countries
WHERE sales.prod_id=products.prod_id AND sales.cust_id=customers.cust_id AND
  sales.time_id=times.time_id AND sales.channel_id=channels.channel_id AND
  times.calendar_month_desc IN ('2000-09', '2000-10') AND country_iso_code='US'
GROUP BY channel_desc;

CHANNEL_DESC         SALES$         DEFAULT_RANK CUSTOM_RANK
-------------------- -------------- ------------ -----------
Direct Sales              2,443,392            3           1
Partners                  1,365,963            2           2
Internet                    467,478            1           3
```

While the data in this result is ordered on the measure SALES$, in general, it is not guaranteed by the RANK function that the data will be sorted on the measures. If you want the data to be sorted on SALES$ in your result, you must specify it explicitly with an ORDER BY clause, at the end of the SELECT statement.

Ranking on Multiple Expressions

Ranking functions need to resolve ties between values in the set. If the first expression cannot resolve ties, the second expression is used to resolve ties and so on. For example, here is a query ranking three of the sales channels over two months based on their dollar sales, breaking ties with the unit sales. (Note that the TRUNC function is used here only to create tie values for this query.)

Example 21–2 Ranking On Multiple Expressions

```
SELECT channel_desc, calendar_month_desc, TO_CHAR(TRUNC(SUM(amount_sold),-5),
  '9,999,999,999') SALES$, TO_CHAR(SUM(quantity_sold), '9,999,999,999')
  SALES_Count, RANK() OVER (ORDER BY TRUNC(SUM(amount_sold), -5) DESC,
SUM(quantity_sold) DESC) AS col_rank
FROM sales, products, customers, times, channels
WHERE sales.prod_id=products.prod_id AND sales.cust_id=customers.cust_id AND
   sales.time_id=times.time_id AND sales.channel_id=channels.channel_id AND
   times.calendar_month_desc IN ('2000-09', '2000-10') AND
   channels.channel_desc<>'Tele Sales'
GROUP BY channel_desc, calendar_month_desc;

CHANNEL_DESC         CALENDAR SALES$         SALES_COUNT     COL_RANK
-------------------- -------- -------------- -------------- ---------
Direct Sales         2000-10       1,200,000         12,584         1
Direct Sales         2000-09       1,200,000         11,995         2
Partners             2000-10         600,000          7,508         3
```

```
Partners             2000-09         600,000          6,165          4
Internet             2000-09         200,000          1,887          5
Internet             2000-10         200,000          1,450          6
```

The sales_count column breaks the ties for three pairs of values.

RANK and DENSE_RANK Difference

The difference between RANK and DENSE_RANK functions is illustrated as follows:

Example 21–3 RANK and DENSE_RANK

```
SELECT channel_desc, calendar_month_desc,
   TO_CHAR(TRUNC(SUM(amount_sold),-4), '9,999,999,999') SALES$,
      RANK() OVER (ORDER BY TRUNC(SUM(amount_sold),-4) DESC) AS RANK,
DENSE_RANK() OVER (ORDER BY TRUNC(SUM(amount_sold),-4) DESC) AS DENSE_RANK
FROM sales, products, customers, times, channels
WHERE sales.prod_id=products.prod_id AND sales.cust_id=customers.cust_id AND
  sales.time_id=times.time_id AND sales.channel_id=channels.channel_id AND
  times.calendar_month_desc IN ('2000-09', '2000-10') AND
      channels.channel_desc<>'Tele Sales'
GROUP BY channel_desc, calendar_month_desc;

CHANNEL_DESC         CALENDAR SALES$              RANK DENSE_RANK
-------------------- -------- -------------- --------- ----------
Direct Sales         2000-09       1,200,000          1          1
Direct Sales         2000-10       1,200,000          1          1
Partners             2000-09         600,000          3          2
Partners             2000-10         600,000          3          2
Internet             2000-09         200,000          5          3
Internet             2000-10         200,000          5          3
```

Note that, in the case of DENSE_RANK, the largest rank value gives the number of distinct values in the data set.

Per Group Ranking

The RANK function can be made to operate within groups, that is, the rank gets reset whenever the group changes. This is accomplished with the PARTITION BY clause. The group expressions in the PARTITION BY subclause divide the data set into groups within which RANK operates. For example, to rank products within each channel by their dollar sales, you could issue the following statement.

Example 21–4 Per Group Ranking Example 1

```
SELECT channel_desc, calendar_month_desc, TO_CHAR(SUM(amount_sold),
 '9,999,999,999') SALES$, RANK() OVER (PARTITION BY channel_desc
  ORDER BY SUM(amount_sold) DESC) AS RANK_BY_CHANNEL
FROM sales, products, customers, times, channels
WHERE sales.prod_id=products.prod_id AND sales.cust_id=customers.cust_id AND
      sales.time_id=times.time_id AND sales.channel_id=channels.channel_id AND
      times.calendar_month_desc IN ('2000-08', '2000-09', '2000-10', '2000-11')
      AND channels.channel_desc IN ('Direct Sales', 'Internet')
GROUP BY channel_desc, calendar_month_desc;
```

A single query block can contain more than one ranking function, each partitioning the data into different groups (that is, reset on different boundaries). The groups can be mutually exclusive. The following query ranks products based on their dollar sales within each month (`rank_of_product_per_region`) and within each channel (`rank_of_product_total`).

Example 21–5 Per Group Ranking Example 2

```
SELECT channel_desc, calendar_month_desc, TO_CHAR(SUM(amount_sold),
  '9,999,999,999') SALES$, RANK() OVER (PARTITION BY calendar_month_desc
  ORDER BY SUM(amount_sold) DESC) AS RANK_WITHIN_MONTH, RANK() OVER (PARTITION
 BY channel_desc ORDER BY SUM(amount_sold) DESC) AS RANK_WITHIN_CHANNEL
FROM sales, products, customers, times, channels
WHERE sales.prod_id=products.prod_id AND sales.cust_id=customers.cust_id AND
      sales.time_id=times.time_id AND sales.channel_id=channels.channel_id AND
      times.calendar_month_desc IN ('2000-08', '2000-09', '2000-10', '2000-11')
    AND channels.channel_desc IN ('Direct Sales', 'Internet')
GROUP BY channel_desc, calendar_month_desc;
```

```
CHANNEL_DESC           CALENDAR   SALES$         RANK_WITHIN_MONTH RANK_WITHIN_CHANNEL
--------------------   --------   ------------- ----------------- ------------------
Direct Sales         2000-08       1,236,104                 1                  1
Internet             2000-08         215,107                 2                  4
Direct Sales         2000-09       1,217,808                 1                  3
Internet             2000-09         228,241                 2                  3
Direct Sales         2000-10       1,225,584                 1                  2
Internet             2000-10         239,236                 2                  2
Direct Sales         2000-11       1,115,239                 1                  4
Internet             2000-11         284,742                 2                  1
```

Per Cube and Rollup Group Ranking

Analytic functions, RANK for example, can be reset based on the groupings provided by a CUBE, ROLLUP, or GROUPING SETS operator. It is useful to assign ranks to the groups created by CUBE, ROLLUP, and GROUPING SETS queries. See Chapter 20, "SQL for Aggregation in Data Warehouses" for further information about the GROUPING function.

A sample CUBE and ROLLUP query is the following:

```
SELECT channel_desc, country_iso_code,
  TO_CHAR(SUM(amount_sold), '9,999,999,999')
 SALES$, RANK() OVER (PARTITION BY GROUPING_ID(channel_desc, country_iso_code)
    ORDER BY SUM(amount_sold) DESC) AS RANK_PER_GROUP
FROM sales, customers, times, channels, countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND
   sales.channel_id = channels.channel_id AND channels.channel_desc
     IN ('Direct Sales', 'Internet') AND times.calendar_month_desc='2000-09'
   AND country_iso_code IN ('GB', 'US', 'JP')
GROUP BY CUBE(channel_desc, country_iso_code);

CHANNEL_DESC         CO SALES$         RANK_PER_GROUP
-------------------- -- -------------- --------------
Direct Sales         GB      1,217,808              1
Direct Sales         JP      1,217,808              1
Direct Sales         US      1,217,808              1
Internet             GB        228,241              4
Internet             JP        228,241              4
Internet             US        228,241              4
Direct Sales                 3,653,423              1
Internet                       684,724              2
                     GB      1,446,049              1
                     JP      1,446,049              1
                     US      1,446,049              1
                             4,338,147              1
```

Treatment of NULLs

NULLS are treated like normal values. Also, for rank computation, a NULL value is assumed to be equal to another NULL value. Depending on the ASC | DESC options provided for measures and the NULLS FIRST | NULLS LAST clause, NULLS will either sort low or high and hence, are given ranks appropriately. The following example shows how NULLS are ranked in different cases:

```
SELECT  times.time_id time,  sold,
  RANK() OVER (ORDER BY (sold) DESC NULLS LAST) AS NLAST_DESC,
```

```
  RANK() OVER (ORDER BY (sold) DESC NULLS FIRST) AS NFIRST_DESC,
  RANK() OVER (ORDER BY (sold) ASC NULLS FIRST) AS NFIRST,
  RANK() OVER (ORDER BY (sold) ASC NULLS LAST) AS NLAST
FROM
  (
   SELECT   time_id , sum(sales.amount_sold)  sold
   FROM sales, products, customers, countries
   WHERE sales.prod_id=products.prod_id AND
     sales.cust_id=customers.cust_id AND prod_name IN ('Envoy Ambassador',
       'Mouse Pad') AND country_iso_code ='GB'
   GROUP BY  time_id)
 v, times
WHERE v.time_id (+) =times.time_id AND calendar_year=1999
      AND calendar_month_number=1
ORDER BY sold  DESC NULLS LAST;

TIME            SOLD NLAST_DESC NFIRST_DESC     NFIRST      NLAST
--------- ---------- ---------- ----------- ---------- ----------
14-JAN-99   25241.48          1          13         31         19
21-JAN-99   24365.05          2          14         30         18
10-JAN-99   22901.24          3          15         29         17
20-JAN-99   16578.19          4          16         28         16
16-JAN-99   15881.12          5          17         27         15
30-JAN-99   15637.49          6          18         26         14
17-JAN-99   13262.87          7          19         25         13
25-JAN-99   13227.08          8          20         24         12
03-JAN-99    9885.74          9          21         23         11
28-JAN-99    4471.08         10          22         22         10
27-JAN-99    3453.66         11          23         21          9
23-JAN-99     925.45         12          24         20          8
07-JAN-99     756.87         13          25         19          7
08-JAN-99      571.8         14          26         18          6
13-JAN-99     569.21         15          27         17          5
02-JAN-99     316.87         16          28         16          4
12-JAN-99     195.54         17          29         15          3
26-JAN-99      92.96         18          30         14          2
19-JAN-99      86.04         19          31         13          1
05-JAN-99                    20           1          1         20
01-JAN-99                    20           1          1         20
31-JAN-99                    20           1          1         20
11-JAN-99                    20           1          1         20
06-JAN-99                    20           1          1         20
18-JAN-99                    20           1          1         20
09-JAN-99                    20           1          1         20
29-JAN-99                    20           1          1         20
```

```
22-JAN-99                  20          1          1         20
04-JAN-99                  20          1          1         20
24-JAN-99                  20          1          1         20
15-JAN-99                  20          1          1         20
```

Bottom N Ranking

Bottom N is similar to top N except for the ordering sequence within the rank expression. Using the previous example, you can order SUM(s_amount) ascending instead of descending.

CUME_DIST Function

The CUME_DIST function (defined as the inverse of percentile in some statistical books) computes the position of a specified value relative to a set of values. The order can be ascending or descending. Ascending is the default. The range of values for CUME_DIST is from greater than 0 to 1. To compute the CUME_DIST of a value x in a set S of size N, you use the formula:

```
CUME_DIST(x) =  number of values in S coming before
   and including x in the specified order/ N
```

Its syntax is:

```
CUME_DIST ( ) OVER ( [query_partition_clause] order_by_clause )
```

The semantics of various options in the CUME_DIST function are similar to those in the RANK function. The default order is ascending, implying that the lowest value gets the lowest CUME_DIST (as all other values come later than this value in the order). NULLS are treated the same as they are in the RANK function. They are counted toward both the numerator and the denominator as they are treated like non-NULL values. The following example finds cumulative distribution of sales by channel within each month:

```
SELECT calendar_month_desc AS MONTH, channel_desc,
     TO_CHAR(SUM(amount_sold) , '9,999,999,999') SALES$,
     CUME_DIST() OVER (PARTITION BY calendar_month_desc ORDER BY
         SUM(amount_sold) ) AS CUME_DIST_BY_CHANNEL
FROM sales, products, customers, times, channels
WHERE sales.prod_id=products.prod_id AND sales.cust_id=customers.cust_id AND
      sales.time_id=times.time_id AND sales.channel_id=channels.channel_id AND
      times.calendar_month_desc IN ('2000-09', '2000-07','2000-08')
GROUP BY calendar_month_desc, channel_desc;
```

```
MONTH    CHANNEL_DESC         SALES$         CUME_DIST_BY_CHANNEL
-------- -------------------- -------------- --------------------
2000-07  Internet                  140,423           .333333333
2000-07  Partners                  611,064           .666666667
2000-07  Direct Sales            1,145,275                    1
2000-08  Internet                  215,107           .333333333
2000-08  Partners                  661,045           .666666667
2000-08  Direct Sales            1,236,104                    1
2000-09  Internet                  228,241           .333333333
2000-09  Partners                  666,172           .666666667
2000-09  Direct Sales            1,217,808                    1
```

PERCENT_RANK Function

PERCENT_RANK is similar to CUME_DIST, but it uses rank values rather than row counts in its numerator. Therefore, it returns the percent rank of a value relative to a group of values. The function is available in many popular spreadsheets. PERCENT_RANK of a row is calculated as:

```
(rank of row in its partition - 1) / (number of rows in the partition - 1)
```

PERCENT_RANK returns values in the range zero to one. The row(s) with a rank of 1 will have a PERCENT_RANK of zero. Its syntax is:

```
PERCENT_RANK () OVER ([query_partition_clause] order_by_clause)
```

NTILE Function

NTILE allows easy calculation of tertiles, quartiles, deciles and other common summary statistics. This function divides an ordered partition into a specified number of groups called **buckets** and assigns a bucket number to each row in the partition. NTILE is a very useful calculation because it lets users divide a data set into fourths, thirds, and other groupings.

The buckets are calculated so that each bucket has exactly the same number of rows assigned to it or at most 1 row more than the others. For instance, if you have 100 rows in a partition and ask for an NTILE function with four buckets, 25 rows will be assigned a value of 1, 25 rows will have value 2, and so on. These buckets are referred to as equiheight buckets.

If the number of rows in the partition does not divide evenly (without a remainder) into the number of buckets, then the number of rows assigned for each bucket will differ by one at most. The extra rows will be distributed one for each bucket starting from the lowest bucket number. For instance, if there are 103 rows in a partition

which has an NTILE(5) function, the first 21 rows will be in the first bucket, the next 21 in the second bucket, the next 21 in the third bucket, the next 20 in the fourth bucket and the final 20 in the fifth bucket.

The NTILE function has the following syntax:

```
NTILE (expr) OVER ([query_partition_clause] order_by_clause)
```

In this, the N in NTILE(N) can be a constant (for example, 5) or an expression.

This function, like RANK and CUME_DIST, has a PARTITION BY clause for per group computation, an ORDER BY clause for specifying the measures and their sort order, and NULLS FIRST | NULLS LAST clause for the specific treatment of NULLs. For example, the following is an example assigning each month's sales total into one of 4 buckets:

```
SELECT calendar_month_desc AS MONTH , TO_CHAR(SUM(amount_sold),
 '9,999,999,999')
 SALES$, NTILE(4) OVER (ORDER BY SUM(amount_sold)) AS TILE4
FROM sales, products, customers, times, channels
WHERE sales.prod_id=products.prod_id AND sales.cust_id=customers.cust_id AND
      sales.time_id=times.time_id AND sales.channel_id=channels.channel_id AND
      times.calendar_year=2000 AND prod_category= 'Electronics'
GROUP BY calendar_month_desc;

MONTH    SALES$              TILE4
-------- -------------- ----------
2000-02         242,416          1
2000-01         257,286          1
2000-03         280,011          1
2000-06         315,951          2
2000-05         316,824          2
2000-04         318,106          2
2000-07         433,824          3
2000-08         477,833          3
2000-12         553,534          3
2000-10         652,225          4
2000-11         661,147          4
2000-09         691,449          4
```

NTILE ORDER BY statements must be fully specified to yield reproducible results. Equal values can get distributed across adjacent buckets. To ensure deterministic results, you must order on a unique key.

ROW_NUMBER Function

The ROW_NUMBER function assigns a unique number (sequentially, starting from 1, as defined by ORDER BY) to each row within the partition. It has the following syntax:

```
ROW_NUMBER ( ) OVER ( [query_partition_clause] order_by_clause )
```

Example 21–6 ROW_NUMBER

```
SELECT channel_desc, calendar_month_desc,
   TO_CHAR(TRUNC(SUM(amount_sold), -5), '9,999,999,999') SALES$,
   ROW_NUMBER() OVER (ORDER BY TRUNC(SUM(amount_sold), -6) DESC) AS ROW_NUMBER
FROM sales, products, customers, times, channels
WHERE sales.prod_id=products.prod_id AND sales.cust_id=customers.cust_id AND
      sales.time_id=times.time_id AND sales.channel_id=channels.channel_id AND
      times.calendar_month_desc IN ('2001-09', '2001-10')
GROUP BY channel_desc, calendar_month_desc;

CHANNEL_DESC         CALENDAR SALES$         ROW_NUMBER
-------------------- -------- -------------- ----------
Direct Sales         2001-09       1,100,000          1
Direct Sales         2001-10       1,000,000          2
Internet             2001-09         500,000          3
Internet             2001-10         700,000          4
Partners             2001-09         600,000          5
Partners             2001-10         600,000          6
```

Note that there are three pairs of tie values in these results. Like NTILE, ROW_NUMBER is a non-deterministic function, so each tied value could have its row number switched. To ensure deterministic results, you must order on a unique key. Inmost cases, that will require adding a new tie breaker column to the query and using it in the ORDER BY specification.

Windowing Aggregate Functions

Windowing functions can be used to compute cumulative, moving, and centered aggregates. They return a value for each row in the table, which depends on other rows in the corresponding window. These functions include moving sum, moving average, moving min / max, cumulative sum, as well as statistical functions. They can be used only in the SELECT and ORDER BY clauses of the query. Two other functions are available: FIRST_VALUE, which returns the first value in the window; and LAST_VALUE, which returns the last value in the window. These functions

provide access to more than one row of a table without a self-join. The syntax of the windowing functions is:

```
{SUM|AVG|MAX|MIN|COUNT|STDDEV|VARIANCE|FIRST_VALUE|LAST_VALUE}
  ({value expression1 | *}) OVER
    ([PARTITION BY value expression2[,...])
      ORDER BY value expression3 [collate clause>]
                [ASC| DESC] [NULLS FIRST | NULLS LAST] [,...]
{ROWS | RANGE} {BETWEEN
  {UNBOUNDED PRECEDING | CURRENT ROW | value_expr {PRECEDING | FOLLOWING}} AND
  { UNBOUNDED FOLLOWING | CURRENT ROW | value_expr { PRECEDING | FOLLOWING } }
  | { UNBOUNDED PRECEDING | CURRENT ROW | value_expr PRECEDING}}
```

> **See Also:** *Oracle Database SQL Reference* for further information regarding syntax and restrictions

Treatment of NULLs as Input to Window Functions

Window functions' NULL semantics match the NULL semantics for SQL aggregate functions. Other semantics can be obtained by user-defined functions, or by using the DECODE or a CASE expression within the window function.

Windowing Functions with Logical Offset

A logical offset can be specified with constants such as RANGE 10 PRECEDING, or an expression that evaluates to a constant, or by an interval specification like RANGE INTERVAL N DAY/MONTH/YEAR PRECEDING or an expression that evaluates to an interval. With logical offset, there can only be one expression in the ORDER BY expression list in the function, with type compatible to NUMERIC if offset is numeric, or DATE if an interval is specified.

Example 21–7 Cumulative Aggregate Function

The following is an example of cumulative amount_sold by customer ID by quarter in 1999:

```
SELECT c.cust_id, t.calendar_quarter_desc, TO_CHAR (SUM(amount_sold),
  '9,999,999,999.99') AS Q_SALES, TO_CHAR(SUM(SUM(amount_sold))

OVER (PARTITION BY c.cust_id ORDER BY c.cust_id, t.calendar_quarter_desc
ROWS UNBOUNDED
PRECEDING), '9,999,999,999.99') AS CUM_SALES
  FROM sales s, times t, customers c
  WHERE s.time_id=t.time_id AND s.cust_id=c.cust_id AND t.calendar_year=2000
```

```
  AND c.cust_id IN (2595, 9646, 11111)
  GROUP BY c.cust_id, t.calendar_quarter_desc
  ORDER BY c.cust_id, t.calendar_quarter_desc;

   CUST_ID CALENDA Q_SALES           CUM_SALES
---------- ------- ----------------- -----------------
      2595 2000-01           659.92            659.92
      2595 2000-02           224.79            884.71
      2595 2000-03           313.90          1,198.61
      2595 2000-04         6,015.08          7,213.69
      9646 2000-01         1,337.09          1,337.09
      9646 2000-02           185.67          1,522.76
      9646 2000-03           203.86          1,726.62
      9646 2000-04           458.29          2,184.91
     11111 2000-01            43.18             43.18
     11111 2000-02            33.33             76.51
     11111 2000-03           579.73            656.24
     11111 2000-04           307.58            963.82
```

In this example, the analytic function SUM defines, for each row, a window that starts at the beginning of the partition (UNBOUNDED PRECEDING) and ends, by default, at the current row.

Nested SUMs are needed in this example since we are performing a SUM over a value that is itself a SUM. Nested aggregations are used very often in analytic aggregate functions.

Example 21–8 Moving Aggregate Function

This example of a time-based window shows, for one customer, the moving average of sales for the current month and preceding two months:

```
SELECT c.cust_id, t.calendar_month_desc, TO_CHAR (SUM(amount_sold),
      '9,999,999,999') AS SALES, TO_CHAR(AVG(SUM(amount_sold))
OVER (ORDER BY c.cust_id, t.calendar_month_desc  ROWS 2 PRECEDING),
'9,999,999,999') AS MOVING_3_MONTH_AVG
FROM sales s, times t, customers c
WHERE s.time_id=t.time_id AND s.cust_id=c.cust_id AND
      t.calendar_year=1999 AND c.cust_id IN (6510)
GROUP BY c.cust_id, t.calendar_month_desc
ORDER BY c.cust_id, t.calendar_month_desc;

   CUST_ID CALENDAR SALES          MOVING_3_MONTH
---------- -------- -------------- --------------
      6510 1999-04            125            125
```

```
      6510 1999-05           3,395          1,760
      6510 1999-06           4,080          2,533
      6510 1999-07           6,435          4,637
      6510 1999-08           5,105          5,207
      6510 1999-09           4,676          5,405
      6510 1999-10           5,109          4,963
      6510 1999-11             802          3,529
```

Note that the first two rows for the three month moving average calculation in the output data are based on a smaller interval size than specified because the window calculation cannot reach past the data retrieved by the query. You need to consider the different window sizes found at the borders of result sets. In other words, you may need to modify the query to include exactly what you want.

Centered Aggregate Function

Calculating windowing aggregate functions centered around the current row is straightforward. This example computes for all customers a centered moving average of sales for one week in late December 1999. It finds an average of the sales total for the one day preceding the current row and one day following the current row including the current row as well.

Example 21–9 Centered Aggregate

```
SELECT t.time_id, TO_CHAR (SUM(amount_sold), '9,999,999,999')
AS SALES, TO_CHAR(AVG(SUM(amount_sold)) OVER
  (ORDER BY t.time_id
   RANGE BETWEEN INTERVAL '1' DAY PRECEDING AND
   INTERVAL '1' DAY FOLLOWING), '9,999,999,999') AS CENTERED_3_DAY_AVG
   FROM sales s, times t
   WHERE s.time_id=t.time_id AND t.calendar_week_number IN (51)
   AND calendar_year=1999
   GROUP BY t.time_id
   ORDER BY t.time_id;

TIME_ID   SALES          CENTERED_3_DAY
--------- -------------- --------------
20-DEC-99        134,337        106,676
21-DEC-99         79,015        102,539
22-DEC-99         94,264         85,342
23-DEC-99         82,746         93,322
24-DEC-99        102,957         82,937
25-DEC-99         63,107         87,062
26-DEC-99         95,123         79,115
```

The starting and ending rows for each product's centered moving average calculation in the output data are based on just two days, since the window calculation cannot reach past the data retrieved by the query. Users need to consider the different window sizes found at the borders of result sets: the query may need to be adjusted.

Windowing Aggregate Functions in the Presence of Duplicates

The following example illustrates how window aggregate functions compute values when there are duplicates, that is, when multiple rows are returned for a single ordering value. The query retrieves the quantity sold to several customers during a specified time range. (Although we use an inline view to define our base data set, it has no special significance and can be ignored.) The query defines a moving window that runs from the date of the current row to 10 days earlier.

Note that the RANGE keyword is used to define the windowing clause of this example. This means that the window can potentially hold many rows for each value in the range. In this case, there are three pairs of rows with duplicate date values.

Example 21–10 Windowing Aggregate Functions with Logical Offsets

```
SELECT time_id, daily_sum, SUM(daily_sum) OVER (ORDER BY time_id
RANGE BETWEEN INTERVAL '10' DAY PRECEDING AND CURRENT ROW)
AS current_group_sum
FROM (SELECT time_id, channel_id, SUM(s.quantity_sold)
AS daily_sum
FROM customers c, sales s, countries
WHERE c.cust_id=s.cust_id
AND s.cust_id IN (638, 634, 753, 440 ) AND s.time_id BETWEEN '01-MAY-00' AND
 '13-MAY-00' GROUP BY time_id, channel_id);

TIME_ID    DAILY_SUM CURRENT_GROUP_SUM
--------- ---------- -----------------
06-MAY-00        161               161   /*  161   */
10-MAY-00         23               207   /*  161 +(23+23)*/
10-MAY-00         23               207   /*  161 +(23+23)  */
11-MAY-00         46               345   /*  161 +(23+23)+(46+92)  */
11-MAY-00         92               345   /*  161 +(23+23)+(46+92)   */
12-MAY-00         23               368   /*  161 +(23+23)+(46+92)+23  */
13-MAY-00         46               529   /*  161 +(23+23)+(46+92)+23+(46+115) */
13-MAY-00        115               529   /*  161 +(23+23)+(46+92)+23+(46+115) */
```

In the output of this example, all dates except May 6 and May 12 return two rows with duplicate dates. Examine the commented numbers to the right of the output to see how the values are calculated. Note that each group in parentheses represents the values returned for a single day.

Note that this example applies only when you use the RANGE keyword rather than the ROWS keyword. It is also important to remember that with RANGE, you can only use 1 ORDER BY expression in the analytic function's ORDER BY clause. With the ROWS keyword, you can use multiple order by expressions in the analytic function's ORDER BY clause.

Varying Window Size for Each Row

There are situations where it is useful to vary the size of a window for each row, based on a specified condition. For instance, you may want to make the window larger for certain dates and smaller for others. Assume that you want to calculate the moving average of stock price over three working days. If you have an equal number of rows for each day for all working days and no non-working days are stored, then you can use a physical window function. However, if the conditions noted are not met, you can still calculate a moving average by using an expression in the window size parameters.

Expressions in a window size specification can be made in several different sources. the expression could be a reference to a column in a table, such as a time table. It could also be a function that returns the appropriate boundary for the window based on values in the current row. The following statement for a hypothetical stock price database uses a user-defined function in its RANGE clause to set window size:

```
SELECT t_timekey, AVG(stock_price)
       OVER (ORDER BY t_timekey RANGE fn(t_timekey) PRECEDING) av_price
FROM stock, time WHERE st_timekey = t_timekey
ORDER BY t_timekey;
```

In this statement, `t_timekey` is a date field. Here, *fn* could be a PL/SQL function with the following specification:

`fn(t_timekey)` returns

- 4 if `t_timekey` is Monday, Tuesday
- 2 otherwise
- If any of the previous days are holidays, it adjusts the count appropriately.

Note that, when window is specified using a number in a window function with ORDER BY on a date column, then it is converted to mean the number of days. You

could have also used the interval literal conversion function, as NUMTODSINTERVAL(fn(t_timekey), 'DAY') instead of just fn(t_timekey) to mean the same thing. You can also write a PL/SQL function that returns an INTERVAL datatype value.

Windowing Aggregate Functions with Physical Offsets

For windows expressed in rows, the ordering expressions should be unique to produce deterministic results. For example, the following query is not deterministic because time_id is not unique in this result set.

Example 21–11 Windowing Aggregate Functions With Physical Offsets

```
SELECT t.time_id, TO_CHAR(amount_sold, '9,999,999,999') AS INDIV_SALE,
       TO_CHAR(SUM(amount_sold) OVER (PARTITION BY t.time_id ORDER BY t.time_id
ROWS UNBOUNDED PRECEDING), '9,999,999,999') AS CUM_SALES
FROM sales s, times t, customers c
WHERE s.time_id=t.time_id AND s.cust_id=c.cust_id AND t.time_id IN
 (TO_DATE('11-DEC-1999'), TO_DATE('12-DEC-1999')) AND c.cust_id
 BETWEEN 6500 AND 6600
ORDER BY t.time_id;

TIME_ID   INDIV_SALE    CUM_SALES
--------- ----------    ---------
12-DEC-99         23           23
12-DEC-99          9           32
12-DEC-99         14           46
12-DEC-99         24           70
12-DEC-99         19           89
```

One way to handle this problem would be to add the prod_id column to the result set and order on both time_id and prod_id.

FIRST_VALUE and LAST_VALUE Functions

The FIRST_VALUE and LAST_VALUE functions allow you to select the first and last rows from a window. These rows are especially valuable because they are often used as the baselines in calculations. For instance, with a partition holding sales data ordered by day, you might ask "How much was each day's sales compared to the first sales day (FIRST_VALUE) of the period?" Or you might wish to know, for a set of rows in increasing sales order, "What was the percentage size of each sale in the region compared to the largest sale (LAST_VALUE) in the region?"

If the IGNORE NULLS option is used with FIRST_VALUE, it will return the first non-null value in the set, or NULL if all values are NULL. If IGNORE NULLS is used with LAST_VALUE, it will return the last non-null value in the set, or NULL if all values are NULL. The IGNORE NULLS option is particularly useful in populating an inventory table properly.

Reporting Aggregate Functions

After a query has been processed, aggregate values like the number of resulting rows or an average value in a column can be easily computed within a partition and made available to other reporting functions. Reporting aggregate functions return the same aggregate value for every row in a partition. Their behavior with respect to NULLs is the same as the SQL aggregate functions. The syntax is:

```
{SUM | AVG | MAX | MIN | COUNT | STDDEV | VARIANCE}
  ([ALL | DISTINCT] {value expression1 | *})
   OVER ([PARTITION BY value expression2[,...]])
```

In addition, the following conditions apply:

- An asterisk (*) is only allowed in COUNT(*)
- DISTINCT is supported only if corresponding aggregate functions allow it
- *value expression1* and *value expression2* can be any valid expression involving column references or aggregates.
- The PARTITION BY clause defines the groups on which the windowing functions would be computed. If the PARTITION BY clause is absent, then the function is computed over the whole query result set.

Reporting functions can appear only in the SELECT clause or the ORDER BY clause. The major benefit of reporting functions is their ability to do multiple passes of data in a single query block and speed up query performance. Queries such as "Count the number of salesmen with sales more than 10% of city sales" do not require joins between separate query blocks.

For example, consider the question "For each product category, find the region in which it had maximum sales". The equivalent SQL query using the MAX reporting aggregate function would be:

```
SELECT prod_category, country_region, sales
FROM (SELECT SUBSTR(p.prod_category,1,8) AS prod_category, co.country_region,
SUM(amount_sold) AS sales,
MAX(SUM(amount_sold)) OVER (PARTITION BY prod_category) AS MAX_REG_SALES
FROM sales s, customers c, countries co, products p
```

```
WHERE s.cust_id=c.cust_id AND c.country_id=co.country_id AND
      s.prod_id =p.prod_id AND s.time_id = TO_DATE('11-OCT-2001')
GROUP BY prod_category, country_region)
WHERE sales = MAX_REG_SALES;
```

The inner query with the reporting aggregate function `MAX(SUM(amount_sold))` returns:

```
PROD_CAT COUNTRY_REGION            SALES MAX_REG_SALES
-------- -------------------- ---------- -------------
Electron Americas                581.92        581.92
Hardware Americas                925.93        925.93
Peripher Americas               3084.48       4290.38
Peripher Asia                   2616.51       4290.38
Peripher Europe                 4290.38       4290.38
Peripher Oceania                 940.43       4290.38
Software Americas                4445.7        4445.7
Software Asia                   1408.19        4445.7
Software Europe                 3288.83        4445.7
Software Oceania                 890.25        4445.7
```

The full query results are:

```
PROD_CAT COUNTRY_REGION            SALES
-------- -------------------- ----------
Electron Americas                581.92
Hardware Americas                925.93
Peripher Europe                 4290.38
Software Americas                4445.7
```

Example 21–12 Reporting Aggregate Example

Reporting aggregates combined with nested queries enable you to answer complex queries efficiently. For example, what if you want to know the best selling products in your most significant product subcategories? The following is a query which finds the 5 top-selling products for each product subcategory that contributes more than 20% of the sales within its product category:

```
SELECT SUBSTR(prod_category,1,8) AS CATEG, prod_subcategory, prod_id, SALES
FROM (SELECT p.prod_category, p.prod_subcategory, p.prod_id,
      SUM(amount_sold) AS SALES,
      SUM(SUM(amount_sold)) OVER (PARTITION BY p.prod_category) AS CAT_SALES,
      SUM(SUM(amount_sold)) OVER
         (PARTITION BY p.prod_subcategory) AS SUBCAT_SALES,
      RANK() OVER  (PARTITION BY p.prod_subcategory
         ORDER BY SUM(amount_sold) ) AS RANK_IN_LINE
```

```
    FROM sales s, customers c, countries co, products p
    WHERE s.cust_id=c.cust_id AND
       c.country_id=co.country_id AND  s.prod_id=p.prod_id AND
       s.time_id=to_DATE('11-OCT-2000')
    GROUP BY p.prod_category, p.prod_subcategory, p.prod_id
    ORDER BY prod_category, prod_subcategory)
WHERE SUBCAT_SALES>0.2*CAT_SALES AND RANK_IN_LINE<=5;
```

RATIO_TO_REPORT Function

The RATIO_TO_REPORT function computes the ratio of a value to the sum of a set of values. If the expression value expression evaluates to NULL, RATIO_TO_REPORT also evaluates to NULL, but it is treated as zero for computing the sum of values for the denominator. Its syntax is:

```
RATIO_TO_REPORT ( expr ) OVER ( [query_partition_clause] )
```

In this, the following applies:

- expr can be any valid expression involving column references or aggregates.
- The PARTITION BY clause defines the groups on which the RATIO_TO_REPORT function is to be computed. If the PARTITION BY clause is absent, then the function is computed over the whole query result set.

Example 21–13 RATIO_TO_REPORT

To calculate RATIO_TO_REPORT of sales for each channel, you might use the following syntax:

```
SELECT ch.channel_desc, TO_CHAR(SUM(amount_sold),'9,999,999') AS SALES,
     TO_CHAR(SUM(SUM(amount_sold)) OVER (), '9,999,999') AS TOTAL_SALES,
     TO_CHAR(RATIO_TO_REPORT(SUM(amount_sold)) OVER (), '9.999')
     AS RATIO_TO_REPORT
FROM sales s, channels ch
WHERE s.channel_id=ch.channel_id  AND s.time_id=to_DATE('11-OCT-2000')
GROUP BY ch.channel_desc;

CHANNEL_DESC         SALES      TOTAL_SALE RATIO_
-------------------- ---------- ---------- ------
Direct Sales             14,447     23,183   .623
Internet                    345     23,183   .015
Partners                  8,391     23,183   .362
```

LAG/LEAD Functions

The LAG and LEAD functions are useful for comparing values when the relative positions of rows can be known reliably. They work by specifying the count of rows which separate the target row from the current row. Because the functions provide access to more than one row of a table at the same time without a self-join, they can enhance processing speed. The LAG function provides access to a row at a given offset prior to the current position, and the LEAD function provides access to a row at a given offset after the current position.

LAG/LEAD Syntax

These functions have the following syntax:

```
{LAG | LEAD} ( value_expr [, offset] [, default] )
   OVER ( [query_partition_clause] order_by_clause )
```

offset is an optional parameter and defaults to 1. *default* is an optional parameter and is the value returned if *offset* falls outside the bounds of the table or partition.

Example 21–14 LAG/LEAD

```
SELECT time_id, TO_CHAR(SUM(amount_sold),'9,999,999') AS SALES,
  TO_CHAR(LAG(SUM(amount_sold),1) OVER (ORDER BY time_id),'9,999,999') AS LAG1,
  TO_CHAR(LEAD(SUM(amount_sold),1) OVER (ORDER BY time_id),'9,999,999') AS LEAD1
FROM sales
WHERE time_id>=TO_DATE('10-OCT-2000') AND time_id<=TO_DATE('14-OCT-2000')
GROUP BY time_id;

TIME_ID   SALES      LAG1       LEAD1
--------- ---------- ---------- ----------
10-OCT-00    238,479                 23,183
11-OCT-00     23,183    238,479     24,616
12-OCT-00     24,616     23,183     76,516
13-OCT-00     76,516     24,616     29,795
14-OCT-00     29,795     76,516
```

See "Data Densification for Reporting" for information showing how to use the LAG / LEAD functions for doing period-to-period comparison queries on sparse data.

FIRST/LAST Functions

The FIRST/LAST aggregate functions allow you to rank a data set and work with its top-ranked or bottom-ranked rows. After finding the top or bottom ranked rows, an aggregate function is applied to any desired column. That is, FIRST / LAST lets you rank on column A but return the result of an aggregate applied on the first-ranked or last-ranked rows of column B. This is valuable because it avoids the need for a self-join or subquery, thus improving performance. These functions' syntax begins with a regular aggregate function (MIN, MAX, SUM, AVG, COUNT, VARIANCE, STDDEV) that produces a single return value per group. To specify the ranking used, the FIRST / LAST functions add a new clause starting with the word KEEP.

FIRST/LAST Syntax

These functions have the following syntax:

```
aggregate_function KEEP ( DENSE_RANK LAST ORDER BY
  expr [ DESC | ASC ] [NULLS { FIRST | LAST }]
  [, expr [ DESC | ASC ] [NULLS { FIRST | LAST }]]...)
[OVER query_partitioning_clause]
```

Note that the ORDER BY clause can take multiple expressions.

FIRST/LAST As Regular Aggregates

You can use the FIRST / LAST family of aggregates as regular aggregate functions.

Example 21–15 FIRST/LAST Example 1

The following query lets us compare minimum price and list price of our products. For each product subcategory within the Men's clothing category, it returns the following:

- List price of the product with the lowest minimum price
- Lowest minimum price
- List price of the product with the highest minimum price
- Highest minimum price

```
SELECT prod_subcategory, MIN(prod_list_price)
  KEEP (DENSE_RANK FIRST ORDER BY (prod_min_price)) AS LP_OF_LO_MINP,
MIN(prod_min_price) AS LO_MINP,
MAX(prod_list_price) KEEP (DENSE_RANK LAST ORDER BY (prod_min_price))
```

```
 AS LP_OF_HI_MINP,
MAX(prod_min_price) AS HI_MINP
FROM products WHERE prod_category='Electronics'
GROUP BY prod_subcategory;

PROD_SUBCATEGORY LP_OF_LO_MINP LO_MINP  LP_OF_HI_MINP  HI_MINP
---------------- ------------ -------  ------------ ----------
Game Consoles          299.99  299.99          299.99     299.99
Home Audio             499.99  499.99          599.99     599.99
Y Box Accessories        7.99    7.99           20.99      20.99
Y Box Games              7.99   7.99            29.99      29.99
```

FIRST/LAST As Reporting Aggregates

You can also use the FIRST/LAST family of aggregates as reporting aggregate functions. An example is calculating which months had the greatest and least increase in head count throughout the year. The syntax for these functions is similar to the syntax for any other reporting aggregate.

Consider the example in Example 21–15 for FIRST/LAST. What if we wanted to find the list prices of individual products and compare them to the list prices of the products in their subcategory that had the highest and lowest minimum prices?

The following query lets us find that information for the Documentation subcategory by using FIRST/LAST as reporting aggregates.

Example 21–16 FIRST/LAST Example 2

```
SELECT prod_id, prod_list_price,
    MIN(prod_list_price) KEEP (DENSE_RANK FIRST ORDER BY (prod_min_price))
       OVER(PARTITION BY (prod_subcategory)) AS LP_OF_LO_MINP,
    MAX(prod_list_price) KEEP (DENSE_RANK LAST ORDER BY (prod_min_price))
       OVER(PARTITION BY (prod_subcategory)) AS LP_OF_HI_MINP
FROM products WHERE prod_subcategory = 'Documentation';

   PROD_ID PROD_LIST_PRICE LP_OF_LO_MINP LP_OF_HI_MINP
---------- --------------- ------------- -------------
        40           44.99         44.99         44.99
        41           44.99         44.99         44.99
        42           44.99         44.99         44.99
        43           44.99         44.99         44.99
        44           44.99         44.99         44.99
        45           44.99         44.99         44.99
```

Using the FIRST and LAST functions as reporting aggregates makes it easy to include the results in calculations such "Salary as a percent of the highest salary."

Inverse Percentile Functions

Using the CUME_DIST function, you can find the cumulative distribution (percentile) of a set of values. However, the inverse operation (finding what value computes to a certain percentile) is neither easy to do nor efficiently computed. To overcome this difficulty, the PERCENTILE_CONT and PERCENTILE_DISC functions were introduced. These can be used both as window reporting functions as well as normal aggregate functions.

These functions need a sort specification and a parameter that takes a percentile value between 0 and 1. The sort specification is handled by using an ORDER BY clause with one expression. When used as a normal aggregate function, it returns a single value for each ordered set.

PERCENTILE_CONT, which is a continuous function computed by interpolation, and PERCENTILE_DISC, which is a step function that assumes discrete values. Like other aggregates, PERCENTILE_CONT and PERCENTILE_DISC operate on a group of rows in a grouped query, but with the following differences:

- They require a parameter between 0 and 1 (inclusive). A parameter specified out of this range will result in error. This parameter should be specified as an expression that evaluates to a constant.
- They require a sort specification. This sort specification is an ORDER BY clause with a single expression. Multiple expressions are not allowed.

Normal Aggregate Syntax

```
[PERCENTILE_CONT | PERCENTILE_DISC]( constant expression )
    WITHIN GROUP ( ORDER BY single order by expression
[ASC|DESC] [NULLS FIRST| NULLS LAST])
```

Inverse Percentile Example Basis

We use the following query to return the 17 rows of data used in the examples of this section:

```
SELECT cust_id, cust_credit_limit, CUME_DIST()
   OVER (ORDER BY cust_credit_limit) AS CUME_DIST
FROM customers WHERE cust_city='Marshal';

   CUST_ID CUST_CREDIT_LIMIT  CUME_DIST
```

```
---------- ---------------- ----------
     28344             1500 .173913043
      8962             1500 .173913043
     36651             1500 .173913043
     32497             1500 .173913043
     15192             3000 .347826087
    102077             3000 .347826087
    102343             3000 .347826087
      8270             3000 .347826087
     21380             5000  .52173913
     13808             5000  .52173913
    101784             5000  .52173913
     30420             5000  .52173913
     10346             7000 .652173913
     31112             7000 .652173913
     35266             7000 .652173913
      3424             9000 .739130435
    100977             9000 .739130435
    103066            10000 .782608696
     35225            11000 .956521739
     14459            11000 .956521739
     17268            11000 .956521739
    100421            11000 .956521739
     41496            15000          1
```

PERCENTILE_DISC(x) is computed by scanning up the CUME_DIST values in each group till you find the first one greater than or equal to x, where x is the specified percentile value. For the example query where PERCENTILE_DISC(0.5), the result is 5,000, as the following illustrates:

```
SELECT PERCENTILE_DISC(0.5) WITHIN GROUP
  (ORDER BY cust_credit_limit) AS perc_disc, PERCENTILE_CONT(0.5) WITHIN GROUP
  (ORDER BY cust_credit_limit) AS perc_cont
 FROM customers WHERE cust_city='Marshal';

PERC_DISC   PERC_CONT
---------   ---------
     5000        5000
```

The result of PERCENTILE_CONT is computed by linear interpolation between rows after ordering them. To compute PERCENTILE_CONT(x), we first compute the row number = RN= (1+x*(n-1)), where *n* is the number of rows in the group and *x* is the specified percentile value. The final result of the aggregate function is computed by

linear interpolation between the values from rows at row numbers CRN = CEIL(RN) and FRN = FLOOR(RN).

The final result will be: PERCENTILE_CONT(X) = if (CRN = FRN = RN), then (value of expression from row at RN) else (CRN - RN) * (value of expression for row at FRN) + (RN -FRN) * (value of expression for row at CRN).

Consider the previous example query, where we compute PERCENTILE_CONT(0.5). Here *n* is 17. The row number RN = (1 + 0.5*(n-1))= 9 for both groups. Putting this into the formula, (FRN=CRN=9), we return the value from row 9 as the result.

Another example is, if you want to compute PERCENTILE_CONT(0.66). The computed row number RN=(1 + 0.66*(n-1))= (1 + 0.66*16)= 11.67. PERCENTILE_CONT(0.66) = (12-11.67)*(value of row 11)+(11.67-11)*(value of row 12). These results are:

```
SELECT PERCENTILE_DISC(0.66) WITHIN GROUP
  (ORDER BY cust_credit_limit) AS perc_disc, PERCENTILE_CONT(0.66) WITHIN GROUP
  (ORDER BY cust_credit_limit) AS perc_cont
FROM customers WHERE cust_city='Marshal';

 PERC_DISC  PERC_CONT
---------- ----------
      9000       8040
```

Inverse percentile aggregate functions can appear in the HAVING clause of a query like other existing aggregate functions.

As Reporting Aggregates

You can also use the aggregate functions PERCENTILE_CONT, PERCENTILE_DISC as reporting aggregate functions. When used as reporting aggregate functions, the syntax is similar to those of other reporting aggregates.

```
[PERCENTILE_CONT | PERCENTILE_DISC](constant expression)
WITHIN GROUP ( ORDER BY single order by expression
[ASC|DESC] [NULLS FIRST| NULLS LAST])
OVER ( [PARTITION BY value expression [,...]] )
```

This query computes the same thing (median credit limit for customers in this result set, but reports the result for every row in the result set, as shown in the following output:

```
SELECT cust_id, cust_credit_limit, PERCENTILE_DISC(0.5) WITHIN GROUP
     (ORDER BY cust_credit_limit) OVER () AS perc_disc,
```

```
  PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY cust_credit_limit)
  OVER () AS perc_cont
FROM customers WHERE cust_city='Marshal';

   CUST_ID CUST_CREDIT_LIMIT  PERC_DISC  PERC_CONT
---------- ----------------- ---------- ----------
     28344              1500       5000       5000
      8962              1500       5000       5000
     36651              1500       5000       5000
     32497              1500       5000       5000
     15192              3000       5000       5000
    102077              3000       5000       5000
    102343              3000       5000       5000
      8270              3000       5000       5000
     21380              5000       5000       5000
     13808              5000       5000       5000
    101784              5000       5000       5000
     30420              5000       5000       5000
     10346              7000       5000       5000
     31112              7000       5000       5000
     35266              7000       5000       5000
      3424              9000       5000       5000
    100977              9000       5000       5000
    103066             10000       5000       5000
     35225             11000       5000       5000
     14459             11000       5000       5000
     17268             11000       5000       5000
    100421             11000       5000       5000
     41496             15000       5000       5000
```

Inverse Percentile Restrictions

For PERCENTILE_DISC, the expression in the ORDER BY clause can be of any data type that you can sort (numeric, string, date, and so on). However, the expression in the ORDER BY clause must be a numeric or datetime type (including intervals) because linear interpolation is used to evaluate PERCENTILE_CONT. If the expression is of type DATE, the interpolated result is rounded to the smallest unit for the type. For a DATE type, the interpolated value will be rounded to the nearest second, for interval types to the nearest second (INTERVAL DAY TO SECOND) or to the month(INTERVAL YEAR TO MONTH).

Like other aggregates, the inverse percentile functions ignore NULLS in evaluating the result. For example, when you want to find the median value in a set, Oracle Database ignores the NULLS and finds the median among the non-null values. You

can use the NULLS FIRST/NULLS LAST option in the ORDER BY clause, but they will be ignored as NULLS are ignored.

Hypothetical Rank and Distribution Functions

These functions provide functionality useful for what-if analysis. As an example, what would be the rank of a row, if the row was hypothetically inserted into a set of other rows?

This family of aggregates takes one or more arguments of a hypothetical row and an ordered group of rows, returning the RANK, DENSE_RANK, PERCENT_RANK or CUME_DIST of the row as if it was hypothetically inserted into the group.

Hypothetical Rank and Distribution Syntax

```
[RANK | DENSE_RANK | PERCENT_RANK | CUME_DIST]( constant expression [, ...] )
WITHIN GROUP ( ORDER BY order by expression [ASC|DESC] [NULLS FIRST|NULLS
LAST][, ...] )
```

Here, *constant expression* refers to an expression that evaluates to a constant, and there may be more than one such expressions that are passed as arguments to the function. The ORDER BY clause can contain one or more expressions that define the sorting order on which the ranking will be based. ASC, DESC, NULLS FIRST, NULLS LAST options will be available for each expression in the ORDER BY.

Example 21–17 Hypothetical Rank and Distribution Example 1

Using the list price data from the products table used throughout this section, you can calculate the RANK, PERCENT_RANK and CUME_DIST for a hypothetical sweater with a price of $50 for how it fits within each of the sweater subcategories. The query and results are:

```
SELECT cust_city,
RANK(6000) WITHIN GROUP (ORDER BY CUST_CREDIT_LIMIT DESC) AS HRANK,
TO_CHAR(PERCENT_RANK(6000) WITHIN GROUP
(ORDER BY cust_credit_limit),'9.999') AS HPERC_RANK,
TO_CHAR(CUME_DIST (6000) WITHIN GROUP
(ORDER BY cust_credit_limit),'9.999') AS HCUME_DIST
FROM customers
WHERE cust_city LIKE 'Fo%'
GROUP BY cust_city;

CUST_CITY                          HRANK HPERC_ HCUME_
------------------------------ ---------- ------ ------
```

```
Fondettes                          13   .455   .478
Fords Prairie                      18   .320   .346
Forest City                        47   .370   .378
Forest Heights                     38   .456   .464
Forestville                        58   .412   .418
Forrestcity                        51   .438   .444
Fort Klamath                       59   .356   .363
Fort William                       30   .500   .508
Foxborough                         52   .414   .420
```

Unlike the inverse percentile aggregates, the ORDER BY clause in the sort specification for hypothetical rank and distribution functions may take multiple expressions. The number of arguments and the expressions in the ORDER BY clause should be the same and the arguments must be constant expressions of the same or compatible type to the corresponding ORDER BY expression. The following is an example using two arguments in several hypothetical ranking functions.

Example 21–18 Hypothetical Rank and Distribution Example 2

```
SELECT prod_subcategory,
RANK(10,8) WITHIN GROUP (ORDER BY prod_list_price DESC,prod_min_price) AS
HRANK, TO_CHAR(PERCENT_RANK(10,8) WITHIN GROUP
(ORDER BY prod_list_price, prod_min_price),'9.999') AS HPERC_RANK,
TO_CHAR(CUME_DIST (10,8) WITHIN GROUP
(ORDER BY prod_list_price, prod_min_price),'9.999') AS HCUME_DIST
FROM products WHERE prod_subcategory LIKE 'Recordable%'
GROUP BY prod_subcategory;

PROD_SUBCATEGORY          HRANK   HPERC_   HCUME_
--------------------      -----   ------   ------
Recordable CDs                4     .571     .625
Recordable DVD Discs          5     .200     .333
```

These functions can appear in the HAVING clause of a query just like other aggregate functions. They cannot be used as either reporting aggregate functions or windowing aggregate functions.

Linear Regression Functions

The regression functions support the fitting of an ordinary-least-squares regression line to a set of number pairs. You can use them as both aggregate functions or windowing or reporting functions.

The functions are as follows:

- REGR_COUNT Function
- REGR_AVGY and REGR_AVGX Functions
- REGR_SLOPE and REGR_INTERCEPT Functions
- REGR_R2 Function
- REGR_SXX, REGR_SYY, and REGR_SXY Functions

Oracle applies the function to the set of (e1, e2) pairs after eliminating all pairs for which either of e1 or e2 is null. e1 is interpreted as a value of the dependent variable (a "y value"), and e2 is interpreted as a value of the independent variable (an "x value"). Both expressions must be numbers.

The regression functions are all computed simultaneously during a single pass through the data. They are frequently combined with the COVAR_POP, COVAR_SAMP, and CORR functions.

REGR_COUNT Function

REGR_COUNT returns the number of non-null number pairs used to fit the regression line. If applied to an empty set (or if there are no (e1, e2) pairs where neither of e1 or e2 is null), the function returns 0.

REGR_AVGY and REGR_AVGX Functions

REGR_AVGY and REGR_AVGX compute the averages of the dependent variable and the independent variable of the regression line, respectively. REGR_AVGY computes the average of its first argument (e1) after eliminating (e1, e2) pairs where either of e1 or e2 is null. Similarly, REGR_AVGX computes the average of its second argument (e2) after null elimination. Both functions return NULL if applied to an empty set.

REGR_SLOPE and REGR_INTERCEPT Functions

The REGR_SLOPE function computes the slope of the regression line fitted to non-null (e1, e2) pairs.

The REGR_INTERCEPT function computes the y-intercept of the regression line. REGR_INTERCEPT returns NULL whenever slope or the regression averages are NULL.

REGR_R2 Function

The REGR_R2 function computes the coefficient of determination (usually called "R-squared" or "goodness of fit") for the regression line.

REGR_R2 returns values between 0 and 1 when the regression line is defined (slope of the line is not null), and it returns NULL otherwise. The closer the value is to 1, the better the regression line fits the data.

REGR_SXX, REGR_SYY, and REGR_SXY Functions

REGR_SXX, REGR_SYY and REGR_SXY functions are used in computing various diagnostic statistics for regression analysis. After eliminating (e1, e2) pairs where either of e1 or e2 is null, these functions make the following computations:

```
REGR_SXX:    REGR_COUNT(e1,e2) * VAR_POP(e2)
REGR_SYY:    REGR_COUNT(e1,e2) * VAR_POP(e1)
REGR_SXY:    REGR_COUNT(e1,e2) * COVAR_POP(e1, e2)
```

Linear Regression Statistics Examples

Some common diagnostic statistics that accompany linear regression analysis are given in Table 21–2, "Common Diagnostic Statistics and Their Expressions". Note that this release's new functions allow you to calculate all of these.

Table 21–2 Common Diagnostic Statistics and Their Expressions

Type of Statistic	Expression
Adjusted R2	`1-((1 - REGR_R2)*((REGR_COUNT-1)/(REGR_COUNT-2)))`
Standard error	`SQRT((REGR_SYY-(POWER(REGR_SXY,2)/REGR_SXX))/(REGR_COUNT-2))`
Total sum of squares	`REGR_SYY`
Regression sum of squares	`POWER(REGR_SXY,2) / REGR_SXX`
Residual sum of squares	`REGR_SYY-(POWER(REGR_SXY,2)/REGR_SXX)`
t statistic for slope	`REGR_SLOPE * SQRT(REGR_SXX)` / (Standard error)
t statistic for y-intercept	`REGR_INTERCEPT` / ((Standard error) * `SQRT((1/REGR_COUNT)+(POWER(REGR_AVGX,2)/REGR_SXX))`

Sample Linear Regression Calculation

In this example, we compute an ordinary-least-squares regression line that expresses the quantity sold of a product as a linear function of the product's list price. The calculations are grouped by sales channel. The values SLOPE, INTCPT,

RSQR are slope, intercept, and coefficient of determination of the regression line, respectively. The (integer) value COUNT is the number of products in each channel for whom both quantity sold and list price data are available.

```
SELECT s.channel_id, REGR_SLOPE(s.quantity_sold, p.prod_list_price) SLOPE,
  REGR_INTERCEPT(s.quantity_sold, p.prod_list_price) INTCPT,
  REGR_R2(s.quantity_sold, p.prod_list_price) RSQR,
  REGR_COUNT(s.quantity_sold, p.prod_list_price) COUNT,
  REGR_AVGX(s.quantity_sold, p.prod_list_price) AVGLISTP,
  REGR_AVGY(s.quantity_sold, p.prod_list_price) AVGQSOLD
FROM sales s, products p WHERE s.prod_id=p.prod_id
  AND p.prod_category='Electronics' AND s.time_id=to_DATE('10-OCT-2000')
GROUP BY s.channel_id;

CHANNEL_ID      SLOPE     INTCPT       RSQR      COUNT   AVGLISTP   AVGQSOLD
---------- ---------- ---------- ---------- ---------- ---------- ----------
         2          0          1          1         39 466.656667          1
         3          0          1          1         60     459.99          1
         4          0          1          1         19 526.305789          1
```

Frequent Itemsets

Instead of counting how often a given event occurs (for example, how often someone has purchased milk at the grocery), frequent itemsets provides a mechanism for counting how often multiple events occur together (for example, how often someone has purchased both milk and cereal together at the grocery store).

The input to the frequent-itemsets operation is a set of data that represents collections of items (itemsets). Some examples of itemsets could be all of the products that a given customer purchased in a single trip to the grocery store (commonly called a market basket), the web-pages that a user accessed in a single session, or the financial services that a given customer utilizes. The notion of a frequent itemset is to find those itemsets that occur most often. If you apply the frequent-itemset operator to a grocery store's point-of-sale data, you might, for example, discover that milk and bananas are the most commonly bought pair of items.

Frequent itemsets have thus been used in business intelligence environments for many years, with the most common one being for market basket analysis in the retail industry. Frequent itemsets are integrated with the database, operating on top of relational tables and accessed through SQL. This integration provides a couple of key benefits:

- Applications that previously relied on frequent itemset operations now benefit from significantly improved performance as well as simpler implementation.
- SQL-based applications that did not previously use frequent itemsets can now be easily extended to take advantage of this functionality.

Frequent itemsets analysis is performed with the PL/SQL package DBMS_FREQUENT_ITEMSETS. See *PL/SQL Packages and Types Reference* for more information.

Other Statistical Functions

Oracle introduces a set of SQL statistical functions and a statistics package, DBMS_STAT_FUNCS. This section lists some of the new functions along with basic syntax.

See *PL/SQL Packages and Types Reference* for detailed information about the DBMS_STAT_FUNCS package and *Oracle Database SQL Reference* for syntax and semantics.

Descriptive Statistics

You can calculate the following descriptive statistics:

- Median of a Data Set

  ```
  Median (expr) [OVER (query_partition_clause)]
  ```

- Mode of a Data Set

  ```
  STATS_MODE (expr)
  ```

Hypothesis Testing - Parametric Tests

You can calculate the following descriptive statistics:

- One-Sample T-Test

  ```
  STATS_T_TEST_ONE (expr1, expr2 (a constant) [, return_value])
  ```

- Paired-Samples T-Test

  ```
  STATS_T_TEST_PAIRED (expr1, expr2 [, return_value])
  ```

- Independent-Samples T-Test. Pooled Variances

  ```
  STATS_T_TEST_INDEP (expr1, expr2 [, return_value])
  ```

- Independent-Samples T-Test, Unpooled Variances

```
STATS_T_TEST_INDEPU (expr1, expr2 [, return_value])
```

- The F-Test

  ```
  STATS_F_TEST (expr1, expr2 [, return_value])
  ```

- One-Way ANOVA

  ```
  STATS_ONE_WAY_ANOVA (expr1, expr2 [, return_value])
  ```

Crosstab Statistics

You can calculate crosstab statistics using the following syntax:

```
STATS_CROSSTAB (expr1, expr2 [, return_value])
```

Can return any one of the following:

- Observed value of chi-squared
- Significance of observed chi-squared
- Degree of freedom for chi-squared
- Phi coefficient, Cramer's V statistic
- Contingency coefficient
- Cohen's Kappa

Hypothesis Testing - Non-Parametric Tests

You can calculate hypothesis statistics using the following syntax:

```
STATS_BINOMIAL_TEST (expr1, expr2, p [, return_value])
```

- Binomial Test / Wilcoxon Signed Ranks Test

  ```
  STATS_WSR_TEST (expr1, expr2 [, return_value])
  ```

- Mann-Whitney Test

  ```
  STATS_MW_TEST (expr1, expr2 [, return_value])
  ```

- Kolmogorov-Smirnov Test

  ```
  STATS_KS_TEST (expr1, expr2 [, return_value])
  ```

Non-Parametric Correlation

You can calculate the following parametric statistics:

- Spearman's rho Coefficient

  ```
  CORR_S (expr1, expr2 [, return_value])
  ```

- Kendall's tau-b Coefficient

  ```
  CORR_K (expr1, expr2 [, return_value])
  ```

In addition to the functions, this release has a new PL/SQL package, DBMS_STAT_FUNCS. It contains the descriptive statistical function SUMMARY along with functions to support distribution fitting. The SUMMARY function summarizes a numerical column of a table with a variety of descriptive statistics. The five distribution fitting functions support normal, uniform, Weibull, Poisson, and exponential distributions.

WIDTH_BUCKET Function

For a given expression, the WIDTH_BUCKET function returns the bucket number that the result of this expression will be assigned after it is evaluated. You can generate equiwidth histograms with this function. Equiwidth histograms divide data sets into buckets whose interval size (highest value to lowest value) is equal. The number of rows held by each bucket will vary. A related function, NTILE, creates equiheight buckets.

Equiwidth histograms can be generated only for numeric, date or datetime types. So the first three parameters should be all numeric expressions or all date expressions. Other types of expressions are not allowed. If the first parameter is NULL, the result is NULL. If the second or the third parameter is NULL, an error message is returned, as a NULL value cannot denote any end point (or any point) for a range in a date or numeric value dimension. The last parameter (number of buckets) should be a numeric expression that evaluates to a positive integer value; 0, NULL, or a negative value will result in an error.

Buckets are numbered from 0 to (n+1). Bucket 0 holds the count of values less than the minimum. Bucket(n+1) holds the count of values greater than or equal to the maximum specified value.

WIDTH_BUCKET Syntax

The WIDTH_BUCKET takes four expressions as parameters. The first parameter is the expression that the equiwidth histogram is for. The second and third parameters are

expressions that denote the end points of the acceptable range for the first parameter. The fourth parameter denotes the number of buckets.

```
WIDTH_BUCKET(expression, minval expression, maxval expression, num buckets)
```

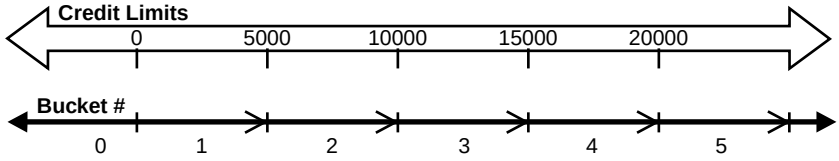
Consider the following data from table `customers`, that shows the credit limits of 17 customers. This data is gathered in the query shown in Example 21–19 on page 21-41.

```
CUST_ID    CUST_CREDIT_LIMIT
---------  -----------------
    10346               7000
    35266               7000
    41496              15000
    35225              11000
     3424               9000
    28344               1500
    31112               7000
     8962               1500
    15192               3000
    21380               5000
    36651               1500
    30420               5000
     8270               3000
    17268              11000
    14459              11000
    13808               5000
    32497               1500
   100977               9000
   102077               3000
   103066              10000
   101784               5000
   100421              11000
   102343               3000
```

In the table `customers`, the column `cust_credit_limit` contains values between 1500 and 15000, and we can assign the values to four equiwidth buckets, numbered from 1 to 4, by using `WIDTH_BUCKET (cust_credit_limit, 0, 20000, 4)`. Ideally each bucket is a closed-open interval of the real number line, for example, bucket number 2 is assigned to scores between 5000.0000 and 9999.9999..., sometimes denoted [5000, 10000) to indicate that 5,000 is included in the interval and 10,000 is excluded. To accommodate values outside the range [0, 20,000), values less than 0 are assigned to a designated underflow bucket which is numbered 0, and values greater than or equal to 20,000 are assigned to a designated

overflow bucket which is numbered 5 (num buckets + 1 in general). See Figure 21–3 for a graphical illustration of how the buckets are assigned.

Figure 21–3 Bucket Assignments



You can specify the bounds in the reverse order, for example, `WIDTH_BUCKET (cust_credit_limit, 20000, 0, 4)`. When the bounds are reversed, the buckets will be open-closed intervals. In this example, bucket number 1 is (`15000,20000`], bucket number 2 is (`10000,15000`], and bucket number 4, is (`0`,5000]. The overflow bucket will be numbered 0 (`20000`, +`infinity`), and the underflow bucket will be numbered 5 (-`infinity`, `0`].

It is an error if the bucket count parameter is 0 or negative.

Example 21–19 WIDTH_BUCKET

The following query shows the bucket numbers for the credit limits in the customers table for both cases where the boundaries are specified in regular or reverse order. We use a range of 0 to 20,000.

```
SELECT cust_id, cust_credit_limit,
 WIDTH_BUCKET(cust_credit_limit,0,20000,4) AS WIDTH_BUCKET_UP,
 WIDTH_BUCKET(cust_credit_limit,20000, 0, 4) AS WIDTH_BUCKET_DOWN
FROM customers WHERE cust_city = 'Marshal';

   CUST_ID CUST_CREDIT_LIMIT WIDTH_BUCKET_UP WIDTH_BUCKET_DOWN
---------- ----------------- --------------- -----------------
     10346              7000               2                 3
     35266              7000               2                 3
     41496             15000               4                 2
     35225             11000               3                 2
      3424              9000               2                 3
     28344              1500               1                 4
     31112              7000               2                 3
      8962              1500               1                 4
     15192              3000               1                 4
     21380              5000               2                 4
```

```
 36651          1500          1          4
 30420          5000          2          4
  8270          3000          1          4
 17268         11000          3          2
 14459         11000          3          2
 13808          5000          2          4
 32497          1500          1          4
100977          9000          2          3
102077          3000          1          4
103066         10000          3          3
101784          5000          2          4
100421         11000          3          2
102343          3000          1          4
```

User-Defined Aggregate Functions

Oracle offers a facility for creating your own functions, called user-defined aggregate functions. These functions are written in programming languages such as PL/SQL, Java, and C, and can be used as analytic functions or aggregates in materialized views. See *Oracle Data Cartridge Developer's Guide* for further information regarding syntax and restrictions.

The advantages of these functions are:

- Highly complex functions can be programmed using a fully procedural language.
- Higher scalability than other techniques when user-defined functions are programmed for parallel processing.
- Object datatypes can be processed.

As a simple example of a user-defined aggregate function, consider the skew statistic. This calculation measures if a data set has a lopsided distribution about its mean. It will tell you if one tail of the distribution is significantly larger than the other. If you created a user-defined aggregate called `udskew` and applied it to the credit limit data in the prior example, the SQL statement and results might look like this:

```
SELECT USERDEF_SKEW(cust_credit_limit) FROM customers
WHERE cust_city='Marshal';

USERDEF_SKEW
============
0.583891
```

Before building user-defined aggregate functions, you should consider if your needs can be met in regular SQL. Many complex calculations are possible directly in SQL, particularly by using the CASE expression.

Staying with regular SQL will enable simpler development, and many query operations are already well-parallelized in SQL. Even the earlier example, the skew statistic, can be created using standard, albeit lengthy, SQL.

CASE Expressions

Oracle now supports simple and searched CASE statements. CASE statements are similar in purpose to the DECODE statement, but they offer more flexibility and logical power. They are also easier to read than traditional DECODE statements, and offer better performance as well. They are commonly used when breaking categories into buckets like age (for example, 20-29, 30-39, and so on). The syntax for simple statements is:

```
expr WHEN comparison_expr THEN return_expr
 [, WHEN comparison_expr THEN return_expr]...
```

The syntax for searched statements is:

```
WHEN condition THEN return_expr [, WHEN condition THEN return_expr]...
```

You can specify only 255 arguments and each WHEN ... THEN pair counts as two arguments. For a workaround to this limit, see *Oracle Database SQL Reference*.

Example 21–20 CASE

Suppose you wanted to find the average salary of all employees in the company. If an employee's salary is less than $2000, you want the query to use $2000 instead. Without a CASE statement, you would have to write this query as follows,

```
SELECT AVG(foo(e.sal)) FROM emps e;
```

In this, foo is a function that returns its input if the input is greater than 2000, and returns 2000 otherwise. The query has performance implications because it needs to invoke a function for each row. Writing custom functions can also add to the development load.

Using CASE expressions in the database without PL/SQL, this query can be rewritten as:

```
SELECT AVG(CASE when e.sal > 2000 THEN e.sal ELSE 2000 end) FROM emps e;
```

Using a CASE expression lets you avoid developing custom functions and can also perform faster.

Creating Histograms With User-Defined Buckets

You can use the CASE statement when you want to obtain histograms with user-defined buckets (both in number of buckets and width of each bucket). The following are two examples of histograms created with CASE statements. In the first example, the histogram totals are shown in multiple columns and a single row is returned. In the second example, the histogram is shown with a label column and a single column for totals, and multiple rows are returned.

Example 21–21 Histogram Example 1

```
SELECT SUM(CASE WHEN cust_credit_limit BETWEEN  0 AND 3999 THEN 1 ELSE 0 END)
  AS "0-3999",
SUM(CASE WHEN cust_credit_limit BETWEEN  4000 AND 7999 THEN 1 ELSE 0 END)
  AS "4000-7999",
SUM(CASE WHEN cust_credit_limit BETWEEN  8000 AND 11999 THEN 1 ELSE 0 END)
  AS "8000-11999",
SUM(CASE WHEN cust_credit_limit BETWEEN  12000 AND 16000 THEN 1 ELSE 0 END)
  AS "12000-16000"
FROM customers WHERE cust_city = 'Marshal';

    0-3999  4000-7999 8000-11999 12000-16000
---------- ---------- ---------- -----------
         8          7          7           1
```

Example 21–22 Histogram Example 2

```
SELECT (CASE WHEN cust_credit_limit BETWEEN  0 AND 3999 THEN  ' 0 - 3999'
   WHEN cust_credit_limit BETWEEN  4000 AND 7999 THEN ' 4000 - 7999'
   WHEN cust_credit_limit BETWEEN  8000 AND 11999 THEN  ' 8000 - 11999'
   WHEN cust_credit_limit BETWEEN  12000 AND 16000 THEN '12000 - 16000' END)
  AS BUCKET, COUNT(*) AS Count_in_Group
FROM customers WHERE cust_city = 'Marshal' GROUP BY
 (CASE WHEN cust_credit_limit BETWEEN  0 AND 3999 THEN ' 0 - 3999'
 WHEN cust_credit_limit BETWEEN  4000 AND 7999 THEN ' 4000 - 7999'
 WHEN cust_credit_limit BETWEEN  8000 AND 11999 THEN  ' 8000 - 11999'
 WHEN cust_credit_limit BETWEEN  12000 AND 16000 THEN '12000 - 16000' END);

BUCKET        COUNT_IN_GROUP
------------- --------------
 0 - 3999                  8
```

```
 4000 - 7999              7
 8000 - 11999             7
12000 - 16000             1
```

Data Densification for Reporting

Data is normally stored in sparse form. That is, if no value exists for a given combination of dimension values, no row exists in the fact table. However, you may want to view the data in dense form, with rows for all combination of dimension values displayed even when no fact data exist for them. For example, if a product did not sell during a particular time period, you may still want to see the product for that time period with zero sales value next to it. Moreover, time series calculations can be performed most easily when data is dense along the time dimension. This is because dense data will fill a consistent number of rows for each period, which in turn makes it simple to use the analytic windowing functions with physical offsets. Data densification is the process of converting spare data into dense form.

To overcome the problem of sparsity, you can use a partitioned outer join to fill the gaps in a time series or any other dimension. Such a join extends the conventional outer join syntax by applying the outer join to each logical partition defined in a query. Oracle logically partitions the rows in your query based on the expression you specify in the PARTITION BY clause. The result of a partitioned outer join is a UNION of the outer joins of each of the partitions in the logically partitioned table with the table on the other side of the join.

Note that you can use this type of join to fill the gaps in any dimension, not just the time dimension. Most of the examples here focus on the time dimension because it is the dimension most frequently used as a basis for comparisons.

Partition Join Syntax

The syntax for partitioned outer join extends the ANSI SQL JOIN clause with the phrase PARTITION BY followed by an expression list. The expressions in the list specify the group to which the outer join is applied. The following are the two forms of syntax normally used for partitioned outer join:

```
SELECT .....
FROM table_reference
PARTITION BY (expr [, expr ]... )
RIGHT OUTER JOIN table_reference

SELECT .....
```

```
FROM table_reference
LEFT OUTER JOIN table_reference
PARTITION BY {expr [,expr ]...)
```

Note that FULL OUTER JOIN is not supported with a partitioned outer join.

Sample of Sparse Data

A typical situation with a sparse dimension is shown in the following example, which computes the weekly sales and year-to-date sales for the product Bounce for weeks 20-30 in 2000 and 2001:

```
SELECT SUBSTR(p.Prod_Name,1,15) Product_Name, t.Calendar_Year Year,
 t.Calendar_Week_Number Week, SUM(Amount_Sold) Sales
FROM Sales s, Times t, Products p
WHERE s.Time_id = t.Time_id AND s.Prod_id = p.Prod_id AND
 p.Prod_name IN ('Bounce') AND t.Calendar_Year IN (2000,2001) AND
 t.Calendar_Week_Number BETWEEN 20 AND 30
GROUP BY p.Prod_Name, t.Calendar_Year, t.Calendar_Week_Number;

PRODUCT_NAME          YEAR       WEEK      SALES
--------------- ---------- ---------- ----------
Bounce                2000         20        801
Bounce                2000         21    4062.24
Bounce                2000         22    2043.16
Bounce                2000         23    2731.14
Bounce                2000         24    4419.36
Bounce                2000         27    2297.29
Bounce                2000         28    1443.13
Bounce                2000         29    1927.38
Bounce                2000         30    1927.38
Bounce                2001         20     1483.3
Bounce                2001         21    4184.49
Bounce                2001         22    2609.19
Bounce                2001         23    1416.95
Bounce                2001         24    3149.62
Bounce                2001         25    2645.98
Bounce                2001         27    2125.12
Bounce                2001         29    2467.92
Bounce                2001         30    2620.17
```

In this example, we would expect 22 rows of data (11 weeks each from 2 years) if the data were dense. However we get only 18 rows because weeks 25 and 26 are missing in 2000, and weeks 26 and 28 in 2001.

Filling Gaps in Data

We can take the sparse data of the preceding query and do a partitioned outer join with a dense set of time data. In the following query, we alias our original query as `v` and we select data from the `times` table, which we alias as `t`. Here we retrieve 22 rows because there are no gaps in the series. The four added rows each have 0 as their Sales value set to 0 by using the `NVL` function.

```
SELECT Product_Name, t.Year, t.Week, NVL(Sales,0) dense_sales
FROM
 (SELECT SUBSTR(p.Prod_Name,1,15) Product_Name,
  t.Calendar_Year Year, t.Calendar_Week_Number Week, SUM(Amount_Sold) Sales
  FROM Sales s, Times t, Products p
  WHERE s.Time_id = t.Time_id AND s.Prod_id = p.Prod_id AND
   p.Prod_name IN ('Bounce') AND t.Calendar_Year IN (2000,2001) AND
   t.Calendar_Week_Number BETWEEN 20 AND 30
  GROUP BY p.Prod_Name, t.Calendar_Year, t.Calendar_Week_Number) v
PARTITION BY (v.Product_Name)
RIGHT OUTER JOIN
 (SELECT DISTINCT Calendar_Week_Number Week, Calendar_Year Year
  FROM Times
  WHERE Calendar_Year IN (2000, 2001)
  AND Calendar_Week_Number BETWEEN 20 AND 30) t
ON (v.week = t.week AND v.Year = t.Year)
ORDER BY t.year, t.week;

PRODUCT_NAME          YEAR       WEEK DENSE_SALES
--------------- ---------- ---------- -----------
Bounce                2000         20         801
Bounce                2000         21     4062.24
Bounce                2000         22     2043.16
Bounce                2000         23     2731.14
Bounce                2000         24     4419.36
Bounce                2000         25           0
Bounce                2000         26           0
Bounce                2000         27     2297.29
Bounce                2000         28     1443.13
Bounce                2000         29     1927.38
Bounce                2000         30     1927.38
Bounce                2001         20      1483.3
Bounce                2001         21     4184.49
Bounce                2001         22     2609.19
Bounce                2001         23     1416.95
Bounce                2001         24     3149.62
Bounce                2001         25     2645.98
```

```
Bounce                2001         26           0
Bounce                2001         27     2125.12
Bounce                2001         28           0
Bounce                2001         29     2467.92
Bounce                2001         30     2620.17
```

Note that in this query, a WHERE condition was placed for weeks between 20 and 30 in the inline view for the time dimension. This was introduced to keep the result set small.

Filling Gaps in Two Dimensions

N-dimensional data is typically displayed as a dense 2-dimensional cross tab of (n - 2) page dimensions. This requires that all dimension values for the two dimensions appearing in the cross tab be filled in. The following is another example where the partitioned outer join capability can be used for filling the gaps on two dimensions:

```
WITH v1 AS
 (SELECT p.prod_id, country_id, calendar_year,
    SUM(quantity_sold) units, SUM(amount_sold) sales
  FROM sales s, products p, customers c, times t
  WHERE s.prod_id in (147, 148) AND t.time_id = s.time_id AND
    c.cust_id = s.cust_id AND p.prod_id = s.prod_id
  GROUP BY p.prod_id, country_id, calendar_year),
v2 AS                          -- countries to use for densifications.
  (SELECT DISTINCT country_id
  FROM customers
  WHERE country_id IN (52782, 52785, 52786, 52787, 52788)),
v3 AS                          --  years to use for densifications
  (SELECT DISTINCT calendar_year FROM times)
SELECT v4.prod_id, v4.country_id, v3.calendar_year, units, sales
FROM
  (SELECT prod_id, v2.country_id, calendar_year, units, sales
   FROM v1 PARTITION BY (prod_id)
   RIGHT OUTER JOIN v2               --densifies on country
   ON (v1.country_id = v2.country_id)) v4
PARTITION BY (prod_id,country_id)
RIGHT OUTER JOIN v3                   --densifies on year
ON (v4.calendar_year = v3.calendar_year)
ORDER BY 1, 2, 3;
```

In this query, the WITH sub-query factoring clause v1, summarizes sales data at the product, country, and year level. This result is sparse but users may want to see all the country, year combinations for each product. To achieve this, we take each

partition of v1 based on product values and outer join it on the country dimension first. This will give us all values of country for each product. We then take that result and partition it on product and country values and then outer join it on time dimension. This will give us all time values for each product and country combination.

PROD_ID	COUNTRY_ID	CALENDAR_YEAR	UNITS	SALES
147	52782	1998		
147	52782	1999	29	209.82
147	52782	2000	71	594.36
147	52782	2001	345	2754.42
147	52782	2002		
147	52785	1998	1	7.99
147	52785	1999		
147	52785	2000		
147	52785	2001		
147	52785	2002		
147	52786	1998	1	7.99
147	52786	1999		
147	52786	2000	2	15.98
147	52786	2001		
147	52786	2002		
147	52787	1998		
147	52787	1999		
147	52787	2000		
147	52787	2001		
147	52787	2002		
147	52788	1998		
147	52788	1999		
147	52788	2000	1	7.99
147	52788	2001		
147	52788	2002		
148	52782	1998	139	4046.67
148	52782	1999	228	5362.57
148	52782	2000	251	5629.47
148	52782	2001	308	7138.98
148	52782	2002		
148	52785	1998		
148	52785	1999		
148	52785	2000		
148	52785	2001		
148	52785	2002		
148	52786	1998		
148	52786	1999		

```
       148      52786          2000
       148      52786          2001
       148      52786          2002
       148      52787          1998
       148      52787          1999
       148      52787          2000
       148      52787          2001
       148      52787          2002
       148      52788          1998          4     117.23
       148      52788          1999
       148      52788          2000
       148      52788          2001
       148      52788          2002
```

Filling Gaps in an Inventory Table

An inventory table typically tracks quantity of units available for various products. This table is sparse: it only stores a row for a product when there is an event. For a sales table, the event is a sale, and for the inventory table, the event is a change in quantity available for a product. For example, consider the following inventory table:

```
CREATE TABLE invent_table (
product VARCHAR2(10),
time_id DATE,
quant NUMBER);

INSERT INTO invent_table VALUES
 ('bottle', TO_DATE('01/04/01', 'DD/MM/YY'), 10);
INSERT INTO invent_table VALUES
 ('bottle', TO_DATE('06/04/01', 'DD/MM/YY'), 8);
INSERT INTO invent_table VALUES
 ('can', TO_DATE('01/04/01', 'DD/MM/YY'), 15);
INSERT INTO invent_table VALUES
 ('can', TO_DATE('04/04/01', 'DD/MM/YY'), 11);
```

The inventory table now has the following rows:

```
PRODUCT    TIME_ID   QUANT
---------- --------- -----
bottle     01-APR-01    10
bottle     06-APR-01     8
can        01-APR-01    15
can        04-APR-01    11
```

For reporting purposes, users may want to see this inventory data differently. For example, they may want to see all values of time for each product. This can be accomplished using partitioned outer join. In addition, for the newly inserted rows of missing time periods, users may want to see the values for quantity of units column to be carried over from the most recent existing time period. The latter can be accomplished using analytic window function LAST_VALUE value. Here is the query and the desired output:

```
WITH v1 AS
 (SELECT time_id
  FROM times
  WHERE times.time_id BETWEEN
   TO_DATE('01/04/01', 'DD/MM/YY')
   AND TO_DATE('07/04/01', 'DD/MM/YY'))
SELECT product, time_id, quant quantity,
  LAST_VALUE(quant IGNORE NULLS)
    OVER (PARTITION BY product ORDER BY time_id)
    repeated_quantity
FROM
 (SELECT product, v1.time_id, quant
  FROM invent_table PARTITION BY (product)
  RIGHT OUTER JOIN v1
  ON (v1.time_id = invent_table.time_id))
ORDER BY 1, 2;
```

The inner query computes a partitioned outer join on time within each product. The inner query densifies the data on the time dimension (meaning the time dimension will now have a row for each day of the week). However, the measure column quantity will have nulls for the newly added rows (see the output in the column quantity in the following results.

The outer query uses the analytic function LAST_VALUE. Applying this function partitions the data by product and orders the data on the time dimension column (time_id). For each row, the function finds the last non-null value in the window due to the option IGNORE NULLS, which you can use with both LAST_VALUE and FIRST_VALUE. We see the desired output in the column repeated_quantity in the following output:

```
PRODUCT    TIME_ID   QUANTITY REPEATED_QUANTITY
---------- --------- -------- -----------------
bottle     01-APR-01       10                10
bottle     02-APR-01                         10
bottle     03-APR-01                         10
bottle     04-APR-01                         10
bottle     05-APR-01                         10
```

```
bottle     06-APR-01        8                8
bottle     07-APR-01                         8
can        01-APR-01       15               15
can        02-APR-01                        15
can        03-APR-01                        15
can        04-APR-01       11               11
can        05-APR-01                        11
can        06-APR-01                        11
can        07-APR-01                        11
```

Computing Data Values to Fill Gaps

Examples in previous section illustrate how to use partitioned outer join to fill gaps in one or more dimensions. However, the result sets produced by partitioned outer join have null values for columns that are not included in the PARTITION BY list. Typically, these are measure columns. Users can make use of analytic SQL functions to replace those null values with a non-null value.

For example, the following query computes monthly totals for products 64MB Memory card and DVD-R Discs (product IDs 122 and 136) for the year 2000. It uses partitioned outer join to densify data for all months. For the missing months, it then uses the analytic SQL function AVG to compute the sales and units to be the average of the months when the product was sold.

If working in SQL*Plus, the following two commands will wrap the column headings for greater readability of results:

```
col computed_units  heading  'Computed|_units'
col computed_sales  heading  'Computed|_sales'

WITH V AS
 (SELECT substr(p.prod_name,1,12) prod_name, calendar_month_desc,
     SUM(quantity_sold) units, SUM(amount_sold) sales
   FROM sales s, products p, times t
   WHERE s.prod_id in (122,136) AND calendar_year = 2000
     AND t.time_id = s.time_id
     AND p.prod_id = s.prod_id
   GROUP BY p.prod_name, calendar_month_desc)
SELECT v.prod_name, calendar_month_desc, units, sales,
  NVL(units, AVG(units) OVER (partition by v.prod_name)) computed_units,
  NVL(sales, AVG(sales) OVER (partition by v.prod_name)) computed_sales
FROM
  (SELECT DISTINCT calendar_month_desc
   FROM times
   WHERE calendar_year = 2000) t
```

```
   LEFT OUTER JOIN V
   PARTITION BY (prod_name)
   USING (calendar_month_desc);
                                                    computed   computed
PROD_NAME    CALENDAR      UNITS      SALES      _units     _sales
------------ -------- ---------- ---------- ---------- ----------
64MB Memory  2000-01         112    4129.72        112    4129.72
64MB Memory  2000-02         190       7049        190       7049
64MB Memory  2000-03          47    1724.98         47    1724.98
64MB Memory  2000-04          20      739.4         20      739.4
64MB Memory  2000-05          47    1738.24         47    1738.24
64MB Memory  2000-06          20      739.4         20      739.4
64MB Memory  2000-07                          72.6666667    2686.79
64MB Memory  2000-08                          72.6666667    2686.79
64MB Memory  2000-09                          72.6666667    2686.79
64MB Memory  2000-10                          72.6666667    2686.79
64MB Memory  2000-11                          72.6666667    2686.79
64MB Memory  2000-12                          72.6666667    2686.79
DVD-R Discs, 2000-01         167     3683.5        167     3683.5
DVD-R Discs, 2000-02         152    3362.24        152    3362.24
DVD-R Discs, 2000-03         188    4148.02        188    4148.02
DVD-R Discs, 2000-04         144    3170.09        144    3170.09
DVD-R Discs, 2000-05         189    4164.87        189    4164.87
DVD-R Discs, 2000-06         145    3192.21        145    3192.21
DVD-R Discs, 2000-07                             124.25    2737.71
DVD-R Discs, 2000-08                             124.25    2737.71
DVD-R Discs, 2000-09           1      18.91          1      18.91
DVD-R Discs, 2000-10                             124.25    2737.71
DVD-R Discs, 2000-11                             124.25    2737.71
DVD-R Discs, 2000-12           8     161.84          8     161.84
```

Time Series Calculations on Densified Data

Densification is not just for reporting purpose. It also enables certain types of calculations, especially, time series calculations. Time series calculations are easier when data is dense along the time dimension. Dense data has a consistent number of rows for each time periods which in turn make it simple to use analytic window functions with physical offsets.

To illustrate, let's first take the example on "Filling Gaps in Data" on page 21-47, and let's add an analytic function to that query. In the following enhanced version, we calculate weekly year-to-date sales alongside the weekly sales. The NULL values that the partitioned outer join inserts in making the time series dense are handled in the usual way: the SUM function treats them as 0's.

```
SELECT Product_Name, t.Year, t.Week, NVL(Sales,0) Current_sales,
 SUM(Sales)
   OVER (PARTITION BY Product_Name, t.year ORDER BY t.week) Cumulative_sales
FROM
 (SELECT SUBSTR(p.Prod_Name,1,15) Product_Name, t.Calendar_Year Year,
   t.Calendar_Week_Number Week, SUM(Amount_Sold) Sales
  FROM Sales s, Times t, Products p
  WHERE s.Time_id = t.Time_id AND
   s.Prod_id = p.Prod_id AND p.Prod_name IN ('Bounce') AND
   t.Calendar_Year IN (2000,2001) AND
   t.Calendar_Week_Number BETWEEN 20 AND 30
  GROUP BY p.Prod_Name, t.Calendar_Year, t.Calendar_Week_Number) v
PARTITION BY (v.Product_Name)
RIGHT OUTER JOIN
(SELECT DISTINCT
 Calendar_Week_Number Week, Calendar_Year Year
 FROM Times
 WHERE Calendar_Year in (2000, 2001)
 AND Calendar_Week_Number BETWEEN 20 AND 30) t
ON (v.week = t.week AND v.Year = t.Year)
ORDER BY t.year, t.week;

PRODUCT_NAME          YEAR       WEEK CURRENT_SALES CUMULATIVE_SALES
--------------- ---------- ---------- ------------- ----------------
Bounce                2000         20           801              801
Bounce                2000         21       4062.24          4863.24
Bounce                2000         22       2043.16           6906.4
Bounce                2000         23       2731.14          9637.54
Bounce                2000         24       4419.36          14056.9
Bounce                2000         25             0          14056.9
Bounce                2000         26             0          14056.9
Bounce                2000         27       2297.29         16354.19
Bounce                2000         28       1443.13         17797.32
Bounce                2000         29       1927.38          19724.7
Bounce                2000         30       1927.38         21652.08
Bounce                2001         20        1483.3           1483.3
Bounce                2001         21       4184.49          5667.79
Bounce                2001         22       2609.19          8276.98
Bounce                2001         23       1416.95          9693.93
Bounce                2001         24       3149.62         12843.55
Bounce                2001         25       2645.98         15489.53
Bounce                2001         26             0         15489.53
Bounce                2001         27       2125.12         17614.65
Bounce                2001         28             0         17614.65
Bounce                2001         29       2467.92         20082.57
```

```
Bounce                2001         30       2620.17         22702.74
```

Period-to-Period Comparison for One Time Level: Example

How do we use this feature to compare values across time periods? Specifically, how do we calculate a year-over-year sales comparison at the week level? The following query returns on the same row, for each product, the year-to-date sales for each week of 2001 with that of 2000.

Note that in this example we start with a `WITH` clause. This improves readability of the query and lets us focus on the partitioned outer join. If working in SQL*Plus, the following command will wrap the column headings for greater readability of results:

```
col Weekly_ytd_sales_prior_year  heading  'Weekly_ytd|_sales_|prior_year'

WITH v AS
  (SELECT SUBSTR(p.Prod_Name,1,6) Prod, t.Calendar_Year Year,
     t.Calendar_Week_Number Week, SUM(Amount_Sold) Sales
   FROM Sales s, Times t, Products p
   WHERE s.Time_id = t.Time_id AND
     s.Prod_id = p.Prod_id AND p.Prod_name in ('Y Box') AND
     t.Calendar_Year in (2000,2001) AND
     t.Calendar_Week_Number BETWEEN 30 AND 40
   GROUP BY p.Prod_Name, t.Calendar_Year, t.Calendar_Week_Number)
SELECT Prod , Year, Week, Sales,
  Weekly_ytd_sales, Weekly_ytd_sales_prior_year
FROM
  (SELECT Prod, Year, Week, Sales, Weekly_ytd_sales,
     LAG(Weekly_ytd_sales, 1) OVER
       (PARTITION BY Prod , Week ORDER BY Year) Weekly_ytd_sales_prior_year
   FROM
     (SELECT v.Prod  Prod , t.Year Year, t.Week Week,
        NVL(v.Sales,0) Sales, SUM(NVL(v.Sales,0)) OVER
          (PARTITION BY v.Prod , t.Year ORDER BY t.week) weekly_ytd_sales
      FROM v
      PARTITION BY (v.Prod )
      RIGHT OUTER JOIN
        (SELECT DISTINCT Calendar_Week_Number Week, Calendar_Year Year
         FROM Times
         WHERE Calendar_Year IN (2000, 2001)) t
      ON (v.week = t.week AND v.Year = t.Year)
     ) dense_sales
  ) year_over_year_sales
WHERE Year = 2001 AND Week BETWEEN 30 AND 40
```

```
ORDER BY 1, 2, 3;
                                                                  Weekly_ytd
                                                                     _sales_
PROD         YEAR       WEEK      SALES WEEKLY_YTD_SALES prior_year
------ ---------- ---------- ---------- ---------------- ----------
Y Box        2001         30    7877.45          7877.45          0
Y Box        2001         31   13082.46         20959.91    1537.35
Y Box        2001         32   11569.02         32528.93    9531.57
Y Box        2001         33   38081.97          70610.9   39048.69
Y Box        2001         34   33109.65        103720.55   69100.79
Y Box        2001         35          0        103720.55   71265.35
Y Box        2001         36     4169.3        107889.85   81156.29
Y Box        2001         37   24616.85         132506.7   95433.09
Y Box        2001         38   37739.65        170246.35  107726.96
Y Box        2001         39     284.95         170531.3   118817.4
Y Box        2001         40   10868.44        181399.74  120969.69
```

In the FROM clause of the in-line view dense_sales, we use a partitioned outer join of aggregate view v and time view t to fill gaps in the sales data along the time dimension. The output of the partitioned outer join is then processed by the analytic function SUM ... OVER to compute the weekly year-to-date sales (the weekly_ytd_sales column). Thus, the view dense_sales computes the year-to-date sales data for each week, including those missing in the aggregate view s. The in-line view year_over_year_sales then computes the year ago weekly year-to-date sales using the LAG function. The LAG function labeled weekly_ytd_sales_prior_year specifies a PARTITION BY clause that pairs rows for the same week of years 2000 and 2001 into a single partition. We then pass an offset of 1 to the LAG function to get the weekly year to date sales for the prior year.

The outermost query block selects data from year_over_year_sales with the condition yr = 2001, and thus the query returns, for each product, its weekly year-to-date sales in the specified weeks of years 2001 and 2000.

Period-to-Period Comparison for Multiple Time Levels: Example

While the prior example shows us a way to create comparisons for a single time level, it would be even more useful to handle multiple time levels in a single query. For example, we could compare sales versus the prior period at the year, quarter, month and day levels. How can we create a query which performs a year-over-year comparison of year-to-date sales for all levels of our time hierarchy?

We will take several steps to perform this task. The goal is a single query with comparisons at the day, week, month, quarter, and year level. The steps are as follows:

1. We will create a view called `cube_prod_time`, which holds a hierarchical cube of sales aggregated across `times` and `products`.
2. Then we will create a view of the time dimension to use as an edge of the cube. The time edge, which holds a complete set of dates, will be partitioned outer joined to the sparse data in the view `cube_prod_time`.
3. Finally, for maximum performance, we will create a materialized view, `mv_prod_time`, built using the same definition as `cube_prod_time`.

For more information regarding hierarchical cubes, see Chapter 20, "SQL for Aggregation in Data Warehouses". The materialized view is defined using the following statement:

Step 1 Create the hierarchical cube view

The materialized view shown in the following may already exist in your system; if not, create it now. If you must generate it, please note that we limit the query to just two products to keep processing time short:

```
CREATE OR REPLACE VIEW cube_prod_time AS
SELECT
  (CASE
     WHEN ((GROUPING(calendar_year)=0 )
       AND (GROUPING(calendar_quarter_desc)=1 ))
       THEN (TO_CHAR(calendar_year) || '_0')
     WHEN ((GROUPING(calendar_quarter_desc)=0 )
       AND (GROUPING(calendar_month_desc)=1 ))
       THEN (TO_CHAR(calendar_quarter_desc) || '_1')
     WHEN ((GROUPING(calendar_month_desc)=0 )
       AND (GROUPING(t.time_id)=1 ))
       THEN (TO_CHAR(calendar_month_desc) || '_2')
     ELSE (TO_CHAR(t.time_id) || '_3')
  END) Hierarchical_Time,
  calendar_year year, calendar_quarter_desc quarter,
  calendar_month_desc month, t.time_id day,
  prod_category cat, prod_subcategory subcat, p.prod_id prod,
  GROUPING_ID(prod_category, prod_subcategory, p.prod_id,
    calendar_year, calendar_quarter_desc, calendar_month_desc,t.time_id) gid,
  GROUPING_ID(prod_category, prod_subcategory, p.prod_id) gid_p,
  GROUPING_ID(calendar_year, calendar_quarter_desc,
    calendar_month_desc, t.time_id) gid_t,
  SUM(amount_sold) s_sold, COUNT(amount_sold) c_sold, COUNT(*) cnt
FROM SALES s, TIMES t, PRODUCTS p
WHERE s.time_id = t.time_id AND
  p.prod_name IN ('Bounce', 'Y Box') AND s.prod_id = p.prod_id
```

```
GROUP BY
  ROLLUP(calendar_year, calendar_quarter_desc, calendar_month_desc, t.time_id),
  ROLLUP(prod_category, prod_subcategory, p.prod_id);
```

Because this view is limited to two products, it returns just over 2200 rows. Note that the column Hierarchical_Time contains string representations of time from all levels of the time hierarchy. The CASE expression used for the Hierarchical_Time column appends a marker (_0, _1, ...) to each date string to denote the time level of the value. A _0 represents the year level, _1 is quarters, _2 is months, and _3 is day. Note that the GROUP BY clause is a concatenated ROLLUP which specifies the rollup hierarchy for the time and product dimensions. The GROUP BY clause is what determines the hierarchical cube contents.

Step 2 Create the view edge_time, which is a complete set of date values

edge_time is the source for filling time gaps in the hierarchical cube using a partitioned outer join. The column Hierarchical_Time in edge_time will be used in a partitioned join with the Hierarchical_Time column in the view cube_prod_time. The following statement defines edge_time:

```
CREATE OR REPLACE VIEW edge_time AS
SELECT
  (CASE
     WHEN ((GROUPING(calendar_year)=0 )
       AND (GROUPING(calendar_quarter_desc)=1 ))
       THEN (TO_CHAR(calendar_year) || '_0')
     WHEN ((GROUPING(calendar_quarter_desc)=0 )
       AND (GROUPING(calendar_month_desc)=1 ))
       THEN (TO_CHAR(calendar_quarter_desc) || '_1')
     WHEN ((GROUPING(calendar_month_desc)=0 )
       AND (GROUPING(time_id)=1 ))
       THEN (TO_CHAR(calendar_month_desc) || '_2')
     ELSE (TO_CHAR(time_id) || '_3')
   END) Hierarchical_Time,
   calendar_year yr, calendar_quarter_number qtr_num,
   calendar_quarter_desc qtr, calendar_month_number mon_num,
   calendar_month_desc mon, time_id - TRUNC(time_id, 'YEAR') + 1 day_num,
   time_id day,
GROUPING_ID(calendar_year, calendar_quarter_desc,
  calendar_month_desc, time_id) gid_t
FROM TIMES
GROUP BY ROLLUP
 (calendar_year, (calendar_quarter_desc, calendar_quarter_number),
 (calendar_month_desc, calendar_month_number), time_id);
```

Step 3 Create the materialized view mv_prod_time to support faster performance

The materialized view definition is a duplicate of the view `cube_prod_time` defined earlier. Because it is a duplicate query, references to `cube_prod_time` will be rewritten to use the `mv_prod_time` materialized view. The following materialized may already exist in your system; if not, create it now. If you must generate it, please note that we limit the query to just two products to keep processing time short.

```
CREATE MATERIALIZED VIEW mv_prod_time
REFRESH COMPLETE ON DEMAND AS
SELECT
  (CASE
     WHEN ((GROUPING(calendar_year)=0 )
       AND (GROUPING(calendar_quarter_desc)=1 ))
       THEN (TO_CHAR(calendar_year) || '_0')
     WHEN ((GROUPING(calendar_quarter_desc)=0 )
       AND (GROUPING(calendar_month_desc)=1 ))
       THEN (TO_CHAR(calendar_quarter_desc) || '_1')
     WHEN ((GROUPING(calendar_month_desc)=0 )
       AND (GROUPING(t.time_id)=1 ))
       THEN (TO_CHAR(calendar_month_desc) || '_2')
     ELSE (TO_CHAR(t.time_id) || '_3')
  END) Hierarchical_Time,
  calendar_year year, calendar_quarter_desc quarter,
  calendar_month_desc month, t.time_id day,
  prod_category cat, prod_subcategory subcat, p.prod_id prod,
  GROUPING_ID(prod_category, prod_subcategory, p.prod_id,
    calendar_year, calendar_quarter_desc, calendar_month_desc,t.time_id) gid,
  GROUPING_ID(prod_category, prod_subcategory, p.prod_id) gid_p,
  GROUPING_ID(calendar_year, calendar_quarter_desc,
    calendar_month_desc, t.time_id) gid_t,
  SUM(amount_sold) s_sold, COUNT(amount_sold) c_sold, COUNT(*) cnt
FROM SALES s, TIMES t, PRODUCTS p
WHERE s.time_id = t.time_id AND
  p.prod_name IN ('Bounce', 'Y Box') AND s.prod_id = p.prod_id
GROUP BY
  ROLLUP(calendar_year, calendar_quarter_desc, calendar_month_desc, t.time_id),
  ROLLUP(prod_category, prod_subcategory, p.prod_id);
```

Step 4 Create the comparison query

We have now set the stage for our comparison query. We can obtain period-to-period comparison calculations at all time levels. It requires applying analytic functions to a hierarchical cube with dense data along the time dimension.

Some of the calculations we can achieve for each time level are:

- Sum of sales for prior period at all levels of time.
- Variance in sales over prior period.
- Sum of sales in the same period a year ago at all levels of time.
- Variance in sales over the same period last year.

The following example performs all four of these calculations. It uses a partitioned outer join of the views `cube_prod_time` and `edge_time` to create an in-line view of dense data called `dense_cube_prod_time`. The query then uses the `LAG` function in the same way as the prior single-level example. The outer `WHERE` clause specifies time at three levels: the days of August 2001, the entire month, and the entire third quarter of 2001. Note that the last two rows of the results contain the month level and quarter level aggregations.

Note: To make the results easier to read if you are using SQL*Plus, the column headings should be adjusted with the following commands. The commands will fold the column headings to reduce line length:

```
col sales_prior_period heading 'sales_prior|_period'
col variance_prior_period heading 'variance|_prior|_period'
col sales_same_period_prior_year heading 'sales_same|_period_prior|_year'
col variance_same_period_p_year heading 'variance|_same_period|_prior_year'
```

Here is the query comparing current sales to prior and year ago sales:

```
SELECT SUBSTR(prod,1,4) prod, SUBSTR(Hierarchical_Time,1,12) ht,
  sales, sales_prior_period,
  sales - sales_prior_period variance_prior_period,
  sales_same_period_prior_year,
  sales - sales_same_period_prior_year variance_same_period_p_year
FROM
 (SELECT cat, subcat, prod, gid_p, gid_t,
    Hierarchical_Time, yr, qtr, mon, day, sales,
    LAG(sales, 1) OVER (PARTITION BY gid_p, cat, subcat, prod,
      gid_t ORDER BY yr, qtr, mon, day)
      sales_prior_period,
    LAG(sales, 1) OVER (PARTITION BY gid_p, cat, subcat, prod,
      gid_t, qtr_num, mon_num, day_num ORDER BY yr)
      sales_same_period_prior_year
  FROM
   (SELECT c.gid, c.cat, c.subcat, c.prod, c.gid_p,
      t.gid_t, t.yr, t.qtr, t.qtr_num, t.mon, t.mon_num,
      t.day, t.day_num, t.Hierarchical_Time, NVL(s_sold,0) sales
```

```
    FROM cube_prod_time c
    PARTITION BY (gid_p, cat, subcat, prod)
    RIGHT OUTER JOIN edge_time t
    ON ( c.gid_t = t.gid_t AND
      c.Hierarchical_Time = t.Hierarchical_Time)
   ) dense_cube_prod_time
 )                        --side by side current and prior year sales
WHERE prod IN (139) AND gid_p=0 AND    --1 product and product level data
  ( (mon IN ('2001-08' ) AND gid_t IN (0, 1)) OR --day and month data
  (qtr IN ('2001-03' ) AND gid_t IN (3)))        --quarter level data
ORDER BY day;
```

PROD	HT	SALES	sales_prior _period	variance _prior _period	sales_same _period_prior _year	variance _same_period _prior_year
139	01-AUG-01_3	0	0	0	0	0
139	02-AUG-01_3	1347.53	0	1347.53	0	1347.53
139	03-AUG-01_3	0	1347.53	-1347.53	42.36	-42.36
139	04-AUG-01_3	57.83	0	57.83	995.75	-937.92
139	05-AUG-01_3	0	57.83	-57.83	0	0
139	06-AUG-01_3	0	0	0	0	0
139	07-AUG-01_3	134.81	0	134.81	880.27	-745.46
139	08-AUG-01_3	1289.89	134.81	1155.08	0	1289.89
139	09-AUG-01_3	0	1289.89	-1289.89	0	0
139	10-AUG-01_3	0	0	0	0	0
139	11-AUG-01_3	0	0	0	0	0
139	12-AUG-01_3	0	0	0	0	0
139	13-AUG-01_3	0	0	0	0	0
139	14-AUG-01_3	0	0	0	0	0
139	15-AUG-01_3	38.49	0	38.49	1104.55	-1066.06
139	16-AUG-01_3	0	38.49	-38.49	0	0
139	17-AUG-01_3	77.17	0	77.17	1052.03	-974.86
139	18-AUG-01_3	2467.54	77.17	2390.37	0	2467.54
139	19-AUG-01_3	0	2467.54	-2467.54	127.08	-127.08
139	20-AUG-01_3	0	0	0	0	0
139	21-AUG-01_3	0	0	0	0	0
139	22-AUG-01_3	0	0	0	0	0
139	23-AUG-01_3	1371.43	0	1371.43	0	1371.43
139	24-AUG-01_3	153.96	1371.43	-1217.47	2091.3	-1937.34
139	25-AUG-01_3	0	153.96	-153.96	0	0
139	26-AUG-01_3	0	0	0	0	0
139	27-AUG-01_3	1235.48	0	1235.48	0	1235.48
139	28-AUG-01_3	173.3	1235.48	-1062.18	2075.64	-1902.34
139	29-AUG-01_3	0	173.3	-173.3	0	0

```
139  30-AUG-01_3          0          0          0          0          0
139  31-AUG-01_3          0          0          0          0          0
139  2001-08_2       8347.43    7213.21    1134.22    8368.98     -21.55
139  2001-03_1       24356.8   28862.14   -4505.34   24168.99     187.81
```

The first LAG function (sales_prior_period) partitions the data on gid_p, cat, subcat, prod, gid_t and orders the rows on all the time dimension columns. It gets the sales value of the prior period by passing an offset of 1. The second LAG function (sales_same_period_prior_year) partitions the data on additional columns qtr_num, mon_num, and day_num and orders it on yr so that, with an offset of 1, it can compute the year ago sales for the same period. The outermost SELECT clause computes the variances.

Creating a Custom Member in a Dimension: Example

In many OLAP tasks, it is helpful to define custom members in a dimension. For instance, you might define a specialized time period for analyses. You can use a partitioned outer join to temporarily add a member to a dimension. Note that the new SQL MODEL clause is suitable for creating more complex scenarios involving new members in dimensions. See Chapter 22, "SQL for Modeling" for more information on this topic.

As an example of a task, what if we want to define a new member for our time dimension? We want to create a 13th member of the Month level in our time dimension. This 13th month is defined as the summation of the sales for each product in the first month of each quarter of year 2001.

The solution has two steps. Note that we will build this solution using the views and tables created in the prior example. Two steps are required. First, create a view with the new member added to the appropriate dimension. The view uses a UNION ALL operation to add the new member. To query using the custom member, use a CASE expression and a partitioned outer join.

Our new member for the time dimension is created with the following view:

```
CREATE OR REPLACE VIEW time_c AS
(SELECT * FROM edge_time
UNION ALL
SELECT '2001-13_2', 2001, 5, '2001-05', 13, '2001-13', null, null,
8 -- <gid_of_mon>
FROM DUAL);
```

In this statement, the view time_c is defined by performing a UNION ALL of the edge_time view (defined in the prior example) and the user-defined 13th month.

The gid_t value of 8 was chosen to differentiate the custom member from the standard members. The UNION ALL specifies the attributes for a 13th month member by doing a SELECT from the DUAL table. Note that the grouping id, column gid_t, is set to 8, and the quarter number is set to 5.

Then, the second step is to use an inline view of the query to perform a partitioned outer join of cube_prod_time with time_c. This step creates sales data for the 13th month at each level of product aggregation. In the main query, the analytic function SUM is used with a CASE expression to compute the 13th month, which is defined as the summation of the first month's sales of each quarter.

```
SELECT * FROM
  (SELECT SUBSTR(cat,1,12) cat, SUBSTR(subcat,1,12) subcat,
     prod, mon, mon_num,
     SUM(CASE WHEN mon_num IN (1, 4, 7, 10)
           THEN s_sold
           ELSE NULL
         END)
       OVER (PARTITION BY gid_p, prod, subcat, cat, yr) sales_month_13
   FROM
     (SELECT c.gid, c.prod, c.subcat, c.cat, gid_p,
        t.gid_t, t.day, t.mon, t.mon_num,
        t.qtr, t.yr, NVL(s_sold,0) s_sold
      FROM cube_prod_time c
      PARTITION BY (gid_p, prod, subcat, cat)
      RIGHT OUTER JOIN time_c t
      ON (c.gid_t = t.gid_t AND
        c.Hierarchical_Time = t.Hierarchical_Time)
     )
  )
WHERE mon_num=13;

CAT          SUBCAT             PROD MON         MON_NUM SALES_MONTH_13
------------ ------------ ---------- -------- ---------- --------------
Electronics  Game Console         16 2001-13          13      762334.34
Electronics  Y Box Games         139 2001-13          13       75650.22
Electronics  Game Console             2001-13          13      762334.34
Electronics  Y Box Games              2001-13          13       75650.22
Electronics                           2001-13          13      837984.56
                                      2001-13          13      837984.56
```

The SUM function uses a CASE to limit the data to months 1, 4, 7, and 10 within each year. Due to the tiny data set, with just 2 products, the rollup values of the results are necessarily repetitions of lower level aggregations. For more realistic set of

rollup values, you can include more products from the Game Console and Y Box Games subcategories in the underlying materialized view.

22

SQL for Modeling

This chapter discusses using SQL modeling, and includes:

- Overview of SQL Modeling
- Basic Topics in SQL Modeling
- Advanced Topics in SQL Modeling
- Performance Considerations with SQL Modeling
- Examples of SQL Modeling

Overview of SQL Modeling

The MODEL clause brings a new level of power and flexibility to SQL calculations. With the MODEL clause, you can create a multidimensional array from query results and then apply formulas (called rules) to this array to calculate new values. The rules can range from basic arithmetic to simultaneous equations using recursion. For some applications, the MODEL clause can replace PC-based spreadsheets. Models in SQL leverage Oracle Database's strengths in scalability, manageability, collaboration, and security. The core query engine can work with unlimited quantities of data. By defining and executing models within the database, users avoid transferring large data sets to and from separate modeling environments. Models can be shared easily across workgroups, ensuring that calculations are consistent for all applications. Just as models can be shared, access can also be controlled precisely with Oracle's security features. With its rich functionality, the MODEL clause can enhance all types of applications.

The MODEL clause enables you to create a multidimensional array by mapping the columns of a query into three groups: partitioning, dimension, and measure columns. These elements perform the following tasks:

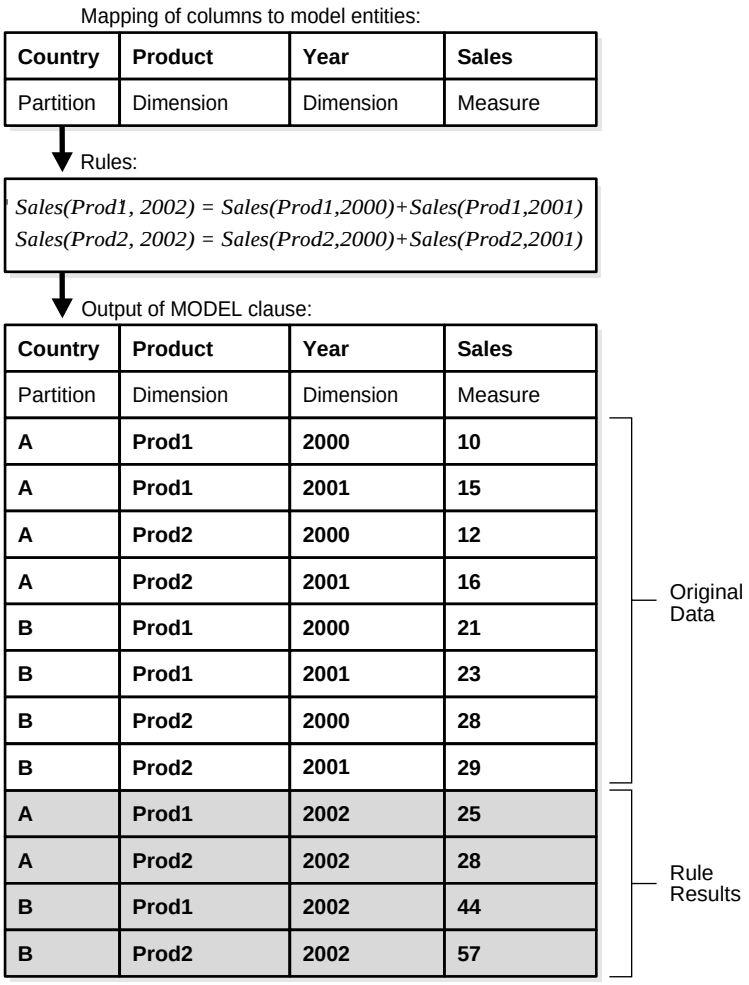
- Partition columns define the logical blocks of the result set in a way similar to the partitions of the analytical functions described in Chapter 21, "SQL for Analysis and Reporting". Rules in the MODEL clause are applied to each partition independent of other partitions. Thus, partitions serve as a boundary point for parallelizing the MODEL computation.
- Dimension columns define the multi-dimensional array and are used to identify cells within a partition. By default, a full combination of dimensions should identify just one cell in a partition. In default mode, they can be considered analogous to the key of a relational table.
- Measures are equivalent to the measures of a fact table in a star schema. They typically contain numeric values such as sales units or cost. Each cell is accessed by specifying its full combination of dimensions. Note that each partition may have a cell that matches a given combination of dimensions.

The MODEL clause enables you to specify rules to manipulate the measure values of the cells in the multi-dimensional array defined by partition and dimension columns. Rules access and update measure column values by specifying dimension values symbolically. Such symbolic references used in rules result in a highly readable model. Rules are concise and flexible, and can use wild cards and looping constructs for maximum expressiveness. Oracle evaluates the rules in an efficient way, parallelizes the model computation whenever possible, and provides a seamless integration of the MODEL clause with other SQL clauses. The MODEL clause,

thus, is a scalable and manageable way of computing business models in the database.

Figure 22–1 offers a conceptual overview of the modeling feature of SQL. The figure has three parts. The top segment shows the concept of dividing a typical table into partition, dimension, and measure columns. The middle segment shows two rules that calculate the value of `Prod1` and `Prod2` for the year 2002. Finally, the third part shows the output of a query that applies the rules to such a table with hypothetical data. The unshaded output is the original data as it is retrieved from the database, while the shaded output shows the rows calculated by the rules. Note that results in partition A are calculated independently from results of partition B.

Figure 22–1 Model Elements



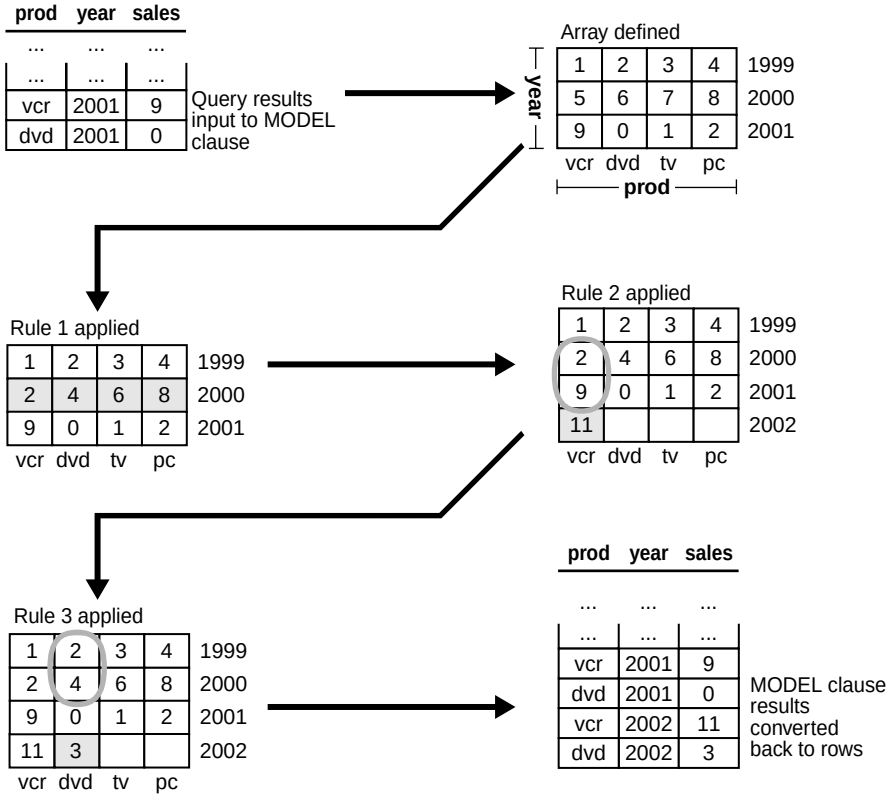
How Data is Processed in a SQL Model

Figure 22–2 shows the flow of processing within a simple `MODEL` clause. In this case, we will follow data through a `MODEL` clause that includes three rules. One of the rules updates an existing value, while the other two create new values for a forecast. The figure shows that the rows of data retrieved by a query are fed into the `MODEL`

clause and rearranged into an array. Once the array is defined, rules are applied one by one to the data. Finally, the data, including both its updated values and newly created values, is rearranged into row form and presented as the results of the query.

Figure 22–2 Model Flow Processing

```
MODEL
DIMENSION BY (prod, year)
MEASURES (sales s)
RULES UPSERT
(s[ANY, 2000]=s[CV(prod), CV(year-1]*2],   --Rule 1
 s[vcr, 2002]=s[vcr, 2001]+s[vcr, 2000],   --Rule 2
 s[dvd, 2002]=AVG(s)[CV(prod), year<2001]) --Rule 3
```



Why Use SQL Modeling?

Oracle modeling enables you to perform sophisticated calculations on your data. A typical case is when you want to apply business rules to data and then generate reports. Because Oracle Database integrates modeling calculations into the database, performance and manageability are enhanced significantly. Consider the following query:

```
SELECT SUBSTR(country, 1, 20) country,
      SUBSTR(product, 1, 15) product, year, sales
FROM sales_view
WHERE country IN ('Italy', 'Japan')
MODEL
  PARTITION BY (country) DIMENSION BY (product, year)
  MEASURES (sales sales)
  RULES
  (sales['Bounce', 2002] = sales['Bounce', 2001] + sales['Bounce', 2000],
   sales['Y Box', 2002] = sales['Y Box', 2001],
   sales['All_Products', 2002] = sales['Bounce', 2002] + sales['Y Box', 2002])
ORDER BY country, product, year;
```

This query partitions the data in `sales_view` (which is illustrated in "Base Schema" on page 22-11) on country so that the model computation, as defined by the three rules, is performed on each country. This model calculates the sales of Bounce in 2002 as the sum of its sales in 2000 and 2001, and sets the sales for Y Box in 2002 to the same value as they were in 2001. Also, it introduces a new product category All_Products (`sales_view` does not have the product All_Products) for year 2002 to be the sum of sales of Bounce and Y Box for that year. The output of this query is as follows, where bold text indicates new values:

```
COUNTRY              PRODUCT               YEAR      SALES
-------------------- --------------- ---------- ----------
Italy                Bounce                1999    2474.78
Italy                Bounce                2000    4333.69
Italy                Bounce                2001     4846.3
Italy                Bounce                2002    9179.99
...
Italy                Y Box                 1999   15215.16
Italy                Y Box                 2000   29322.89
Italy                Y Box                 2001   81207.55
Italy                Y Box                 2002   81207.55
...
Italy                All_Products          2002   90387.54
...
Japan                Bounce                1999     2961.3
```

```
Japan              Bounce              2000    5133.53
Japan              Bounce              2001     6303.6
Japan              Bounce              2002   11437.13
...
Japan              Y Box               1999   22161.91
Japan              Y Box               2000   45690.66
Japan              Y Box               2001   89634.83
Japan              Y Box               2002   89634.83
...
Japan              All_Products        2002  101071.96
...
```

Note that, while the sales values for Bounce and Y Box exist in the input, the values for All_Products are derived.

SQL Modeling Capabilities

Oracle Database provides the following capabilities with the `MODEL` clause:

- Symbolic cell addressing

  Measure columns in individual rows are treated like cells in a multi-dimensional array and can be referenced and updated using symbolic references to dimension values. For example, in a fact table `ft(country, year, sales)`, you can designate `country` and `year` to be dimension columns and `sales` to be the measure and reference sales for a given country and year as `sales[country='Spain', year=1999]`. This gives you the sales value for Spain in 1999. You can also use a shorthand form `sales['Spain', 1999]` to mean the same thing. There are a few semantic differences between these notations, though. See "Cell Referencing" on page 22-15 for further details.

- Symbolic array computation

  You can specify a series of formulas, called rules, to operate on the data. Rules can invoke functions on individual cells or on a set or range of cells. An example involving individual cells is the following:

  ```
  sales[country='Spain',year=2001] = sales['Spain',2000]+ sales['Spain',1999]
  ```

  This sets the sales in Spain for the year 2001 to the sum of sales in Spain for 1999 and 2000. An example involving a range of cells is the following:

  ```
  sales[country='Spain',year=2001] =
     MAX(sales)['Spain',year BETWEEN 1997 AND 2000]
  ```

This sets the sales in Spain for the year 2001 equal to the maximum sales in Spain between 1997 and 2000.

- The UPSERT and UPDATE options

  Using the UPSERT option, which is the default, you can create cell values that do not exist in the input data. If the cell referenced exists in the data, it is updated. Otherwise, it is inserted. The UPDATE option, on the other hand, would not insert any new cells.

  You can specify these options globally, in which case they apply to all rules, or per each rule. If you specify an option at the rule level, it overrides the global option. Consider the following rules:

  ```
  UPDATE sales['Spain', 1999] = 3567.99,
  UPSERT sales['Spain', 2001] = sales['Spain', 2000]+ sales['Spain', 1999]
  ```

  The first rule updates the cell for sales in Spain for 1999. The second rule updates the cell for sales in Spain for 2001 if it exists, otherwise, it creates a new cell.

- Wildcard specification of dimensions

  You can use ANY and IS ANY to specify all values in a dimension. As an example, consider the following statement:

  ```
  sales[ANY, 2001] = sales['Japan', 2000]
  ```

  This rule sets the 2001 sales of all countries equal to the sales value of Japan for the year 2000. All values for the dimension, including nulls, satisfy the ANY specification. You can specify the same in symbolic form using an IS ANY predicate as in the following:

  ```
  sales[country IS ANY, 2001] = sales['Japan', 2000]
  ```

- Accessing dimension values using the CV function

  You can use the CV function on the right side of a rule to access the value of a dimension column of the cell referenced on the left side of a rule. It enables you to combine multiple rules performing similar computation into a single rule, thus resulting in concise specification. For example, you can combine the following rules:

  ```
  sales[country='Spain', year=2002] = 1.2 * sales['Spain', 2001],
  sales[country='Italy', year=2002] = 1.2 * sales['Italy', 2001],
  sales[country='Japan', year=2002] = 1.2 * sales['Japan', 2001]
  ```

They can be combined into one single rule:

```
sales[country IN ('Spain', 'Italy', 'Japan'), year=2002] = 1.2 *
   sales[CV(country), 2001]
```

Observe that the CV function passes the value for the country dimension from the left to the right side of the rule.

- Ordered computation

  For rules updating a set of cells, the result may depend on the ordering of dimension values. You can force a particular order for the dimension values by specifying an ORDER BY in the rule. An example is the following rule:

  ```
  sales[country IS ANY, year BETWEEN 2000 AND 2003] ORDER BY year =
    1.05 * sales[CV(country), CV(year)-1]
  ```

  This ensures that the years are referenced in increasing chronological order.

- Automatic rule ordering

  Rules in the MODEL clause can be automatically ordered based on dependencies among the cells using the AUTOMATIC ORDER keywords. For example, in the following assignments, the last two rules will be processed before the first rule because the first depends on the second and third:

  ```
  RULES AUTOMATIC ORDER
  {sales[c='Spain', y=2001] = sales[c='Spain', y=2000]
    + sales[c='Spain', y=1999]
  sales[c='Spain', y=2000] = 50000,
  sales[c='Spain', y=1999] = 40000}
  ```

- Iterative rule evaluation

  You can specify iterative rule evaluation, in which case the rules are evaluated iteratively until the termination condition is satisfied. Consider the following specification:

  ```
  MODEL DIMENSION BY (x) MEASURES (s)
    RULES ITERATE (4) (s[x=1] = s[x=1]/2)
  ```

  This statement specifies that the formula s[x=1] = s[x=1]/2 evaluation be repeated four times. The number of iterations is specified in the ITERATE option of the MODEL clause. It is also possible to specify a termination condition by using an UNTIL clause.

Iterative rule evaluation is an important tool for modeling recursive relationships between entities in a business application. For example, a loan amount might depend on the interest rate where the interest rate in turn depends on the amount of the loan.

- Reference models

  Rules can reference cells from different multi-dimensional arrays. All but one of the multi-dimensional arrays used in the model are read-only and are called reference models. Rules can update or insert cells in only one multi-dimensional array, which is called the main model. The use of reference models enables you to relate models with different dimensionality. For example, assume that, in addition to the fact table `ft(country, year, sales)`, you have a table with currency conversion ratios `cr(country, ratio)` with `country` as the dimension column and `ratio` as the measure. Each row in this table gives the conversion ratio of that country's currency to that of US dollar. These two tables could be used in rules such as the following:

  ```
  dollar_sales['Spain',2001] = sales['Spain',2000] * ratio['Spain']
  ```

- Scalable computation

  You can partition data and evaluate rules within each partition independent of other partitions. This enables parallelization of model computation based on partitions. For example, consider the following model:

  ```
  MODEL PARTITION BY (c) DIMENSION BY (y) MEASURES (s)
    (sales[y=2001] = AVG(s)[y BETWEEN 1990 AND 2000]
  ```

  The data is partitioned by country and, within each partition, you can compute the sales in 2001 to be the average of sales in the years between 1990 and 2000. Partitions can be processed in parallel and this results in a scalable execution of the model.

Basic Topics in SQL Modeling

This section introduces some of the basic ideas and uses for models, and includes:

- Base Schema
- MODEL Clause Syntax
- Keywords in SQL Modeling
- Rules and Restrictions when Using SQL for Modeling

- Cell Referencing
- Differences Between Update and Upsert

Base Schema

This chapter's examples are based on the following view `sales_view`, which is derived from the `sh` sample schema.

```
CREATE VIEW sales_view AS
SELECT country_name country, prod_name product, calendar_year year,
  SUM(amount_sold) sales, COUNT(amount_sold) cnt,
  MAX(calendar_year) KEEP (DENSE_RANK FIRST ORDER BY SUM(amount_sold) DESC)
  OVER (PARTITION BY country_name, prod_name) best_year,
  MAX(calendar_year) KEEP (DENSE_RANK LAST ORDER BY SUM(amount_sold) DESC)
  OVER (PARTITION BY country_name, prod_name) worst_year
FROM sales, times, customers, countries, products
WHERE sales.time_id = times.time_id AND sales.prod_id = products.prod_id AND
  sales.cust_id =customers.cust_id AND customers.country_id=countries.country_id
GROUP BY country_name, prod_name, calendar_year;
```

This query computes `SUM` and `COUNT` aggregates on the sales data grouped by country, product, and year. It will report for each product sold in a country, the year when the sales were the highest for that product in that country. This is called the `best_year` of the product. Similarly, `worst_year` gives the year when the sales were the lowest.

MODEL Clause Syntax

The `MODEL` clause enables you to define multi-dimensional calculations on the data in the SQL query block. In multi-dimensional applications, a fact table consists of columns that uniquely identify a row with the rest serving as dependent measures or attributes. The `MODEL` clause lets you specify the `PARTITION`, `DIMENSION`, and `MEASURE` columns that define the multi-dimensional array, the rules that operate on this multi-dimensional array, and the processing options.

The `MODEL` clause contains a list of updates representing array computation within a partition and is a part of a SQL query block. Its structure is as follows:

```
MODEL
[<global reference options>]
[<reference models>]
[MAIN <main-name>]
  [PARTITION BY (<cols>)]
  DIMENSION BY (<cols>)
```

```
  MEASURES (<cols>)
  [<reference options>]
  [RULES]  <rule options>
  (<rule>, <rule>,.., <rule>)

  <global reference options> ::= <reference options> <ret-opt>
   <ret-opt> ::= RETURN {ALL|UPDATED} ROWS
  <reference options> ::=
  [IGNORE NAV | [KEEP NAV]
  [UNIQUE DIMENSION | UNIQUE SINGLE REFERENCE]
  <rule options> ::=
  [UPSERT | UPDATE]
  [AUTOMATIC ORDER | SEQUENTIAL ORDER]
  [ITERATE (<number>)  [UNTIL <condition>]]
  <reference models> ::= REFERENCE ON <ref-name> ON (<query>)
  DIMENSION BY (<cols>) MEASURES (<cols>) <reference options>
```

Each rule represents an assignment. Its left side references a cell or a set of cells and the right side can contain expressions involving constants, host variables, individual cells or aggregates over ranges of cells. For example, consider Example 22–1.

Example 22–1 Simple Query with the MODEL Clause

```
SELECT SUBSTR(country,1,20) country, SUBSTR(product,1,15) product, year, sales
FROM sales_view
WHERE country in ('Italy', 'Japan')
MODEL
  RETURN UPDATED ROWS
  MAIN simple_model
  PARTITION BY (country)
  DIMENSION BY (product, year)
  MEASURES (sales)
  RULES
   (sales['Bounce', 2001] = 1000,
    sales['Bounce', 2002] = sales['Bounce', 2001] + sales['Bounce', 2000],
    sales['Y Box', 2002] = sales['Y Box', 2001])
ORDER BY country, product, year;
```

This query defines model computation on the rows from `sales_view` for the countries Italy and Japan. This model has been given the name `simple_model`. It partitions the data on country and defines, within each partition, a two-dimensional array on product and year. Each cell in this array holds the measure value sales. The first rule of this model sets the sales of Bounce in year 2001 to 1000. The next two

rules define that the sales of Bounce in 2002 are the sum of its sales in years 2001 and 2000, and the sales of Y Box in 2002 are same as that of the previous year 2001.

Specifying RETURN UPDATED ROWS makes the preceding query return only those rows that are updated or inserted by the model computation. By default or if you use RETURN ALL ROWS, you would get all rows not just the ones updated or inserted by the MODEL clause. The query produces the following output:

```
COUNTRY              PRODUCT              YEAR      SALES
-------------------- -------------- ---------- ----------
Italy                Bounce               2001       1000
Italy                Bounce               2002    5333.69
Italy                Y Box                2002   81207.55
Japan                Bounce               2001       1000
Japan                Bounce               2002    6133.53
Japan                Y Box                2002   89634.83
```

Note that the MODEL clause does not update or insert rows into database tables. The following query illustrates this be showing that sales_view has not been altered:

```
SELECT SUBSTR(country,1,20) country, SUBSTR(product,1,15) product, year, sales
FROM sales_view
WHERE country IN ('Italy', 'Japan');

COUNTRY              PRODUCT              YEAR      SALES
-------------------- -------------- ---------- ----------
Italy                Bounce               1999    2474.78
Italy                Bounce               2000    4333.69
Italy                Bounce               2001     4846.3
...
```

Observe that the update of the sales value for Bounce in the 2001 done by this MODEL clause is not reflected in the database. If you want to update or insert rows in the database tables, you should use the INSERT, UPDATE, or MERGE statements.

In the preceding example, columns are specified in the PARTITION BY, DIMENSION BY, and MEASURES list. You can also specify constants, host variables, single-row functions, aggregate functions, analytical functions, or expressions involving them as partition and dimension keys and measures. However, you need to alias them in PARTITION BY, DIMENSION BY, and MEASURES lists. You need to use aliases to refer these expressions in the rules, SELECT list, and the query ORDER BY. The following example shows how to use expressions and aliases:

```
SELECT country, p product, year, sales, profits
FROM sales_view
```

```
WHERE country IN ('Italy', 'Japan')
MODEL
  RETURN UPDATED ROWS
  PARTITION BY (SUBSTR(country,1,20) AS country)
  DIMENSION BY (product AS p, year)
  MEASURES (sales, 0 AS profits)
  RULES
   (profits['Bounce', 2001] = sales['Bounce', 2001] * 0.25,
    sales['Bounce', 2002] = sales['Bounce', 2001] + sales['Bounce', 2000],
    profits['Bounce', 2002] = sales['Bounce', 2002] * 0.35)
ORDER BY country, year;

COUNTRY   PRODUCT    YEAR   SALES         PROFITS
-------   ---------  ----   --------     --------
Italy     Bounce     2001     4846.3     1211.575
Italy     Bounce     2002    9179.99    3212.9965
Japan     Bounce     2001     6303.6       1575.9
Japan     Bounce     2002   11437.13    4002.9955
```

See *Oracle Database SQL Reference* for more information regarding `MODEL` clause syntax.

Keywords in SQL Modeling

This section defines keywords used in SQL modeling.

Assigning Values and Null Handling

- `UPDATE`

  This updates existing cell values. If the cell values do not exist, no updates are done.

- `UPSERT`

  This updates existing cell values. If the cell values do not exist, they are inserted.

- `IGNORE NAV`

  For numeric cells, this treats nulls and absent values as 0. This means that a cell not supplied to `MODEL` by the query result set will be treated as a zero for the calculation. This can be used at a global level for all measures in a model.

- `KEEP NAV`

This keeps null and absent cell values unchanged. It is useful for making exceptions when IGNORE NAV is specified at the global level. This is the default, and can be omitted.

Calculation Definition

- MEASURES

  The set of values that are modified or created by the model.

- RULES

  The rules that assign values to measures.

- AUTOMATIC ORDER

  This causes all rules to be evaluated in an order based on their dependencies.

- SEQUENTIAL ORDER

  This causes rules to be evaluated in the order they are written. This is the default.

- UNIQUE DIMENSION

  This is the default, and it means that the PARTITION BY and DIMENSION BY columns in the MODEL clause must uniquely identify each and every cell in the model. This uniqueness is explicitly verified at run time when necessary, in which case it causes some overhead.

- UNIQUE SINGLE REFERENCE

  The PARTITION BY and DIMENSION BY clauses uniquely identify single point references on the right-hand side of the rules. This may reduce processing time by avoiding explicit checks for uniqueness at run time.

- RETURN [ALL|UPDATED] ROWS

  This enables you to specify whether to return all rows selected or only those rows updated by the rules. The default is ALL, while the alternative is UPDATED ROWS.

Cell Referencing

In the MODEL clause, a relation can be viewed as a multi-dimensional array of cells. A cell of this multi-dimensional array contains the measure values and is indexed using DIMENSION BY keys, within each partition defined by the PARTITION BY keys. For example, consider the following:

```
SELECT country, product, year, sales, best_year, best_year
FROM sales_view
MODEL
  PARTITION BY (country) DIMENSION BY (product, year)
MEASURES (sales, best_year)
(<rules> ..)
ORDER BY country, product, year;
```

This partitions the data by country and defines within each partition, a two-dimensional array on product and year. The cells of this array contain two measures: `sales` and `best_year`.

Accessing the measure value of a cell by specifying the `DIMENSION BY` keys constitutes a cell reference. An example of a cell reference is `sales[product= 'Bounce', year=2000]`.

Here, we are accessing the sales value of a cell referenced by product Bounce and the year 2000. In a cell reference, you can specify `DIMENSION BY` keys either symbolically as in the preceding cell reference or positionally as in `sales['Bounce', 2000]`.

Symbolic Dimension References

A symbolic dimension reference (or symbolic reference) is one in which `DIMENSION BY` key values are specified with a boolean expression. For example, the cell reference `sales[year >= 2001]` has a symbolic reference on the `DIMENSION BY` key year and specifies all cells whose year value is greater than or equal to 2001. An example of symbolic references on product and year dimensions is `sales[product = 'Bounce', year >= 2001]`. Rules that use symbolic references cannot insert new cells.

Positional Dimension References

A positional dimension reference (or positional reference, in short) is a constant or an expression involving constants specified for a dimension. For example, the cell reference `sales['Bounce']` has a positional reference on the product dimension and accesses sales value for the product Bounce. The constants (or expressions involving constants) in a cell reference are matched positionally with `DIMENSION BY` keys. The following example shows the usage of positional references on dimensions:

```
sales['Bounce', 2001]
```

Only rules with positional references can insert new cells. Assuming DIMENSION BY keys to be product and year in that order, it accesses the sales value for Bounce and 2001.

Based on how they are specified, cell references are either single cell or multi-cell reference.

Single Cell References on the Right Side

A cell reference in which a single value of interest is specified for each dimension either symbolically or positionally is called a single cell reference. This references a single cell in the default mode in which dimension keys are unique within a partition. For example, considering the following cell reference:

```
sales['Bounce', year=2001]
```

This is a single cell reference in which a single value is specified for the first dimension positionally and a single value for second dimension (year) is specified symbolically.

Multi-Cell References

Cell references that are not single cell references are called multi-cell references. Examples of multi-cell reference are:

```
sales[year>=2001],
sales[product='Bounce', year < 2001], AND
sales[product LIKE '%Finding Fido%', year IN (1999, 2000, 2001)]
```

Rules

Model computation is expressed in rules that manipulate the cells of the multi-dimensional array defined by PARTITION BY, DIMENSION BY, and MEASURES clauses. A rule is an assignment statement whose left side represents a cell or a range of cells and whose right side is an expression involving constants, bind variables, individual cells or an aggregate function on a range of cells. Rules can use wild cards and looping constructs for maximum expressiveness. An example of a rule is the following:

```
sales['Bounce', 2003] = 1.2 * sales['Bounce', 2002]
```

This rule says that, for the product Bounce, the sales for 2003 are 20% more than that of 2002.

Note that this rule has single cell references on both left and right side and is relatively simple. Complex rules can be written with multi-cell references, aggregates, and nested cell references. These are discussed in the following sections.

Single Cell References

This type of rule involves single cell reference on the left side with constants and single cell references on the right side. Some examples are the following:

```
sales[product='Finding Fido', year=2003] = 100000
sales['Bounce', 2003] = 1.2 * sales['Bounce', 2002]
sales[product='Finding Fido', year=2004] = 0.8 * sales['Standard Mouse Pad',
  year=2003] + sales['Finding Fido', 2003]
```

Multi-Cell References on the Right Side

Multi-cell references can be used on the right side of rules, in which case an aggregate function needs to be applied on them to convert them to a single value. All existing aggregate functions including OLAP aggregates (inverse percentile functions, hypothetical rank and distribution functions and so on) and statistical aggregates (correlation, regression slope and so on), and user-defined aggregate functions can be used. For example, the rule to compute the sales of Bounce for 2003 to be 100 more than the maximum sales in the period 1998 to 2002 would be:

```
sales['Bounce', 2003] = 100 + MAX(sales)['Bounce', year BETWEEN 1998 AND 2002]
```

The following example illustrates the usage of inverse percentile function PERCENTILE_DISC. It projects Finding Fido sales for year 2003 to be 30% more than the median sales for products Finding Fido, Standard Mouse Pad, and Boat for all years prior to 2003.

```
sales[product='Finding Fido', year=2003] = 1.3 *
  PERCENTILE_DISC(0.5) WITHIN GROUP (ORDER BY sales) [product IN ('Finding
  Fido','Standard Mouse Pad','Boat'), year < 2003]
```

Consider the following model:

```
SELECT country, product, year, sales
FROM sales_view
WHERE country in ('Poland', 'France')
MODEL
  PARTITION BY (country)
  DIMENSION BY (product, year)
  MEASURES (sales sales, year y)
  RULES UPSERT
  (sales['Finding Fido', 2003] =
```

```
     REGR_SLOPE(sales, y)['Finding Fido', year < 2002] *
     sales['Finding Fido', 2002] + sales['Finding Fido', 2002]);
```

This shows the usage of regression slope REGR_SLOPE function in rules. This function computes the slope of the change of a measure with respect to a dimension of the measure. In the preceding example, it gives the slope of the changes in the sales value with respect to year. This model projects Finding Fido sales for 2003 to be the sales in 2002 scaled by the growth (or slope) in sales for years less than 2002.

Aggregate functions can appear only on the right side of rules. Arguments to the aggregate function can be constants, bind variables, measures of the MODEL clause, or expressions involving them. For example, the rule computes the sales of Bounce for 2003 to be the weighted average of its sales for years from 1998 to 2002 would be:

```
sales['Bounce', 2003] =
   AVG(sales * weight)['Bounce', year BETWEEN 1998 AND 2002]
```

Multi-Cell References on the Left Side

Rules can have multi-cell references on the left side as in the following:

```
sales['Standard Mouse Pad', year > 2000] =
   0.2 * sales['Finding Fido', year=2000]
```

This rule accesses a range of cells on the left side (cells for product Standard Mouse Pad and year greater than 2000) and assigns sales measure of each such cell to the value computed by the right side expression. Computation by the preceding rule is described as "sales of Standard Mouse Pad for years after 2000 is 20% of the sales of Finding Fido for year 2000". This computation is simple in that the right side cell references and hence the right side expression are the same for all cells referenced on the left.

```
sales[product='Standard Mouse Pad', year>2000] =
  sales[CV(product), CV(year)] + 0.2 * sales['Finding Fido', 2000]
```

The CV function provides the value of a DIMENSION BY key of the cell currently referenced on the left side. When the left side references the cell Standard Mouse Pad and 2001, the right side expression would be:

```
sales['Standard Mouse Pad', 2001] + 0.2 * sales['Finding Fido', 2000]
```

Similarly, when the left side references the cell Standard Mouse Pad and 2002, the right side expression we would evaluate is:

```
sales['Standard Mouse Pad', 2002] + 0.2 * sales['Finding Fido', 2000]
```

The use of the CV function provides the capability of relative indexing where dimension values of the cell referenced on the left side are used on the right side cell references. The CV function takes a dimension key as its argument. It is also possible to use CV without any argument as in CV() and in which case, positional referencing is implied. CV() may be used outside a cell reference, but when used in this way its argument must contain the name of the dimension desired. You can also write the preceding rule as:

```
sales[product='Standard Mouse Pad', year>2000] =
  sales[CV(), CV()] + 0.2 * sales['Finding Fido', 2000]
```

The first CV() reference corresponds to CV(product) and the latter corresponds to CV(year). The CV function can be used only in right side cell references. Another example of the usage of CV function is the following:

```
sales[product IN ('Finding Fido','Standard Mouse Pad','Bounce'), year
  BETWEEN 2002 AND 2004] = 2 * sales[CV(product), CV(year)-10]
```

This rule says that, for products Finding Fido, Standard Mouse Pad, and Bounce, the sales for years between 2002 and 2004 will be twice of what their sales were 10 years ago.

Use of the ANY Wildcard

You can use the wild card ANY in cell references to qualify all dimension values including nulls. ANY maybe be used on both the left and right side of rules. For example, a rule for the computation "sales of all products for 2003 are 10% more than their sales for 2002" would be the following:

```
sales[product IS ANY, 2003] = 1.1 * sales[CV(product), 2002]
```

Using positional references, it can also be written as:

```
sales[ANY, 2003] = 1.1 * sales[CV(), 2002]
```

Nested Cell References

Cell references can be nested. In other words, cell references providing dimension values can be used within a cell reference. An example, assuming best_year is a measure, for nested cell reference is given as follows:

```
sales[product='Bounce', year = best_year['Bounce', 2003]]
```

Here, the nested cell reference best_year['Bounce', 2003] provides value for the dimension key year and is used in the symbolic reference for year. Measures

best_year and worst_year give, for each year (y) and product (p) combination, the year for which sales of product p were highest or lowest. The following rule computes the sales of Standard Mouse Pad for 2003 to be the average of Standard Mouse Pad sales for the years in which Finding Fido sales were highest and lowest:

```
sales['Standard Mouse Pad', 2003] = (sales[CV(), best_year['Finding Fido',
  CV(year)]] + sales[CV(), worst_year['Finding Fido', CV(year)]]) / 2
```

Oracle allows only one level of nesting, and only single cell references can be used as nested cell references. Aggregates on multi-cell references cannot be used in nested cell references.

Order of Evaluation of Rules

By default, rules are evaluated in the order they appear in the MODEL clause. You can specify an optional keyword SEQUENTIAL ORDER in the MODEL clause to make such an evaluation order explicit. SQL models with sequential rule order of evaluation are called sequential order models. For example, the following RULES specification makes Oracle evaluate rules in the specified sequence:

```
RULES SEQUENTIAL ORDER
 (sales['Bounce', 2001] =
    sales['Bounce', 2000] + sales['Bounce', 1999],   --Rule R1
  sales['Bounce', 2000] = 50000,                     --Rule R2
  sales['Bounce', 1999] = 40000)                     --Rule R3
```

Alternatively, the option AUTOMATIC ORDER enables Oracle to determine the order of evaluation of rules automatically. Oracle examines the cell references within rules and constructs a dependency graph based on dependencies among rules. If cells referenced on the left side of rule R1 are referenced on the right side of another rule R2, then R2 is considered to depend on R1. In other words, rule R1 should be evaluated before rule R2. If you specify AUTOMATIC ORDER in the preceding example as in:

```
RULES AUTOMATIC ORDER
 (sales['Bounce', 2001] = sales['Bounce', 2000] + sales['Bounce', 1999],
  sales['Bounce', 2000] = 50000,
  sales['Bounce', 1999] = 40000)
```

Rules 2 and 3 are evaluated, in some arbitrary order, before rule 1. This is because rule 1 depends on rules 2 and 3 and hence need to be evaluated after rules 2 and 3. The order of evaluation among second and third rules can be arbitrary as they do not depend on one another. The order of evaluation among rules independent of one another can be arbitrary. A dependency graph is analyzed to come up with the

rule evaluation order. SQL models with automatic order of evaluation, as in the preceding fragment, are called automatic order models.

In an automatic order model, multiple assignments to the same cell are not allowed. In other words, measure of a cell can be assigned only once. Oracle will return an error in such cases as results would be non-deterministic. For example, the following rule specification will generate an error as `sales['Bounce', 2001]` is assigned more than once:

```
RULES AUTOMATIC ORDER
 (sales['Bounce', 2001] = sales['Bounce', 2000] + sales['Bounce', 1999],
  sales['Bounce', 2001] = 50000,
  sales['Bounce', 2001] = 40000)
```

The rules assigning the sales of product Bounce for 2001 do not depend on one another and hence, no particular evaluation order can be fixed among them. This leads to non-deterministic results as the evaluation order is arbitrary - `sales['Bounce', 2001]` can be 40000 or 50000 or sum of Bounce sales for years 1999 and 2000. Oracle prevents this by disallowing multiple assignments when `AUTOMATIC ORDER` is specified. However, multiple assignments are fine in sequential order models. If `SEQUENTIAL ORDER` was specified instead of `AUTOMATIC ORDER` in the preceding example, the result of `sales['Bounce', 2001]` would be 40000.

Differences Between Update and Upsert

Rules in the `MODEL` clause have `UPSERT` semantics by default. If the cell referenced on the left side of a rule exists, then its measure is updated with the value of the right side expression. Otherwise, if a cell reference is positional, a new cell is created (that is, inserted in to the multi-dimensional array) with the measure value equal to the value of the right side expression. If a cell reference is not positional, it will not insert cells. Note that if any column in a cell is symbolic, inserts are not possible. For example, consider the following rule:

```
sales['Bounce', 2003] = sales['Bounce', 2001] + sales ['Bounce', 2002]
```

The cell for product Bounce and year 2003, if it exists, gets updated with the sum of Bounce sales for years 2001 and 2002, otherwise, it gets created. An optional `UPSERT` keyword can be specified in the `MODEL` clause to make this upsert semantic explicit.

Alternatively, the `UPDATE` option forces strict update mode. In this mode, the rule is ignored if the cell it references on the left side does not exist.

You can specify an UPDATE or UPSERT option at the global level in the RULES clause in which case all rules operate in the respective mode. These options can be specified at a local level with each rule and in which case, they override the global behavior. For example, in the following specification:

```
RULES UPDATE
(UPDATE s['Bounce',2001] = sales['Bounce',2000] + sales['Bounce',1999],
 UPSERT s['Y Box', 2001] = sales['Y Box', 2000] + sales['Y Box', 1999],
   sales['Mouse Pad', 2001] = sales['Mouse Pad', 2000] +
   sales['Mouse Pad',1999])
```

The UPDATE option is specified at global level so, first and third rules operate in update mode. The second rule operates in upsert mode as an UPSERT keyword is specified with that rule. Note that no option was specified for the third rule and hence it inherits the update behavior from the global option.

Using UPSERT would create a new cell corresponding to the one referenced on the left side of the rule when the cell is missing, and the cell reference contains only positional references qualified by constants.

Assuming we do not have cells for years greater than 2003, consider the following rule:

```
UPSERT sales['Bounce', year = 2004] = 1.1 * sales['Bounce', 2002]
```

This would not create any new cell because of the symbolic reference year = 2004. However, consider the following:

```
UPSERT sales['Bounce', 2004] = 1.1 * sales['Bounce', 2002]
```

This would create a new cell for product Bounce for year 2004. On a related note, new cells will not be created if any of the positional reference is ANY. This is because ANY is predicate that qualifies all dimensional values including NULL. If there is positional reference ANY for a dimension d, then it can be considered as a predicate (`d IS NOT NULL OR d IS NULL`).

Treatment of NULLs and Missing Cells

Applications using models would not only have to deal with non-deterministic values for a cell measure in the form of NULL, but also with non-determinism in the form of missing cells. A cell, referenced by a single cell reference, that is missing in the data is called a missing cell. The MODEL clause provides a default treatment for nulls and missing cells that is consistent with the ANSI SQL standard and also

provides options to treat them in other useful ways according to business logic, for example, to treat nulls as zero for arithmetic operations.

By default, NULL cell measure values are treated the same way as nulls are treated elsewhere in SQL. For example, in the following rule:

```
sales['Bounce', 2001] = sales['Bounce', 1999] + sales['Bounce', 2000]
```

The right side expression would evaluate to NULL if Bounce sales for one of the years 1999 and 2000 is NULL. Similarly, aggregate functions in rules would treat NULL values in the same way as their regular behavior where NULL values are ignored during aggregation.

Missing cells are treated as cells with NULL measure values. For example, in the preceding rule, if the cell for Bounce and 2000 is missing, then it is treated as a NULL value and the right side expression would evaluate to NULL.

Distinguishing Missing Cells from NULLs

The functions PRESENTV and PRESENTNNV enable you to identify missing cells and distinguish them from NULL values. These functions take a single cell reference and two expressions as arguments as in PRESENTV(cell, expr1, expr2). PRESENTV returns the first expression expr1 if the cell cell is existent in the data input to the MODEL clause. Otherwise, it returns the second expression expr2. For example, consider the following:

```
PRESENTV(sales['Bounce', 2000], 1.1*sales['Bounce', 2000], 100)
```

If the cell for product Bounce and year 2000 exists, it returns the corresponding sales multiplied by 1.1, otherwise, it returns 100. Note that if sales for the product Bounce for year 2000 is NULL, the preceding specification would return NULL.

The PRESENTNNV function not only checks for the presence of a cell but also whether it is NULL or not. It returns the first expression expr1 if the cell exists and is not NULL, otherwise, it returns the second expression expr2. For example, consider the following:

```
PRESENTNNV(sales['Bounce', 2000], 1.1*sales['Bounce', 2000], 100)
```

This would return 1.1*sales['Bounce', 2000] if sales['Bounce', 2000] exists and is not NULL. Otherwise, it returns 100.

Applications can use the new IS PRESENT predicate in their model to check the presence of a cell in an explicit fashion.This predicate returns TRUE if cell exists and FALSE otherwise. The preceding example using PRESENTNNV can be written using IS PRESENT as:

```
CASE WHEN sales['Bounce', 2000] IS PRESENT AND sales['Bounce', 2000] IS NOT NULL
THEN
1.1 * sales['Bounce', 2000]
ELSE
100
END
```

The IS PRESENT predicate, like the PRESENTV and PRESENTNNV functions, checks for cell existence in the input data, that is, the data as existed before the execution of the MODEL clause. This enables you to initialize multiple measures of a cell newly inserted by an UPSERT rule. For example, if you want to initialize sales and profit values of a cell, if it does not exist in the data, for product Bounce and year 2003 to 1000 and 500 respectively, you can do so by the following:

```
RULES
 (UPSERT sales['Bounce', 2003] =
  PRESENTV(sales['Bounce', 2003], sales['Bounce', 2003], 1000),
  UPSERT profit['Bounce', 2003] =
  PRESENTV(profit['Bounce', 2003], profit['Bounce', 2003], 500))
```

The PRESENTV functions used in this formulation would both return TRUE or FALSE based on the existence of the cell in the input data. If the cell for Bounce and 2003 gets inserted by one of the rules, based on their evaluation order, PRESENTV function in the other rule would still evaluate to FALSE. You can consider this behavior as a preprocessing step to rule evaluation that evaluates and replaces all PRESENTV and PRESENTNNV functions and IS PRESENT predicate by their respective values.

Use Defaults for Missing Cells and NULLs

The MODEL clause, by default, treats missing cells as cells with NULL measure values. An optional KEEP NAV keyword can be specified in the MODEL clause to get this behavior.

If your application wants to default missing cells and nulls to some values, you can do so by using IS PRESENT, IS NULL predicates and PRESENTV, PRESENTNNV functions. But it may become cumbersome if you have lot of single cell references and rules. You can use IGNORE NAV option instead of the default KEEP NAV option to default nulls and missing cells to:

- 0 for numeric data
- Empty string for character / string data
- 01-JAN-2001 for data type data

- NULL for all other data types

Consider the following query:

```
SELECT product, year, sales
FROM sales_view
WHERE country = 'Poland'
MODEL
  DIMENSION BY (product, year) MEASURES (sales sales) IGNORE NAV
  RULES UPSERT
  (sales['Bounce', 2003] = sales['Bounce', 2002] + sales['Bounce', 2001]);
```

In this, the input to the MODEL clause does not have a cell for product Bounce and year 2002. Because of IGNORE NAV option, `sales['Bounce', 2002]` value would default to 0 (as sales is of numeric type) instead of NULL. Thus, `sales['Bounce', 2003]` value would be same as that of `sales['Bounce', 2001]`.

Qualifying NULLs for a Dimension

To qualify NULL values for a dimension, you must use one of the following:

- Positional reference using wild card ANY as in `sales[ANY]`.
- Symbolic reference using the IS ANY predicate as in `sales[product is ANY]`.
- Positional reference of NULL as in `sales[NULL]`.
- Symbolic reference using IS NULL predicate as in `sales[product IS NULL]`.

Note that symbolic reference `sales[product = NULL]` would not qualify nulls in product dimension. This behavior is in conformance with the handling of the predicate `"product= NULL"` by SQL.

Reference Models

In addition to the multi-dimensional array on which rules operate, which is called the main model, one or more read-only multi-dimensional arrays, called reference models, can be created and referenced in the MODEL clause to act as look-up tables. Like the main model, a reference model is defined over a query block and has DIMENSION BY and MEASURES clauses to indicate its dimensions and measures respectively. A reference model is created by the following subclause:

```
REFERENCE model_name ON (query) DIMENSION BY (cols) MEASURES (cols)
   [reference options]
```

Like the main model, a multi-dimensional array for the reference model is built before evaluating the rules. But, unlike the main model, reference models are read-only in that their cells cannot be updated and no new cells can be inserted after they are built. Thus, the rules in the main model can access cells of a reference model, but they cannot update or insert new cells into the reference model. References to the cells of a reference model can only appear on the right side of rules. You can view reference models as look-up tables on which the rules of the main model perform look-ups to obtain cell values. The following is an example using a currency conversion table as a reference model:

```
CREATE TABLE dollar_conv_tbl(country VARCHAR2(30), exchange_rate NUMBER);
INSERT INTO dollar_conv_tbl VALUES('Poland', 0.25);
INSERT INTO dollar_conv_tbl VALUES('France', 0.14);
...
```

Now, to convert the projected sales of Poland and France for 2003 to the US dollar, you can use the dollar conversion table as a reference model as in the following:

```
SELECT country, year, sales, dollar_sales
FROM sales_view
GROUP BY country, year
MODEL
  REFERENCE conv_ref ON (SELECT country, exchange_rate FROM dollar_conv_tbl)
  DIMENSION BY (country) MEASURES (exchange_rate) IGNORE NAV
MAIN conversion
DIMENSION BY (country, year)
   MEASURES (SUM(sales) sales, SUM(sales) dollar_sales) IGNORE NAV
RULES
(dollar_sales['France', 2003] = sales[CV(country), 2002] * 1.02 *
   conv_ref.exchange_rate['France'],
   dollar_sales['Poland', 2003] =
      sales['Poland', 2002] * 1.05 * exchange_rate['Poland']);
```

Observe in this example that:

- A one dimensional reference model named `conv_ref` is created on rows from the table `dollar_conv_tbl` and that its measure `exchange_rate` has been referenced in the rules of the main model.
- The main model (called `conversion`) has two dimensions, country and year, whereas the reference model `conv_ref` has one dimension, country.
- Different styles of accessing the `exchange_rate` measure of the reference model. For France, it is rather explicit with `model_name.measure_name`

notation conv_ref.exchange_rate, whereas for Poland, it is a simple measure_name reference exchange_rate. The former notation needs to be used to resolve any ambiguities in column names across main and reference models.

Growth rates, in this example, are hard coded in the rules. The growth rate for France is 2% and that of Poland is 5%. But they could come from a separate table and you can have a reference model defined on top of that. Assume that you have a growth_rate(country, year, rate) table defined as the following:

```
CREATE TABLE growth_rate_tbl(country VARCHAR2(30),
   year NUMBER, growth_rate NUMBER);
INSERT INTO growth_rate_tbl VALUES('Poland', 2002, 2.5);
INSERT INTO growth_rate_tbl VALUES('Poland', 2003, 5);
...
INSERT INTO growth_rate_tbl VALUES('France', 2002, 3);
INSERT INTO growth_rate_tbl VALUES('France', 2003, 2.5);
```

Then the following query computes the projected sales in dollars for 2003 for all countries:

```
SELECT country, year, sales, dollar_sales
FROM sales_view
GROUP BY country, year
MODEL
  REFERENCE conv_ref ON
          (SELECT country, exchange_rate FROM dollar_conv_tbl)
           DIMENSION BY (country c) MEASURES (exchange_rate) IGNORE NAV
  REFERENCE growth_ref ON
          (SELECT country, year, growth_rate FROM growth_rate_tbl)
           DIMENSION BY (country c, year y) MEASURES (growth_rate) IGNORE NAV
  MAIN projection
   DIMENSION BY (country, year) MEASURES (SUM(sales) sales, 0 dollar_sales)
  IGNORE NAV
  RULES
  (dollar_sales[ANY, 2003] = sales[CV(country), 2002] *
   growth_rate[CV(country), CV(year)] *
   exchange_rate[CV(country)]);
```

This query shows the capability of the MODEL clause in dealing with and relating objects of different dimensionality. Reference model conv_ref has one dimension while the reference model growth_ref and the main model have two dimensions. Dimensions in the single cell references on reference models are specified using the CV function thus relating the cells in main model with the reference model. This

specification, in effect, is performing a relational join between main and reference models.

Reference models also help you convert keys to sequence numbers, perform computations using sequence numbers (for example, where a prior period would be used in a subtraction operation), and then convert sequence numbers back to keys. For example, consider a view that assigns sequence numbers to years:

```
CREATE or REPLACE VIEW year_2_seq (i, year) AS
SELECT ROW_NUMBER() OVER (ORDER BY calendar_year), calendar_year
FROM (SELECT DISTINCT calendar_year FROM TIMES);
```

This view can define two lookup tables: integer-to-year `i2y`, which maps sequence numbers to integers, and year-to-integer `y2i`, which performs the reverse mapping. The references `y2i.i[year]` and `y2i.i[year] - 1` return sequence numbers of the current and previous years respectively and the reference `i2y.y[y2i.i[year]-1]` returns the year key value of the previous year. The following query demonstrates such a usage of reference models:

```
SELECT country, product, year, sales, prior_period
FROM sales_view
MODEL
  REFERENCE y2i ON (SELECT year, i FROM year_2_seq) DIMENSION BY (year y)
   MEASURES (i)
  REFERENCE i2y ON (SELECT year, i FROM year_2_seq) DIMENSION BY (i)
   MEASURES (year y)
  MAIN projection2 PARTITION BY (country)
  DIMENSION BY (product, year)
  MEASURES (sales, CAST(NULL AS NUMBER) prior_period)
(prior_period[ANY, ANY] = sales[CV(product), i2y.y[y2i.i[CV(year)]-1]])
ORDER BY country, product, year;
```

Nesting of reference model cell references is evident in the preceding example. Cell reference on the reference model `y2i` is nested inside the cell reference on `i2y` which, in turn, is nested in the cell reference on the main SQL model. There is no limitation on the levels of nesting you can have on reference model cell references. However, you can only have two levels of nesting on the main SQL model cell references.

Finally, the following are restrictions on the specification and usage of reference models:

- Reference models cannot have a `PARTITION BY` clause.

- The query block on which the reference model is defined cannot be correlated to an outer query.
- Reference models must be named and their names should be unique.
- All references to the cells of a reference model should be single cell references.

Advanced Topics in SQL Modeling

This section discusses more advanced topics in SQL modeling, and includes:

- FOR Loops
- Iterative Models
- Ordered Rules
- Unique Dimensions Versus Unique Single References

FOR Loops

The MODEL clause provides a FOR construct that can be used inside rules to express computations more compactly. It can be used on both the left and right side of a rule. For example, consider the following computation, which estimates the sales of several products for 2004 to be 10% higher than their sales for 2003:

```
RULES UPSERT
(sales['Bounce', 2004] = 1.1 * sales['Bounce', 2003],
 sales['Standard Mouse Pad', 2004] = 1.1 * sales['Standard Mouse Pad', 2003],
...
 sales['Y Box', 2004] = 1.1 * sales['Y Box', 2003])
```

The UPSERT option is used in this computation so that cells for these products and 2004 will be inserted if they are not previously present in the multi-dimensional array. This is rather bulky as you have to have as many rules as there are products. Using the FOR construct, this computation can be represented compactly and with exactly the same semantics as in:

```
RULES UPSERT
(sales[FOR product IN ('Bounce', 'Standard Mouse Pad', ..., 'Y Box'), 2004] =
   1.1 * sales[CV(product), 2003])
```

If you write a specification similar to this, but without the FOR keyword as in the following:

```
RULES UPSERT
```

```
(sales[product IN ('Bounce', 'Standard Mouse Pad', ..., 'Y Box'), 2004] =
   1.1 * sales[CV(product), 2003])
```

You would get UPDATE semantics even though you have specified UPSERT. In other words, existing cells will be updated but no new cells will be created by this specification. This is because of the symbolic multi-cell reference on product that is treated as a predicate. You can view a FOR construct as a macro that generates multiple rules with positional references from a single rule, thus preserving the UPSERT semantics. Conceptually, the following rule:

```
sales[FOR product IN ('Bounce', 'Standard Mouse Pad', ..., 'Y Box'),
      FOR year IN (2004, 2005)] = 1.1 * sales[CV(product),
    CV(year)-1]
```

Can be treated as an ordered collection of the following rules:

```
sales['Bounce', 2004] = 1.1 * sales[CV(product), CV(year)-1],
sales['Bounce', 2005] = 1.1 * sales[CV(product), CV(year)-1],
sales['Standard Mouse Pad', 2004] = 1.1 *
  sales[CV(product), CV(year)-1],
sales['Standard Mouse Pad', 2005] = 1.1 * sales[CV(product),
 CV(year)-1],
...
sales['Y Box', 2004] = 1.1 * sales[CV(product), CV(year)-1],
sales['Y Box', 2005] = 1.1 * sales[CV(product), CV(year)-1]
```

The FOR construct in the preceding examples is of type FOR dimension IN (list of values). Values in the list should either be constants or expressions involving constants. In this example, there are separate FOR constructs on product and year. It is also possible to specify all dimensions using one FOR construct. Consider for example, we want only to estimate sales for Bounce in 2004, Standard Mouse Pad in 2005 and Y Box in 2004 and 2005. This can be formulated as the following:

```
sales[FOR (product, year) IN (('Bounce', 2004), ('Standard Mouse Pad', 2005),
  ('Y Box', 2004), ('Y Box', 2005))] =
     1.1 * sales[CV(product), CV(year)-1]
```

This FOR construct should be of form `FOR (d1, ..., dn) IN ((d1_val1, ..., dn_val1), ..., (d1_valm, ..., dn_valm)]` when there are n dimensions `d1, ..., dn` and `m` values in the list.

In some cases, the list of values for a dimension in FOR can be stored in a table or they can be the result of a subquery. Oracle Database provides a flavor of FOR construct as in FOR dimension in (subquery) to handle these cases. For example,

assume that the products of interest are stored in a table interesting_products, then the following rule estimates their sales in 2004 and 2005:

```
sales[FOR product IN (SELECT product_name FROM interesting_products)
    FOR year IN (2004, 2005)] = 1.1 * sales[CV(product), CV(year)-1]
```

As another example, consider the scenario where you want to introduce a new country, called new_country, with sales that mimic those of Poland. This is accomplished by issuing the following statement:

```
SELECT country, product, year, s
FROM sales_view
MODEL
  DIMENSION BY (country, product, year)
  MEASURES (sales s) IGNORE NAV
  RULES UPSERT
(s[FOR (country, product, year) IN (SELECT DISTINCT 'new_country', product, year
FROM sales_view
WHERE country = 'Poland')] = s['Poland',CV(),CV()])
ORDER BY country, year, product;
```

Note the multi-column IN-list produced by evaluating the subquery in this specification. The subquery used to obtain the IN-list cannot be correlated to outer query blocks and it should return fewer than 10000 rows. Otherwise, Oracle returns an error. Model has a 10,000 rule limit and each combination of values created in a cell reference creates a rule. Therefore, FOR constructs should be designed so that the combination of values they generate does not exceed 10,000.

If the FOR construct is used to densify sparse data on multiple dimensions, it is possible to encounter the 10,000 rule limit. In those cases, densification can be done outside the MODEL clause using a partitioned outer join. See Chapter 21, "SQL for Analysis and Reporting" for further information.

If you know that the values of interest come from a discrete domain, you can use FOR construct FOR dimension FROM value1 TO value2 [INCREMENT | DECREMENT] value3. This specification results in values between value1 and value2 by starting from value1 and incrementing (or decrementing) by value3. The values value1, value2, and value3 should be constants or expressions involving constants. For example, the following rule:

```
sales['Bounce', FOR year FROM 2001 TO 2005 INCREMENT 1] =
  sales['Bounce', year=CV(year)-1] * 1.2
```

This is semantically equivalent to the following rules in order:

```
sales['Bounce', 2001] = sales['Bounce', 2000] * 1.2,
```

```
sales['Bounce', 2002] = sales['Bounce', 2001] * 1.2,
...
sales['Bounce', 2005] = sales['Bounce', 2004] * 1.2
```

This kind of FOR construct can be used for dimensions of numeric, date and datetime datatypes. The increment / decrement expression value3 should be numeric for numeric dimensions and can be numeric or interval for dimensions of date or datetime types. Also, value3 should be positive. Oracle will return an error if you use FOR year FROM 2005 TO 2001 INCREMENT -1. You should use either FOR year FROM 2005 TO 2001 DECREMENT 1 or FOR year FROM 2001 TO 2005 INCREMENT 1. Oracle will also report an error if the domain (or the range) is empty, as in FOR year FROM 2005 TO 2001 INCREMENT 1.

To generate string values, you can use the FOR construct FOR dimension LIKE string FROM value1 TO value2 [INCREMENT | DECREMENT] value3. The string string should contain only one % character. This specification results in string by replacing % with values between value1 and value2 with appropriate increment / decrement value value3. For example, the following rule:

```
sales[FOR product LIKE 'product-%' FROM 1 TO 3 INCREMENT 1, 2003] =
sales[CV(product), 2002] * 1.2
```

This equivalent to the following:

```
sales['product-1', 2003] = sales['product-1', 2002] * 1.2,
sales['product-2', 2003] = sales['product-2', 2002] * 1.2,
sales['product-3', 2003] = sales['product-3', 2002] * 1.2
```

For this kind of FOR construct, value1, value2, and value3 should all be of numeric type.

```
sales['product-1', 2003] = sales['product-1', 2002] * 1.2,
sales['product-2', 2003] = sales['product-2', 2002] * 1.2,
sales['product-3', 2003] = sales['product-3', 2002] * 1.2
```

In SEQUENTIAL ORDER models, rules represented by a FOR construct are evaluated in the order they are generated. On the contrary, rule evaluation order would be dependency based if AUTOMATIC ORDER is specified. For example, the evaluation order for the rules represented by the rule:

```
sales['Bounce', FOR year FROM 2004 TO 2001 DECREMENT 1] =
  1.1 * sales['Bounce', CV(year)-1]
```

For SEQUENTIAL ORDER models would be:

```
sales['Bounce', 2004] = 1.1 * sales['Bounce', 2003],
```

```
sales['Bounce', 2003] = 1.1 * sales['Bounce', 2002],
sales['Bounce', 2002] = 1.1 * sales['Bounce', 2001],
sales['Bounce', 2001] = 1.1 * sales['Bounce', 2000]
```

While for AUTOMATIC ORDER models, it would be:

```
sales['Bounce', 2001] = 1.1 * sales['Bounce', 2000],
sales['Bounce', 2002] = 1.1 * sales['Bounce', 2001],
sales['Bounce', 2003] = 1.1 * sales['Bounce', 2002],
sales['Bounce', 2004] = 1.1 * sales['Bounce', 2003]
```

Iterative Models

Using the ITERATE option of the MODEL clause, you can evaluate rules iteratively for a certain number of times, which you can specify as an argument to the ITERATE clause. ITERATE can be specified only for SEQUENTIAL ORDER models and such models are referred to as iterative models. For example, consider the following:

```
SELECT x, s FROM DUAL
MODEL
  DIMENSION BY (1 AS x) MEASURES (1024 AS s)
  RULES UPDATE ITERATE (4)
(s[1] = s[1]/2);
```

In Oracle, the table DUAL has only one row. Hence this model defines a 1-dimensional array, dimensioned by x with a measure s, with a single element s[1] = 1024. The rule s[1] = s[1]/2 evaluation will be repeated four times. The result of this query will be a single row with values 1 and 64 for columns x and s respectively. The number of iterations arguments for the ITERATE clause should be a positive integer constant. Optionally, you can specify an early termination condition to stop rule evaluation before reaching the maximum iteration. This condition is specified in the UNTIL subclause of ITERATE and is checked at the end of an iteration. So, you will have at least one iteration when ITERATE is specified. The syntax of the ITERATE clause is:

```
ITERATE (number_of_iterations) [ UNTIL (condition) ]
```

Iterative evaluation will stop either after finishing the specified number of iterations or when the termination condition evaluates to TRUE, whichever comes first.

In some cases, you may want the termination condition to be based on the change, across iterations, in value of a cell. Oracle provides a mechanism to specify such conditions in that it enables you to access cell values as they existed before and after the current iteration in the UNTIL condition. Oracle's PREVIOUS function takes a

single cell reference as an argument and returns the measure value of the cell as it existed after the previous iteration. You can also access the current iteration number by using the system variable ITERATION_NUMBER, which starts at value 0 and is incremented after each iteration. By using PREVIOUS and ITERATION_NUMBER, you can construct complex termination conditions.

Consider the following iterative model that specifies iteration over rules till the change in the value of s[1] across successive iterations falls below 1, up to a maximum of 1000 times:

```
SELECT x, s, iterations FROM DUAL
MODEL
  DIMENSION BY (1 AS x) MEASURES (1024 AS s, 0 AS iterations)
  RULES ITERATE (1000) UNTIL ABS(PREVIOUS(s[1]) - s[1]) < 1
 (s[1] = s[1]/2, iterations[1] = ITERATION_NUMBER);
```

The absolute value function (ABS) can be helpful for termination conditions because you may not know if the most recent value is positive or negative. Rules in this model will be iterated over 11 times as after 11th iteration the value of s[1] would be 0.5. This query results in a single row with values 1, 0.5, 10 for x, s and iterations respectively.

You can use the PREVIOUS function only in the UNTIL condition. However, ITERATION_NUMBER can be anywhere in the main model. In the following example, ITERATION_NUMBER is used in cell references:

```
SELECT country, product, year, sales
FROM sales_view
MODEL
  PARTITION BY (country) DIMENSION BY (product, year) MEASURES (sales sales)
  IGNORE NAV
  RULES ITERATE(3)
(sales['Bounce', 2002 + ITERATION_NUMBER] = sales['Bounce', 1999
  + ITERATION_NUMBER]);
```

This statement achieves an array copy of sales of Bounce from cells in the array 1999-2001 to 2002-2005.

Rule Dependency in AUTOMATIC ORDER Models

Oracle Database determines the order of evaluation of rules in an AUTOMATIC ORDER model based on their dependencies. A rule will be evaluated only after the rules it depends on are evaluated. The algorithm chosen to evaluate the rules is based on the dependency graph analysis and whether rules in your model have

circular (or cyclical) dependencies. A cyclic dependency can be of the form "rule A depends on B and rule B depends on A" or of the self-cyclic "rule depending on itself" form. An example of the former is:

```
sales['Bounce', 2002] = 1.5 * sales['Y Box', 2002],
sales['Y Box', 2002] = 100000 / sales['Bounce', 2002
```

An example of the latter is:

```
sales['Bounce', 2002] = 25000 / sales['Bounce', 2002]
```

However, there is no self-cycle in the following rule as different measures are being accessed on the left and right side:

```
projected_sales['Bounce', 2002] = 25000 / sales['Bounce', 2002]
```

When the analysis of an AUTOMATIC ORDER model finds that the rules have no circular dependencies (that is, the dependency graph is acyclic), Oracle Database will evaluate the rules in their dependency order. For example, in the following AUTOMATIC ORDER model:

```
MODEL DIMENSION BY (prod, year) MEASURES (sale sales) IGNORE NAV
  RULES AUTOMATIC ORDER
 (sales['SUV', 2001] = 10000,
  sales['Standard Mouse Pad', 2001] = sales['Finding Fido', 2001]
    * 0.10 + sales['Boat', 2001] * 0.50,
  sales['Boat', 2001] = sales['Finding Fido', 2001]
    * 0.25 + sales['SUV', 2001]* 0.75,
  sales['Finding Fido', 2001] = 20000)
```

Rule 2 depends on rules 3 and 4, while rule 3 depends on rules 1 and 4, and rules 1 and 4 do not depend on any rule. Oracle, in this case, will find that the rule dependencies are acyclic and will evaluate rules in one of the possible evaluation orders (1, 4, 3, 2) or (4, 1, 3, 2). This type of rule evaluation is called an ACYCLIC algorithm.

In some cases, Oracle Database may not be able to ascertain that your model is acyclic even though there is no cyclical dependency among the rules. This can happen if you have complex expressions in your cell references. Oracle Database assumes that the rules are cyclic and employs a CYCLIC algorithm that evaluates the model iteratively based on the rules and data. Iteration will stop as soon as convergence is reached and the results will be returned. Convergence is defined as the state in which further executions of the model will not change values of any of the cell in the model. Convergence is certain to be reached when there are no cyclical dependencies.

If your AUTOMATIC ORDER model has rules with cyclical dependencies, Oracle will employ the earlier mentioned CYCLIC algorithm. Results are produced if convergence can be reached within the number of iterations Oracle is going to try the algorithm. Otherwise, Oracle will report a cycle detection error. You can circumvent this problem by manually ordering the rules and specifying SEQUENTIAL ORDER.

Ordered Rules

An ordered rule is one that has ORDER BY specified on the left side. It accesses cells in the order prescribed by ORDER BY and applies the right side computation. When you have a positional ANY or symbolic references on the left side of a rule but without the ORDER BY clause, Oracle might return an error saying that the rule's results depend on the order in which cells are accessed and hence are non-deterministic. Consider the following SEQUENTIAL ORDER model:

```
SELECT t, s
FROM sales, times
WHERE sales.time_id = times.time_id
GROUP BY calendar_year
MODEL
  DIMENSION BY (calendar_year t) MEASURES (SUM(amount_sold) s)
  RULES SEQUENTIAL ORDER
  (s[ANY] = s[CV(t)-1]);
```

This query attempts to set, for all years t, sales s value for a year to the sales value of the prior year. Unfortunately, the result of this rule depends on the order in which the cells are accessed. If cells are accessed in the ascending order of year, the result would be that of column 3 in Table 22–1. If they are accessed in descending order, the result would be that of column 4.

Table 22–1 Ordered Rules

t	s	If ascending	If descending
1998	1210000982	null	null
1999	1473757581	null	1210000982
2000	2376222384	null	1473757581
2001	1267107764	null	2376222384

If you want the cells to be considered in descending order and get the result given in column 4, you should specify:

```
SELECT t, s
FROM sales, times
WHERE sales.time_id = times.time_id
GROUP BY calendar_year
MODEL
  DIMENSION BY (calendar_year t) MEASURES (SUM(amount_sold) s)
  RULES SEQUENTIAL ORDER
  (s[ANY] ORDER BY t DESC = s[CV(t)-1]);
```

In general, you can use any ORDER BY specification as long as it produces a unique order among cells that qualify the left side cell reference. Expressions in the ORDER BY of a rule can involve constants, measures and dimension keys and you can specify the ordering options [ASC | DESC] [NULLS FIRST | NULLS LAST] to get the order you want.

You can also specify ORDER BY for rules in an AUTOMATIC ORDER model to make Oracle consider cells in a particular order during rule evaluation. Rules are never considered self-cyclic if they have ORDER BY. For example, to make the following AUTOMATIC ORDER model with a self-cyclic formula:

```
MODEL
  DIMENSION BY (calendar_year t) MEASURES (SUM(amount_sold) s)
  RULES AUTOMATIC ORDER
  (s[ANY] = s[CV(t)-1])
```

acyclic, you need to provide the order in which cells need to be accessed for evaluation using ORDER BY. For example, you can say:

```
s[ANY] ORDER BY t = s[CV(t) - 1]
```

Then Oracle will pick an ACYCLIC algorithm (which is certain to produce the result) for formula evaluation.

Unique Dimensions Versus Unique Single References

The MODEL clause, in its default behavior, requires the PARTITION BY and DIMENSION BY keys to uniquely identify each row in the input to the model. Oracle verifies that and returns an error if the data is not unique. Uniqueness of the input rowset on the PARTITION BY and DIMENSION BY keys guarantees that any single cell reference accesses one and only one cell in the model. You can specify an optional UNIQUE DIMENSION keyword in the MODEL clause to make this behavior explicit. For example, the following query:

```
SELECT country, product, sales
FROM sales_view
```

```
WHERE country IN ('France', 'Poland')
MODEL UNIQUE DIMENSION
  PARTITION BY (country) DIMENSION BY (product) MEASURES (sales sales)
  IGNORE NAV RULES UPSERT
(sales['Bounce'] = sales['All Products'] * 0.24);
```

This would return a uniqueness violation error as the rowset input to model is not unique on country and product:

```
ERROR at line 2:
ORA-32638: Non unique addressing in MODEL dimensions
```

However, the following query does not return such an error:

```
SELECT country, product, year, sales
FROM sales_view
WHERE country IN ('Italy', 'Japan')
MODEL UNIQUE DIMENSION
  PARTITION BY (country) DIMENSION BY (product, year) MEASURES (sales sales)
  RULES UPSERT
(sales['Bounce', 2003] = sales['All Products', 2002] * 0.24);
```

Input to the MODEL clause in this case is unique on country, product, and year as shown in:

```
COUNTRY   PRODUCT                          YEAR   SALES
-------   ------------------------------   ----   --------
Italy     1.44MB External 3.5" Diskette   1998    3141.84
Italy     1.44MB External 3.5" Diskette   1999    3086.87
Italy     1.44MB External 3.5" Diskette   2000    3440.37
Italy     1.44MB External 3.5" Diskette   2001     855.23
...
```

If you want to relax this uniqueness checking, you can specify UNIQUE SINGLE REFERENCE keyword. This can save processing time. In this case, the MODEL clause checks the uniqueness of only the single cell references appearing on the right side of rules. So the query that returned the uniqueness violation error would be successful if you specify UNIQUE SINGLE REFERENCE instead of UNIQUE DIMENSION.

Another difference between UNIQUE DIMENSION and UNIQUE SINGLE REFERENCE semantics is the number of cells that can be updated by a rule with a single cell reference on left side. In the case of UNIQUE DIMENSION, such a rule can update at most one row as only one cell would qualify the single cell reference on the left side. This is because the input rowset would be unique on PARTITION BY and

DIMENSION BY keys. With UNIQUE SINGLE REFERENCE, all cells that qualify the left side single cell reference would be updated by the rule.

Rules and Restrictions when Using SQL for Modeling

The following rules and restrictions apply when using the MODEL clause:

- The only columns that can be updated are the columns specified in the MEASURES subclause of the main SQL model. Measures of reference models cannot be updated.
- The MODEL clause is evaluated after all clauses in the query block except SELECT DISTINCT, and ORDER BY clause are evaluated. These clauses and expressions in the SELECT list are evaluated after the MODEL clause.
- If your query has a MODEL clause, then the query's SELECT and ORDER BY lists cannot contain aggregates or analytic functions. If needed, these can be specified in PARTITION BY, DIMENSION BY, and MEASURES lists and need to be aliased. Aliases can then be used in the SELECT or ORDER BY clauses. In the following example, the analytic function RANK is specified and aliased in the MEASURES list of the MODEL clause, and its alias is used in the SELECT list so that the outer query can order resulting rows based on their ranks.

```
SELECT country, product, year, s, RNK
FROM (SELECT country, product, year, s, rnk
      FROM sales_view
      MODEL
        PARTITION BY (country) DIMENSION BY (product, year)
        MEASURES (sales s, year y, RANK() OVER (ORDER BY sales) rnk)
        RULES UPSERT
          (s['Bounce Increase 90-99', 2001] =
             REGR_SLOPE(s, y) ['Bounce', year BETWEEN 1990 AND 2000],
           s['Bounce', 2001] = s['Bounce', 2000] *
          (1+s['Bounce increase 90-99', 2001])))
WHERE product <> 'Bounce Increase 90-99'
ORDER BY country, year, rnk, product;
```

- When there is a multi-cell reference on the right hand side of a rule, you need to apply a function to aggregate the measure values of multiple cells referenced into a single value. You can use any kind of aggregate function for this purpose: regular, OLAP aggregate (inverse percentile, hypothetical rank and distribution), or user-defined aggregate.
- You cannot use analytic functions (functions that use the OVER clause) in rules.

- Only rules with positional single cell references on the left side have UPSERT semantics. All other rules have UPDATE semantics, even when you specify the UPSERT option for them.
- Negative increments are not allowed in FOR loops. Also, no empty FOR loops are allowed. FOR d FROM 2005 TO 2001 INCREMENT -1 is illegal. You should use FOR d FROM 2005 TO 2001 DECREMENT 1 instead. FOR d FROM 2005 TO 2001 INCREMENT 1 is illegal as it designates an empty loop.
- You cannot use nested query expressions (subqueries) in rules except in the FOR construct. For example, it would be illegal to issue the following:

```
SELECT *
FROM sales_view WHERE country = 'Poland'
MODEL DIMENSION BY (product, year)
  MEASURES (sales sales)
  RULES UPSERT
  (sales['Bounce', 2003] = sales['Bounce', 2002] +
   (SELECT SUM(sales) FROM sales_view));
```

  This is because the rule has a subquery on its right side. Instead, you can rewrite the preceding query in the following legal way:

```
SELECT *
FROM sales_view WHERE country = 'Poland'
MODEL DIMENSION BY (product, year)
  MEASURES (sales sales, (SELECT SUM(sales) FROM sales_view) AS grand_total)
  RULES UPSERT
  (sales['Bounce', 2003] =sales['Bounce', 2002] +
   grand_total['Bounce', 2002]);
```

- You can also use subqueries in the FOR construct specified on the left side of a rule. However, they:
  - Cannot be correlated
  - Must return fewer than 10000 rows
  - Cannot be a query defined in the WITH clause
  - Will make the cursor unsharable
- Nested cell references must be single cell references. Aggregates on nested cell references are not supported. So, it would be illegal to say s['Bounce', MAX(best_year)['Bounce', ANY]].

- Only one level of nesting is supported for nested cell references on the main model. So, for example, `s['Bounce', best_year['Bounce', 2001]]` is legal, but `s['Bounce', best_year['Bounce', best_year['Bounce', 2001]]]` is not.
- Nested cell references appearing on the left side of rules in an `AUTOMATIC ORDER` model should not be updated in any rule of the model. This restriction ensures that the rule dependency relationships do not arbitrarily change (and hence cause non-deterministic results) due to updates to reference measures.

  There is no such restriction on nested cell references in a `SEQUENTIAL ORDER` model. Also, this restriction is not applicable on nested references appearing on the right side of rules in both `SEQUENTIAL` or `AUTOMATIC ORDER` models.
- Reference models have the following restrictions:
  - The query defining the reference model cannot be correlated to any outer query. It can, however, be a query with subqueries, views and so on.
  - Reference models cannot have a `PARTITION BY` clause.
  - Reference models cannot be updated.

Performance Considerations with SQL Modeling

The following sections describe topics that affect performance when using the `MODEL` clause:

- Parallel Execution
- Aggregate Computation
- Using EXPLAIN PLAN to Understand Model Queries

Parallel Execution

`MODEL` clause computation is scalable in terms of the number of processors you have. Scalability is achieved by performing the `MODEL` computation in parallel across the partitions defined by the `PARTITION BY` clause. Data is distributed among processing elements (also called parallel query slaves) based on the `PARTITION BY` key values such that all rows with the same values for the `PARTITION BY` keys will go to the same slave. Note that the internal processing of partitions will not create a one-to-one match of logical and internally processed partitions. This way, each slave can finish `MODEL` clause computation independent

of other slaves. The data partitioning can be hash based or range based. Consider the following MODEL clause:

```
MODEL
  PARTITION BY (country) DIMENSION BY (product, time) MEASURES (sales)
  RULES UPDATE
  (sales['Bounce', 2002] = 1.2 * sales['Bounce', 2001],
   sales['Car', 2002] = 0.8 * sales['Car', 2001])
```

Here input data will be partitioned among slaves based on the PARTITION BY key country and this partitioning can be hash or range based. Each slave will evaluate the rules on the data it receives.

Parallelism of the model computation is governed or limited by the way you specify the MODEL clause. If your MODEL clause has no PARTITION BY keys, then the computation cannot be parallelized (with exceptions mentioned in the following). If PARTITION BY keys have very low cardinality, then the degree of parallelism will be limited. In such cases, Oracle identifies the DIMENSION BY keys that can used for partitioning. For example, consider a MODEL clause equivalent to the preceding one, but without PARTITION BY keys as in the following:

```
MODEL
  DIMENSION BY (country, product, time) MEASURES (sales)
  RULES UPDATE
  (sales[ANY, 'Bounce', 2002] = 1.2 * sales[CV(country), 'Bounce', 2001],
   sales[ANY, 'Car', 2002] = 0.8 * sales[CV(country), 'Car', 2001])
```

In this case, Oracle Database will identify that it can use the DIMENSION BY key country for partitioning and uses region as the basis of internal partitioning. It partitions the data among slaves on country and thus effects parallel execution.

Aggregate Computation

The MODEL clause processes aggregates in two different ways: first, the regular fashion in which data in the partition is scanned and aggregated and second, an efficient window style aggregation. The first type as illustrated in the following introduces a new dimension member ALL_2002_products and computes its value to be the sum of year 2002 sales for all products:

```
MODEL PARTITION BY (country) DIMENSION BY (product, time) MEASURES (sale sales)
RULES UPSERT
 (sales['ALL_2002_products', 2002] = SUM(sales)[ANY, 2002])
```

To evaluate the aggregate sum in this case, each partition will be scanned to find the cells for 2002 for all products and they will be aggregated. If the left side of the rule were to reference multiple cells, then Oracle will have to compute the right side aggregate by scanning the partition for each cell referenced on the left. For example, consider the following example:

```
MODEL PARTITION BY (country) DIMENSION BY (product, time)
  MEASURES (sale sales, 0 avg_exclusive)
  RULES UPDATE
  (avg_exclusive[ANY, 2002] =  AVG(sales)[product <> CV(product), CV(time)])
```

This rule calculates a measure called `avg_exclusive` for every product in 2002. The measure `avg_exclusive` is defined as the average sales of all products excluding the current product. In this case, Oracle scans the data in a partition for every product in 2002 to calculate the aggregate, and this may be expensive

Oracle Database will optimize the evaluation of such aggregates in some scenarios with window-style computation as used in analytic functions. These scenarios involve rules with multi-cell references on their left side and computing window computations such as moving averages, cumulative sums and so on. Consider the following example:

```
MODEL PARTITION BY (country) DIMENSION BY (product, time)
  MEASURES (sale sales, 0 mavg)
  RULES UPDATE
  (mavg[product IN ('Bounce', 'Y Box', 'Mouse Pad'), ANY] =
   AVG(sales)[CV(product), time BETWEEN CV(time)
   AND CV(time) - 2])
```

It computes the moving average of sales for products Bounce, Y Box, and Mouse Pad over a three year period. It would be very inefficient to evaluate the aggregate by scanning the partition for every cell referenced on the left side. Oracle identifies the computation as being in window-style and evaluates it efficiently. It sorts the input on product, time and then scans the data once to compute the moving average. You can view this rule as an analytic function being applied on the sales data for products Bounce, Y Box, and Mouse Pad:

```
AVG(sales) OVER (PARTITION BY product ORDER BY time
RANGE BETWEEN 2 PRECEDING AND CURRENT ROW)
```

This computation style is called `WINDOW (IN MODEL) SORT`. This style of aggregation is applicable when the rule has a multi-cell reference on its left side with no `ORDER BY`, has a simple aggregate (`SUM`, `COUNT`, `MIN`, `MAX`, `STDEV`, and

VAR) on its right side, only one dimension on the right side has a boolean predicate (<, <=, >, >=, BETWEEN), and all other dimensions on the right are qualified with CV.

Using EXPLAIN PLAN to Understand Model Queries

You will see a line in the main explain plan output showing the model and the algorithm used. Reference models are tagged with the keyword REFERENCE in the plan output. Also, Oracle annotates the plan with WINDOW (IN MODEL) SORT if any of the rules qualify for window-style aggregate computation.

By examining an explain plan, you can find out the algorithm chosen to evaluate your model. If your model has SEQUENTIAL ORDER semantics, then ORDERED is displayed. For AUTOMATIC ORDER models, Oracle displays ACYCLIC or CYCLIC based on whether it chooses ACYCLIC or CYCLIC algorithm for evaluation. In addition, the plan output will have an annotation FAST in case of ORDERED and ACYCLIC algorithms if all left side cell references are single cell references and aggregates, if any, on the right side of rules are simple arithmetic non-distinct aggregates like SUM, COUNT, AVG, and so on. Rule evaluation in this case would be highly efficient and hence the annotation FAST. Thus, the output you will see in the explain plan would be MODEL {ORDERED [FAST] | ACYCLIC [FAST] | CYCLIC}.

Using ORDERED FAST: Example

This model has only single cell references on the left side of rules and the aggregate AVG on the right side of first rule is a simple non-distinct aggregate:

```
EXPLAIN PLAN FOR
SELECT country, prod, year, sales
FROM sales_view
WHERE country IN ('Italy', 'Japan')
MODEL UNIQUE DIMENSION
  PARTITION BY (country) DIMENSION BY (prod, year) MEASURES (sale sales)
  RULES UPSERT
  (sales['Bounce', 2003] = AVG(sales)[ANY, 2002] * 1.24,
   sales['Y Box', 2003] = sales['Bounce', 2003] * 0.25);
```

Using ORDERED: Example

Because the left side of the second rule is a multi-cell reference, the FAST method will not be chosen in the following:

```
EXPLAIN PLAN FOR
SELECT country, prod, year, sales
```

```
FROM sales_view
WHERE country IN ('Italy', 'Japan')
MODEL UNIQUE DIMENSION
  PARTITION BY (country) DIMENSION BY (prod, year) MEASURES (sale sales)
  RULES UPSERT
  (sales['Bounce', 2003] = AVG(sales)[ANY, 2002] * 1.24
   sales[prod <> 'Bounce', 2003] = sales['Bounce', 2003] * 0.25);
```

Using ACYCLIC FAST: Example

Rules in this model are not cyclic and the explain plan will show ACYCLIC. The FAST method is chosen in this case as well.

```
EXPLAIN PLAN FOR
SELECT country, prod, year, sales
FROM sales_view
WHERE country IN ('Italy', 'Japan')
MODEL UNIQUE DIMENSION
  PARTITION BY (country) DIMENSION BY (prod, year) MEASURES (sale sales)
  RULES UPSERT AUTOMATIC ORDER
  (sales['Y Box', 2003] = sales['Bounce', 2003] * 0.25,
   sales['Bounce', 2003] = sales['Bounce', 2002] / SUM(sales)[ANY, 2002] * 2 *
   sales['All Products', 2003],
   sales['All Products', 2003] = 200000);
```

Using ACYCLIC: Example

Rules in this model are not cyclic. The PERCENTILE_DISC aggregate that gives the median sales for year 2002, in the second rule is not a simple aggregate function. Therefore, Oracle will not choose the FAST method, and the explain plan will just show ACYCLIC.

```
SELECT country, prod, year, sales
FROM sales_view
WHERE country IN ('Italy', 'Japan')
MODEL UNIQUE DIMENSION
  PARTITION BY (country) DIMENSION BY (prod, year) MEASURES (sale sales)
  RULES UPSERT AUTOMATIC ORDER
  (sales['Y Box', 2003] = sales['Bounce', 2003] * 0.25,
   sales['Bounce',2003] = PERCENTILE_DISC(0.5) WITHIN GROUP (ORDER BY
     sales)[ANY,2002] / SUM(sales)[ANY, 2002] * 2 * sales['All Products', 2003],
   sales['All Products', 2003] = 200000);
```

Using CYCLIC: Example

Oracle chooses CYCLIC algorithm for this model as there is a cycle among second and third rules.

```
EXPLAIN PLAN FOR
SELECT country, prod, year, sales
FROM sales_view
WHERE country IN ('Italy', 'Japan')
MODEL UNIQUE DIMENSION
  PARTITION BY (country) DIMENSION BY (prod, year) MEASURES (sale sales)
  IGNORE NAV RULES UPSERT AUTOMATIC ORDER
  (sales['All Products', 2003] = 200000,
   sales['Y Box', 2003] = sales['Bounce', 2003] * 0.25,
   sales['Bounce', 2003] = sales['Y Box', 2003] +
    (sales['Bounce', 2002] / SUM(sales)[ANY, 2002] * 2 *
     sales['All Products', 2003]));
```

Examples of SQL Modeling

The examples in this section assume that in addition to sales_view, you have the following view defined. It finds monthly totals of sales and quantities by product and country.

```
CREATE VIEW sales_view2 AS
SELECT country_name country, prod_name product, calendar_year year,
  calendar_month_name month, SUM(amount_sold) sale, COUNT(amount_sold) cnt
FROM sales, times, customers, countries, products
WHERE sales.time_id = times.time_id AND
      sales.prod_id = products.prod_id AND
      sales.cust_id = customers.cust_id AND
      customers.country_id = countries.country_id
GROUP BY country_name, prod_name, calendar_year, calendar_month_name;
```

Example 1 Calculating Sales Differences

Show the sales for Italy and Spain and the difference between the two for each product. The difference should be placed in a new row with country = 'Diff Italy-Spain'.

```
SELECT product, country, sales
FROM sales_view
WHERE country IN ('Italy', 'Spain')
GROUP BY product, country
MODEL
  PARTITION BY (product) DIMENSION BY (country) MEASURES (SUM(sales) AS sales)
```

```
  RULES UPSERT
  (sales['DIFF ITALY-SPAIN'] = sales['Italy'] - sales['Spain']);
```

Example 2 Calculating Percentage Change

If sales for each product in each country grew (or declined) at the same monthly rate from November 2000 to December 2000 as they did from October 2000 to November 2000, what would the fourth quarter's sales be for the whole company and for each country?

```
SELECT country, SUM(sales)
FROM (SELECT product, country, month, sales
      FROM sales_view2
      WHERE year=2000 AND month IN ('October','November')
MODEL
  PARTITION BY (product, country) DIMENSION BY (month) MEASURES (sale sales)
  RULES
   (sales['December']=(sales['November'] /sales['October']) *sales['November']))
GROUP BY GROUPING SETS ((),(country));
```

Example 3 Calculating Net Present Value

You want to calculate the net present value (NPV) of a series of periodic cash flows. Your scenario involves two projects, each of which starts with an initial investment at time 0, represented as a negative cash flow. The initial investment is followed by three years of positive cash flow. First, create a table (`cash_flow`) and populate it with some data, as in the following statements:

```
CREATE TABLE cash_flow (year DATE, i INTEGER, prod VARCHAR2(3), amount NUMBER);
INSERT INTO cash_flow VALUES (TO_DATE('1999', 'YYYY'),  0,  'vcr',  -100.00);
INSERT INTO cash_flow VALUES (TO_DATE('2000', 'YYYY'),  1,  'vcr',   12.00);
INSERT INTO cash_flow VALUES (TO_DATE('2001', 'YYYY'),  2,  'vcr',  10.00);
INSERT INTO cash_flow VALUES (TO_DATE('2002', 'YYYY'),  3,  'vcr',  20.00);
INSERT INTO cash_flow VALUES (TO_DATE('1999', 'YYYY'),  0,  'dvd',  -200.00);
INSERT INTO cash_flow VALUES (TO_DATE('2000', 'YYYY'),  1,  'dvd',  22.00);
INSERT INTO cash_flow VALUES (TO_DATE('2001', 'YYYY'),  2,  'dvd',  12.00);
INSERT INTO cash_flow VALUES (TO_DATE('2002', 'YYYY'),  3,  'dvd',  14.00);
```

To calculate the NPV using a discount rate of 0.14, issue the following statement:

```
SELECT year, i, prod, amount, npv
FROM cash_flow
MODEL PARTITION BY (prod)
  DIMENSION BY (i)
  MEASURES (amount, 0 npv, year)
  RULES
```

```
    (npv[0] = amount[0],
     npv[i !=0] ORDER BY i =
       amount[CV()]/ POWER(1.14,CV(i)) + npv[CV(i)-1]);

YEAR               I PRO     AMOUNT        NPV
--------- ---------- --- ---------- ----------
01-AUG-99          0 dvd       -200       -200
01-AUG-00          1 dvd         22 -180.70175
01-AUG-01          2 dvd         12 -171.46814
01-AUG-02          3 dvd         14 -162.01854
01-AUG-99          0 vcr       -100       -100
01-AUG-00          1 vcr         12 -89.473684
01-AUG-01          2 vcr         10 -81.779009
01-AUG-02          3 vcr         20 -68.279579
```

Example 4 Calculating Using Simultaneous Equations

You want your interest expenses to equal 30% of your net income (net=pay minus tax minus interest). Interest is tax deductible from gross, and taxes are 38% of salary and 28% capital gains. You have salary of $100,000 and capital gains of $15,000. Net income, taxes, and interest expenses are unknown. Observe that this is a simultaneous equation (net depends on interest, which depends on net), thus the ITERATE clause is included.

First, create a table called `ledger`:

```
CREATE TABLE  ledger  (account  VARCHAR2(20), balance  NUMBER(10,2) );
```

Then, insert the following five rows:

```
INSERT INTO ledger VALUES  ('Salary', 100000);
INSERT INTO ledger VALUES  ('Capital_gains', 15000);
INSERT INTO ledger VALUES  ('Net', 0);
INSERT INTO ledger VALUES  ('Tax', 0);
INSERT INTO ledger VALUES  ('Interest', 0);
```

Next, issue the following statement:

```
SELECT s, account
FROM ledger
MODEL
  DIMENSION  BY (account) MEASURES (balance s)
  RULES  ITERATE (100)
  (s['Net']=s['Salary']-s['Interest']-s['Tax'],
 s['Tax']=(s['Salary']-s['Interest'])*0.38 + s['Capital_gains']*0.28,
  s['Interest']=s['Net']*0.30);
```

The output (with numbers rounded) is:

```
         S ACCOUNT
---------- --------------------
    100000 Salary
     15000 Capital_gains
48735.2445 Net
36644.1821 Tax
14620.5734 Interest
```

Example 5 Calculating Using Regression

The sales of Bounce in 2001 will increase in comparison to 2000 as they did in the last three years (between 1998 and 2000). Sales of Shaving Cream in 2001 will also increase in comparison to 2000 as they did between 1998 and 2000. To calculate the increase, use the regression function REGR_SLOPE as follows. Because we are calculating the next period's value, it is sufficient to add the slope to the 2000 value.

```
SELECT * FROM
 (SELECT country, product, year, projected_sale, sales
  FROM sales_view
  WHERE country IN ('Italy', 'Japan') AND product IN ('Shaving Cream', 'Bounce')
MODEL
  PARTITION BY (country) DIMENSION BY (product, year)
  MEASURES (sales sales, year y, CAST(NULL AS NUMBER) projected_sale) IGNORE NAV
  RULES UPSERT
  (projected_sale[FOR product IN ('Bounce', 'Shaving Cream'), 2001] =
             sales[CV(), 2000] +
             REGR_SLOPE(sales, y)[CV(), year BETWEEN 1998 AND 2000]))
ORDER BY country, product, year;
```

The output is as follows:

```
COUNTRY    PRODUCT           YEAR   PROJECTED_SALE  SALES
-------    -------           ----   --------------  -------
Italy      Bounce            1999                    2474.78
Italy      Bounce            2000                    4333.69
Italy      Bounce            2001           6192.6   4846.3
Italy      Shaving Cream     2001
Japan      Bounce            1999                     2961.3
Japan      Bounce            2000                     5133.53
Japan      Bounce            2001          7305.76    6303.6
Japan      Shaving Cream     2001
```

Example 6 Calculating Mortgage Amortization

This example creates mortgage amortization tables for any number of customers, using information about mortgage loans selected from a table of mortgage facts. First, create two tables and insert needed data:

- `mortgage_facts`

  Holds information about individual customer loans, including the name of the customer, the fact about the loan that is stored in that row, and the value of that fact. The facts stored for this example are loan (`Loan`), annual interest rate (`Annual_Interest`), and number of payments (`Payments`) for the loan. Also, the values for two customers, Smith and Jones, are inserted.

  ```
  CREATE TABLE mortgage_facts (customer VARCHAR2(20), fact VARCHAR2(20),
  amount  NUMBER(10,2));
  INSERT INTO mortgage_facts  VALUES ('Smith', 'Loan', 100000);
  INSERT INTO mortgage_facts  VALUES ('Smith', 'Annual_Interest', 12);
  INSERT INTO mortgage_facts  VALUES ('Smith', 'Payments', 360);
  INSERT INTO mortgage_facts  VALUES ('Smith', 'Payment', 0);
  INSERT INTO mortgage_facts  VALUES ('Jones', 'Loan', 200000);
  INSERT INTO mortgage_facts  VALUES ('Jones', 'Annual_Interest', 12);
  INSERT INTO mortgage_facts  VALUES ('Jones', 'Payments', 180);
  INSERT INTO mortgage_facts  VALUES ('Jones', 'Payment', 0);
  ```

- `mortgage`

  Holds output information for the calculations. The columns are customer, payment number (`pmt_num`), principal applied in that payment (`principalp`), interest applied in that payment (`interestp`), and remaining loan balance (`mort_balance`). In order to upsert new cells into a partition, you need to have at least one row pre-existing per partition. Therefore, we seed the mortgage table with the values for the two customers before they have made any payments. This seed information could be easily generated using a SQL Insert statement based on the `mortgage_fact` table.

```
CREATE TABLE mortgage_facts (customer VARCHAR2(20), fact VARCHAR2(20),
   amount  NUMBER(10,2));

INSERT INTO mortgage_facts  VALUES ('Smith', 'Loan', 100000);
INSERT INTO mortgage_facts  VALUES ('Smith', 'Annual_Interest', 12);
INSERT INTO mortgage_facts  VALUES ('Smith', 'Payments', 360);
INSERT INTO mortgage_facts  VALUES ('Smith', 'Payment', 0);
INSERT INTO mortgage_facts  VALUES ('Jones', 'Loan', 200000);
INSERT INTO mortgage_facts  VALUES ('Jones', 'Annual_Interest', 12);
INSERT INTO mortgage_facts  VALUES ('Jones', 'Payments', 180);
```

```
INSERT INTO mortgage_facts  VALUES ('Jones', 'Payment', 0);

CREATE TABLE mortgage (customer VARCHAR2(20), pmt_num NUMBER(4),
   principalp NUMBER(10,2), interestp NUMBER(10,2), mort_balance NUMBER(10,2));

INSERT INTO mortgage VALUES ('Jones',0, 0, 0, 200000);
INSERT INTO mortgage VALUES ('Smith',0, 0, 0, 100000);
```

The following SQL statement is complex, so individual lines have been annotated as needed. These lines are explained in more detail later.

```
SELECT c, p, m, pp, ip
FROM MORTGAGE
MODEL                                                                --See 1
REFERENCE R ON
   (SELECT customer, fact, amt                                       --See 2
    FROM mortgage_facts
    MODEL DIMENSION BY (customer, fact) MEASURES (amount amt)        --See 3
      RULES
       (amt[any, 'PaymentAmt']= (amt[CV(),'Loan']*
          Power(1+ (amt[CV(),'Annual_Interest']/100/12),
                  amt[CV(),'Payments']) *
         (amt[CV(),'Annual_Interest']/100/12)) /
         (Power(1+(amt[CV(),'Annual_Interest']/100/12),
                amt[CV(),'Payments']) - 1)
       )
    )
  DIMENSION BY (customer cust, fact) measures (amt)                   --See 4
MAIN amortization
  PARTITION BY (customer c)                                          --See 5
  DIMENSION BY (0 p)                                                 --See 6
  MEASURES (principalp pp, interestp ip, mort_balance m, customer mc)  --See 7
  RULES
    ITERATE(1000) UNTIL (ITERATION_NUMBER+1 =
r.amt[mc[0],'Payments'])                                             --See 8
    (ip[ITERATION_NUMBER+1] = m[CV()-1] *
       r.amt[mc[0], 'Annual_Interest']/1200,                         --See 9
     pp[ITERATION_NUMBER+1] = r.amt[mc[0], 'PaymentAmt'] - ip[CV()],  --See 10
      m[ITERATION_NUMBER+1] = m[CV()-1] - pp[CV()]                   --See 11
    )
ORDER BY c, p;
```

The following numbers refer to the numbers listed in the example:

1: This is the start of the main model definition.

2 through 4: These lines mark the start and end of the reference model labeled R. This model defines defines a SELECT statement that calculates the monthly payment amount for each customer's loan. The SELECT statement uses its own MODEL clause starting at the line labeled 3 with a single rule that defines the amt value based on information from the mortgage_facts table. The measure returned by reference model R is amt, dimensioned by customer name cust and fact value fact as defined in the line labeled 4.

The reference model is computed once and the values are then used in the main model for computing other calculations. Reference model R will return a row for each existing row of mortgage_fact, and it will return the newly calculated rows for each customer where the fact type is Payment and the amt is the monthly payment amount. If we wish to use a specific amount from the R output, we address it with the expression r.amt[<customer_name>,<fact_name>].

5: This is the continuation of the main model definition. We will partition the output by customer, aliased as c.

6: The main model is dimensioned with a constant value of 0, aliased as p. This represents the payment number of a row.

7: Four measures are defined: principalp (pp) is the principal amount applied to the loan in the month, interestp (ip) is the interest paid that month, mort_balance (m) is the remaining mortgage value after the payment of the loan, and customer (mc) is used to support the partitioning.

8: This begins the rules block. It will perform the rule calculations up to 1000 times. Because the calculations are performed once for each month for each customer, the maximum number of months that can be specified for a loan is 1000. Iteration is stopped when the ITERATION_NUMBER+1 equals the amount of payments derived from reference R. Note that the value from reference R is the amt (amount) measure defined in the reference clause. This reference value is addressed as r.amt[<customer_name>,<fact>]. The expression used in the iterate line, "r.amt[mc[0], 'Payments']" is resolved to be the amount from reference R, where the customer name is the value resolved by mc[0]. Since each partition contains only one customer, mc[0] can have only one value. Thus "r.amt[mc[0], 'Payments']" yields the reference clause's value for the number of payments for the current customer. This means that the rules will be performed as many times as there are payments for that customer.

9 through 11: The first two rules in this block use the same type of r.amt reference that was explained in 8. The difference is that the ip rule defines the fact value as Annual_Interest. Note that each rule refers to the value of one of the other measures. The expression used on the left side of each rule, "[ITERATION_

NUMBER+1]" will create a new dimension value, so the measure will be upserted into the result set. Thus the result will include a monthly amortization row for all payments for each customer.

The final line of the example sorts the results by customer and loan payment number.

23

OLAP and Data Mining

In large data warehouse environments, many different types of analysis can occur. In addition to SQL queries, you may also apply more advanced analytical operations to your data. Two major types of such analysis are OLAP (On-Line Analytic Processing) and data mining. Rather than having a separate OLAP or data mining engine, Oracle has integrated OLAP and data mining capabilities directly into the database server. Oracle OLAP and Oracle Data Mining (ODM) are options to the Oracle Database. This chapter provides a brief introduction to these technologies, and more detail can be found in these products' respective documentation.

The following topics provide an introduction to Oracle's OLAP and data mining capabilities:

- OLAP Overview
- Oracle Data Mining Overview

> **See Also:** *Oracle OLAP Application Developer's Guide* for further information regarding OLAP and *Oracle Data Mining* documentation for further information regarding data mining

OLAP Overview

Oracle Database OLAP adds the query performance and calculation capability previously found only in multidimensional databases to Oracle's relational platform. In addition, it provides a Java OLAP API that is appropriate for the development of internet-ready analytical applications. Unlike other combinations of OLAP and RDBMS technology, Oracle Database OLAP is not a multidimensional database using bridges to move data from the relational data store to a multidimensional data store. Instead, it is truly an OLAP-enabled relational database. As a result, this release provides the benefits of a multidimensional database along with the scalability, accessibility, security, manageability, and high availability of the Oracle Database. The Java OLAP API, which is specifically designed for internet-based analytical applications, offers productive data access. See *Oracle OLAP Application Developer's Guide* for further information regarding OLAP.

Benefits of OLAP and RDBMS Integration

Basing an OLAP system directly on the Oracle server offers the following benefits:

- Scalability
- Availability
- Manageability
- Backup and Recovery
- Security

Scalability

Oracle Database OLAP is highly scalable. In today's environment, there is tremendous growth along three dimensions of analytic applications: number of users, size of data, complexity of analyses. There are more users of analytical applications, and they need access to more data to perform more sophisticated analysis and target marketing. For example, a telephone company might want a customer dimension to include detail such as all telephone numbers as part of an application that is used to analyze customer turnover. This would require support for multi-million row dimension tables and very large volumes of fact data. Oracle Database can handle very large data sets using parallel execution and partitioning, as well as offering support for advanced hardware and clustering.

Availability

Oracle Database includes many features that support high availability. One of the most significant is partitioning, which allows management of precise subsets of tables and indexes, so that management operations affect only small pieces of these data structures. By partitioning tables and indexes, data management processing time is reduced, thus minimizing the time data is unavailable. Another feature supporting high availability is transportable tablespaces. With transportable tablespaces, large data sets, including tables and indexes, can be added with almost no processing to other databases. This enables extremely rapid data loading and updates.

Manageability

Oracle enables you to precisely control resource utilization. The Database Resource Manager, for example, provides a mechanism for allocating the resources of a data warehouse among different sets of end-users. Consider an environment where the marketing department and the sales department share an OLAP system. Using the Database Resource Manager, you could specify that the marketing department receive at least 60 percent of the CPU resources of the machines, while the sales department receive 40 percent of the CPU resources. You can also further specify limits on the total number of active sessions, and the degree of parallelism of individual queries for each department.

Another resource management facility is the progress monitor, which gives end users and administrators the status of long-running operations. Oracle Database 10*g* maintains statistics describing the percent-complete of these operations. Oracle Enterprise Manager enables you to view a bar-graph display of these operations showing what percent complete they are. Moreover, any other tool or any database administrator can also retrieve progress information directly from the Oracle data server, using system views.

Backup and Recovery

Oracle provides a server-managed infrastructure for backup, restore, and recovery tasks that enables simpler, safer operations at terabyte scale. Some of the highlights are:

- Details related to backup, restore, and recovery operations are maintained by the server in a recovery catalog and automatically used as part of these operations. This reduces administrative burden and minimizes the possibility of human errors.

- Backup and recovery operations are fully integrated with partitioning. Individual partitions, when placed in their own tablespaces, can be backed up and restored independently of the other partitions of a table.
- Oracle includes support for incremental backup and recovery using Recovery Manager, enabling operations to be completed efficiently within times proportional to the amount of changes, rather than the overall size of the database.
- The backup and recovery technology is highly scalable, and provides tight interfaces to industry-leading media management subsystems. This provides for efficient operations that can scale up to handle very large volumes of data. Open Platforms for more hardware options & enterprise-level platforms.

Security

Just as the demands of real-world transaction processing required Oracle to develop robust features for scalability, manageability and backup and recovery, they lead Oracle to create industry-leading security features. The security features in Oracle have reached the highest levels of U.S. government certification for database trustworthiness. Oracle's fine grained access control feature, enables cell-level security for OLAP users. Fine grained access control works with minimal burden on query processing, and it enables efficient centralized security management.

Oracle Data Mining Overview

Oracle Data Mining uses data mining algorithms to sift through the large volumes of data generated by businesses to produce, evaluate, and deploy predictive and descriptive models. It also enriches mission critical applications in CRM, manufacturing control, inventory management, customer service and support, Web portals, wireless devices and other fields with context-specific recommendations and predictive monitoring of critical processes. ODM delivers real-time answers to questions such as:

- Which items is a person most likely to buy or like?
- What is the likelihood that this product will be returned for repair?
- What is the likelihood that this person poses a credit risk?

Oracle Data Mining enables data mining inside the database for performance and scalability. Some of the capabilities are:

- Java and PL/SQL interfaces that provide programmatic control and application integration

- Several algorithms:
  - Classification: Naive Bayes, Adaptive Bayes Network, Support Vector Machine
  - Regression: Support Vector Machine
  - Clustering: k-Means, O-Cluster
  - Association: Apriori
  - Attribute Importance: Predictor Variance
  - Feature Extraction: Non-Negative Matrix Factorization
- Real-time and batch scoring modes

Oracle Data Mining also supports sequence similarity search and annotation (BLAST) in the database.

Enabling Data Mining Applications

Oracle Data Mining provides a Java API and PL/SQL packages to exploit the data mining functionality that is embedded in the Oracle database.

By delivering complete programmatic control of data mining in the database, Oracle Data Mining (ODM) delivers powerful, scalable modeling and real-time scoring. This enables businesses to incorporate rules, predictions, and classifications in all processes and decision points throughout the business cycle.

ODM is designed to meet the challenges of vast amounts of data, delivering accurate insights completely integrated into e-business applications. This integrated intelligence enables the automation and decision speed that e-businesses require to compete today.

Data Mining in the Database

Oracle Data Mining performs all phases of data mining within the database. In each data mining phase, this architecture results in significant improvements including performance, automation, and integration.

Performing data mining in the database has the following benefits:

- All phases of data mining take place in the database:
  - All data preparation occurs in the database
  - The data that is mined remains in the database

  - The models produced by mining reside in the database
  - Scoring occurs in the database along with results immediately available as tables
- Data mining automatically inherits important database features, including:
  - Scalability
  - Availability
  - Manageability
  - Backup and recovery
  - Security

Data Preparation

Data preparation usually requires the creation of new tables or views based on existing data. Both options perform faster than moving data to an external data mining utility and offer the programmer the option of snapshots or real-time updates.

Oracle Data Mining provides utilities for complex, data mining-specific tasks. For example, for some types of models, binning improves model build time and model performance; therefore, ODM provides a utility for user-defined binning.

ODM accepts data in either non-transactional (single-record case) format or transactional (multi-record case) format. ODM provides a pivoting utility for converting multiple non-transactional tables into a single transactional table.

ODM data exploration and model evaluation features are extended by Oracle's statistical functions and OLAP capabilities. Because these also operate within the database, they can all be incorporated into a seamless application that shares database objects. This allows for more functional and faster applications.

Model Building

Oracle Data Mining supports all the major data mining functions: classification, regression, association rules, clustering, attribute importance, and feature extraction.

These algorithms address a broad spectrum of business problems, ranging from predicting the future likelihood of a customer purchasing a given product, to understand which products are likely to be purchased together in a single trip to the grocery store. Since all model building takes place inside the database, the data

never needs to move outside the database, and therefore the entire data-mining process is accelerated.

Model Evaluation

Models are stored in the database and are directly accessible for evaluation, reporting, and further analysis by a wide variety of tools and application functions. ODM provides APIs for calculating confusion matrix and lift charts. ODM stores the models, the underlying data, and the results of model evaluation together in the database to enable further analysis, reporting, and application-specific model management.

Model Apply (Scoring)

Oracle Data Mining provides both batch and real-time scoring. In batch mode, ODM takes a table as input. It scores every record, and returns a scored table as a result. In real-time mode, parameters for a single record are passed in and the scores are returned in a Java object.

In both modes, ODM can deliver a variety of scores. It can return a rating or probability of a specific outcome. Alternatively it can return a predicted outcome and the probability of occurrence of the outcome.

ODM Programmatic Interfaces

ODM provides Java and PL/SQL interfaces for data mining. These interfaces make is possible to embed data mining in applications.

ODM Java API

The ODM Java API allows programmers to develop data mining applications or tools in the J2SE/J2EE environment. The API defines a set of classes that can be used to develop a complete data mining solution. The API has built-in data mining metadata management and provides an infrastructure to build data mining applications easily. The API supports importing and exporting of PMML models for Naive Bayes and Association Rules models. The API supports asynchronous execution of mining operations and provides mechanisms to retrieve state transition information for running or completed operations. The API supports real-time scoring for all the supervised models and clustering models. This API supports text mining.

The ODM Java API provides flexible data preparation options. Applications can use either automated data preparation, or they perform data processing using external transformations defined as utility methods in the

oracle.dmt.odm.transformation.Transformation class. In addition, applications can embed externals transformation details in the logical data specification as an input for the build operation, then the system will persist the details with the model and perform the embedded transformations in the future apply and test operations.

The ODM Java API design reflects concepts present in the emerging Java standard (JSR-73) for Data Mining, which is being developed through the Java Community Process.

ODM PL/SQL Packages

The following supplied packages support data mining in PL/SQL programs:

- `DBMS_DATA_MINING`
- `DBMS_DATA_MINING_TRANSFORM`

`DBMS_DATA_MINING` provides support for in-database data mining. This package can be used to build and test models and to apply models to new data (scoring). The package provides the basic building blocks for data mining, along with utilities and functions to inspect models and their results. The package also supports export and import of native models from a user's schema or database instance.

`DBMS_DATA_MINING_TRANSFORM`, a complementary package, provides support for popular data transformations such as numerical and categorical binning and linear and z-score normalization. The `DBMS_DATA_MINING_TRANSFORM` package is open source in nature, in that the package code is distributed with the product, so that users can study the utility routines and learn how to define their own data transformations using Oracle SQL and PL/SQL scripting.

ODM Sequence Similarity Search (BLAST)

ODM also supports specialized sequence matching and annotation algorithms. In life sciences, vast quantities of data including nucleotide and amino acid sequences are stored, typically, in a database. This sequence data help biologists determine the chemical structure, biological function, and evolutionary history of organisms.

A version of BLAST, based on NCBI BLAST 2.0, has been implemented in the Oracle Database using table functions. This enables users to perform BLAST queries against data in an Oracle database.

24

Using Parallel Execution

This chapter covers tuning in a parallel execution environment and discusses:

- Introduction to Parallel Execution Tuning
- How Parallel Execution Works
- Types of Parallelism
- Initializing and Tuning Parameters for Parallel Execution
- Tuning General Parameters for Parallel Execution
- Monitoring and Diagnosing Parallel Execution Performance
- Affinity and Parallel Operations
- Miscellaneous Parallel Execution Tuning Tips

> **Note:** Some features described in this chapter are available only if you have purchased Oracle Database Enterprise Edition with the Oracle Real Application Clusters Option.

Introduction to Parallel Execution Tuning

Parallel execution dramatically reduces response time for data-intensive operations on large databases typically associated with decision support systems (DSS) and data warehouses. You can also implement parallel execution on certain types of online transaction processing (OLTP) and hybrid systems. Parallel execution improves processing for:

- Queries requiring large table scans, joins, or partitioned index scans
- Creation of large indexes
- Creation of large tables (including materialized views)
- Bulk inserts, updates, merges, and deletes

You can also use parallel execution to access object types within an Oracle database. For example, you can use parallel execution to access large objects (LOBs).

Parallel execution benefits systems with all of the following characteristics:

- Symmetric multiprocessors (SMPs), clusters, or massively parallel systems
- Sufficient I/O bandwidth
- Underutilized or intermittently used CPUs (for example, systems where CPU usage is typically less than 30%)
- Sufficient memory to support additional memory-intensive processes, such as sorts, hashing, and I/O buffers

If your system lacks any of these characteristics, parallel execution might not significantly improve performance. In fact, parallel execution may reduce system performance on overutilized systems or systems with small I/O bandwidth.

When to Implement Parallel Execution

The benefits of parallel execution can be seen in DSS and data warehousing environments. OLTP systems can also benefit from parallel execution during batch processing and during schema maintenance operations such as creation of indexes. The average simple DML or SELECT statements that characterize OLTP applications would not see any benefit from being executed in parallel.

When Not to Implement Parallel Execution

Parallel execution is not normally useful for:

- Environments in which the typical query or transaction is very short (a few seconds or less). This includes most online transaction systems. Parallel execution is not useful in these environments because there is a cost associated with coordinating the parallel execution servers; for short transactions, the cost of this coordination may outweigh the benefits of parallelism.
- Environments in which the CPU, memory, or I/O resources are already heavily utilized. Parallel execution is designed to exploit additional available hardware resources; if no such resources are available, then parallel execution will not yield any benefits and indeed may be detrimental to performance.

Operations That Can Be Parallelized

You can use parallel execution for any of the following:

- Access methods

  Some examples are table scans, index full scans, and partitioned index range scans.

- Join methods

  Some examples are nested loop, sort merge, hash, and star transformation.

- DDL statements

  Some examples are `CREATE TABLE AS SELECT`, `CREATE INDEX`, `REBUILD INDEX`, `REBUILD INDEX PARTITION`, and `MOVE/SPLIT/COALESCE PARTITION`.

  You can normally use parallel DDL where you use regular DDL. There are, however, some additional details to consider when designing your database. One important restriction is that parallel DDL cannot be used on tables with object or `LOB` columns.

  All of these DDL operations can be performed in `NOLOGGING` mode for either parallel or serial execution.

  The `CREATE TABLE` statement for an index-organized table can be parallelized either with or without an `AS SELECT` clause.

  Different parallelism is used for different operations. Parallel create (partitioned) table as select and parallel create (partitioned) index run with a degree of parallelism equal to the number of partitions.

  Parallel operations require accurate statistics to perform optimally.

- DML statements

Some examples are INSERT AS SELECT, updates, deletes, and MERGE operations.

Parallel DML (parallel insert, update, merge, and delete) uses parallel execution mechanisms to speed up or scale up large DML operations against large database tables and indexes. You can also use INSERT ... SELECT statements to insert rows into multiple tables as part of a single DML statement. You can normally use parallel DML where you use regular DML.

Although data manipulation language (DML) normally includes queries, the term parallel DML refers only to inserts, updates, upserts and deletes done in parallel.

- Miscellaneous SQL operations

  Some examples are GROUP BY, NOT IN, SELECT DISTINCT, UNION, UNION ALL, CUBE, and ROLLUP, as well as aggregate and table functions.

- Parallel query

  You can parallelize queries and subqueries in SELECT statements, as well as the query portions of DDL statements and DML statements (INSERT, UPDATE, DELETE, and MERGE).

- SQL*Loader

  You can parallelize the use of SQL*Loader, where large amounts of data are routinely encountered. To speed up your loads, you can use a parallel direct-path load as in the following example:

  ```
  sqlldr USERID=SCOTT/TIGER CONTROL=LOAD1.CTL DIRECT=TRUE PARALLEL=TRUE
  sqlldr USERID=SCOTT/TIGER CONTROL=LOAD2.CTL DIRECT=TRUE PARALLEL=TRUE
  sqlldr USERID=SCOTT/TIGER CONTROL=LOAD3.CTL DIRECT=TRUE PARALLEL=TRUE
  ```

  You can also use a parameter file to achieve the same thing.

  An important point to remember is that indexes are not maintained during a parallel load.

How Parallel Execution Works

Parallel execution divides the task of executing a SQL statement into multiple small units, each of which is executed by a separate process. Also the incoming data (tables, indexes, partitions) can be divided into parts called granules. The user shadow process that wants to execute a query in parallel takes on the role as

parallel execution coordinator or query coordinator. The query coordinator does the following:

- Parses the query and determines the degree of parallelism
- Allocates one or two set of slaves (threads or processes)
- Controls the query and sends instructions to the PQ slaves
- Determines which tables or indexes need to be scanned by the PQ slaves
- Produces the final output to the user

Degree of Parallelism

The parallel execution coordinator may enlist two or more of the instance's parallel execution servers to process a SQL statement. The number of parallel execution servers associated with a single operation is known as the **degree of parallelism**.

A single operation is a part of a SQL statement such as an order by, a full table scan to perform a join on a nonindexed column table.

Note that the degree of parallelism applies directly only to intra-operation parallelism. If inter-operation parallelism is possible, the total number of parallel execution servers for a statement can be twice the specified degree of parallelism. No more than two sets of parallel execution servers can run simultaneously. Each set of parallel execution servers may process multiple operations. Only two sets of parallel execution servers need to be active to guarantee optimal inter-operation parallelism.

Parallel execution is designed to effectively use multiple CPUs and disks to answer queries quickly. When multiple users use parallel execution at the same time, it is easy to quickly exhaust available CPU, memory, and disk resources.

Oracle Database provides several ways to manage resource utilization in conjunction with parallel execution environments, including:

- The adaptive multiuser algorithm, which is enabled by default, reduces the degree of parallelism as the load on the system increases.
- User resource limits and profiles, which allow you to set limits on the amount of various system resources available to each user as part of a user's security domain.
- The Database Resource Manager, which lets you allocate resources to different groups of users.

The Parallel Execution Server Pool

When an instance starts up, Oracle creates a pool of parallel execution servers which are available for any parallel operation. The initialization parameter PARALLEL_MIN_SERVERS specifies the number of parallel execution servers that Oracle Database creates at instance startup.

When executing a parallel operation, the parallel execution coordinator obtains parallel execution servers from the pool and assigns them to the operation. If necessary, Oracle can create additional parallel execution servers for the operation. These parallel execution servers remain with the operation throughout job execution, then become available for other operations. After the statement has been processed completely, the parallel execution servers return to the pool.

Note that the parallel execution coordinator and the parallel execution servers can only service one statement at a time. A parallel execution coordinator cannot coordinate, for example, a parallel query and a parallel DML statement at the same time.

When a user issues a SQL statement, the optimizer decides whether to execute the operations in parallel and determines the degree of parallelism (DOP) for each operation. You can specify the number of parallel execution servers required for an operation in various ways.

If the optimizer targets the statement for parallel processing, the following sequence of events takes place:

1. The SQL statement's foreground process becomes a parallel execution coordinator.
2. The parallel execution coordinator obtains as many parallel execution servers as needed (determined by the DOP) from the server pool or creates new parallel execution servers as needed.
3. Oracle executes the statement as a sequence of operations. Each operation is performed in parallel, if possible.
4. When statement processing is completed, the coordinator returns any resulting data to the user process that issued the statement and returns the parallel execution servers to the server pool.

The parallel execution coordinator calls upon the parallel execution servers during the execution of the SQL statement, not during the parsing of the statement. Therefore, when parallel execution is used with the shared server, the server process that processes the EXECUTE call of a user's statement becomes the parallel execution

coordinator for the statement. See "Setting the Degree of Parallelism for Parallel Execution" on page 24-32 for more information.

Variations in the Number of Parallel Execution Servers

If the number of parallel operations processed concurrently by an instance changes significantly, Oracle automatically changes the number of parallel execution servers in the pool.

If the number of parallel operations increases, Oracle creates additional parallel execution servers to handle incoming requests. However, Oracle never creates more parallel execution servers for an instance than the value specified by the initialization parameter PARALLEL_MAX_SERVERS.

If the number of parallel operations decreases, Oracle terminates any parallel execution servers that have been idle for a threshold period of time. Oracle does not reduce the size of the pool less than the value of PARALLEL_MIN_SERVERS, no matter how long the parallel execution servers have been idle.

Processing Without Enough Parallel Execution Servers

Oracle can process a parallel operation with fewer than the requested number of processes.

If all parallel execution servers in the pool are occupied and the maximum number of parallel execution servers has been started, the parallel execution coordinator switches to serial processing.

See "Minimum Number of Parallel Execution Servers" on page 24-35 for information about using the initialization parameter PARALLEL_MIN_PERCENT and "Tuning General Parameters for Parallel Execution" on page 24-47 for information about the PARALLEL_MIN_PERCENT and PARALLEL_MAX_SERVERS initialization parameters.

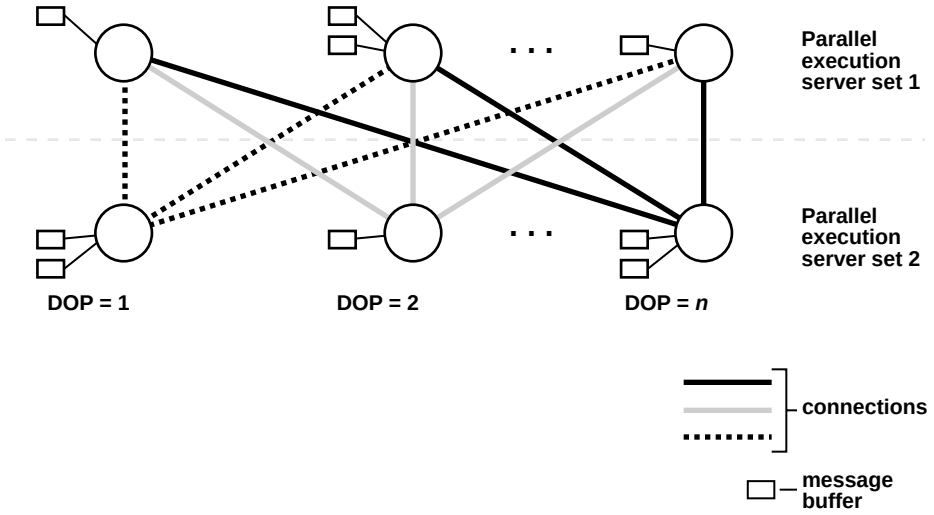
How Parallel Execution Servers Communicate

To execute a query in parallel, Oracle generally creates a producer queue server and a consumer server. The producer queue server retrieves rows from tables and the consumer server performs operations such as join, sort, DML, and DDL on these rows. Each server in the producer execution process set has a connection to each server in the consumer set. This means that the number of virtual connections between parallel execution servers increases as the square of the DOP.

Each communication channel has at least one, and sometimes up to four memory buffers. Multiple memory buffers facilitate asynchronous communication among the parallel execution servers.

A single-instance environment uses at most three buffers for each communication channel. An Oracle Real Application Clusters environment uses at most four buffers for each channel. Figure 24–1 illustrates message buffers and how producer parallel execution servers connect to consumer parallel execution servers.

Figure 24–1 Parallel Execution Server Connections and Buffers



When a connection is between two processes on the same instance, the servers communicate by passing the buffers back and forth. When the connection is between processes in different instances, the messages are sent using external high-speed network protocols. In Figure 24–1, the DOP is equal to the number of parallel execution servers, which in this case is *n*. Figure 24–1 does not show the parallel execution coordinator. Each parallel execution server actually has an additional connection to the parallel execution coordinator.

Parallelizing SQL Statements

Each SQL statement undergoes an optimization and parallelization process when it is parsed. When the data changes, if a more optimal execution or parallelization plan becomes available, Oracle can automatically adapt to the new situation.

After the optimizer determines the execution plan of a statement, the parallel execution coordinator determines the parallelization method for each operation in the plan. For example, the parallelization method might be to parallelize a full table scan by block range or parallelize an index range scan by partition. The coordinator must decide whether an operation can be performed in parallel and, if so, how many parallel execution servers to enlist. The number of parallel execution servers in one set is the DOP. See "Setting the Degree of Parallelism for Parallel Execution" on page 24-32 for more information.

Dividing Work Among Parallel Execution Servers

The parallel execution coordinator examines the redistribution requirements of each operation. An operation's redistribution requirement is the way in which the rows operated on by the operation must be divided or redistributed among the parallel execution servers.

After determining the redistribution requirement for each operation in the execution plan, the optimizer determines the order in which the operations must be performed. With this information, the optimizer determines the data flow of the statement.

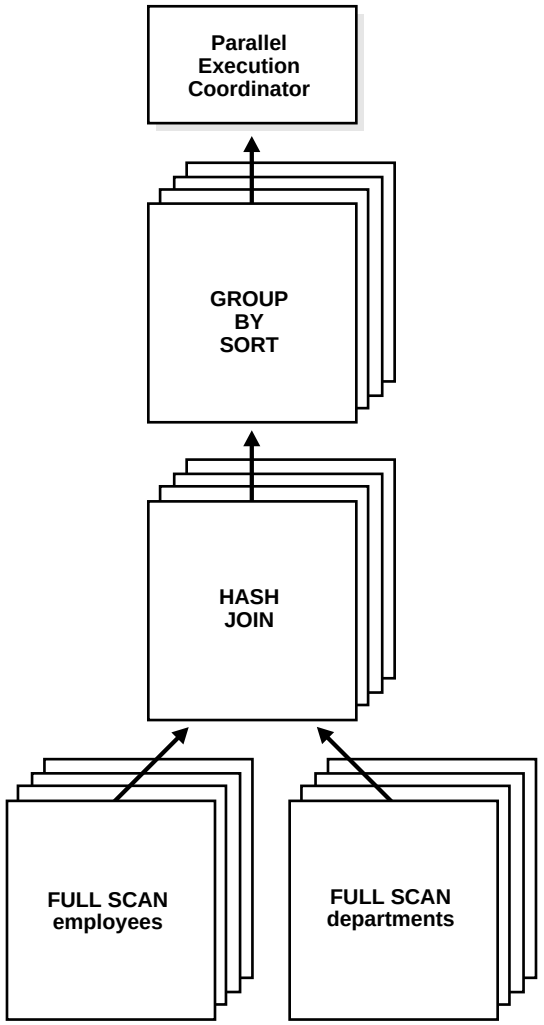
As an example of parallel query with intra- and inter-operation parallelism, consider the following, more complex query:

```
SELECT /*+ PARALLEL(employees 4) PARALLEL(departments 4)
       USE_HASH(employees) ORDERED */ MAX(salary), AVG(salary)
FROM employees, departments
WHERE employees.department_id = departments.department_id
GROUP BY employees.department_id;
```

Note that hints have been used in the query to force the join order and join method, and to specify the DOP of the tables `employees` and `departments`. In general, you should let the optimizer determine the order and method.

Figure 24–2 illustrates the data flow graph or query plan for this query.

Figure 24–2 Data Flow Diagram for Joining Tables



Parallelism Between Operations

Given two sets of parallel execution servers SS1 and SS2 for the query plan illustrated in Figure 24–2, the execution will proceed as follows: each server set (SS1 and SS2) will have four execution processes because of the PARALLEL hint in the

query that specifies the DOP. In other words, the DOP will be four because each set of parallel execution servers will have four processes.

Slave set SS1 first scans the table `employees` while SS2 will fetch rows from SS1 and build a hash table on the rows. In other words, the parent servers in SS2 and the child servers in SS2 work concurrently: one in scanning `employees` in parallel, the other in consuming rows sent to it from SS1 and building the hash table for the hash join in parallel. This is an example of inter-operation parallelism.

After SS1 has finished scanning the entire table `employees` (that is, all granules or task units for `employees` are exhausted), it scans the table `departments` in parallel. It sends its rows to servers in SS2, which then perform the probes to finish the hash-join in parallel. After SS1 is done scanning the table `departments` in parallel and sending the rows to SS2, it switches to performing the `GROUP BY` in parallel. This is how two server sets run concurrently to achieve inter-operation parallelism across various operators in the query tree while achieving intra-operation parallelism in executing each operation in parallel.

Another important aspect of parallel execution is the re-partitioning of rows while they are sent from servers in one server set to another. For the query plan in Figure 24–2, after a server process in SS1 scans a row of `employees`, which server process of SS2 should it send it to? The partitioning of rows flowing up the query tree is decided by the operator into which the rows are flowing into. In this case, the partitioning of rows flowing up from SS1 performing the parallel scan of `employees` into SS2 performing the parallel hash-join is done by hash partitioning on the join column value. That is, a server process scanning `employees` computes a hash function of the value of the column `employees.employee_id` to decide the number of the server process in SS2 to send it to. The partitioning method used in parallel queries is explicitly shown in the `EXPLAIN PLAN` of the query. Note that the partitioning of rows being sent between sets of execution servers should not be confused with Oracle's partitioning feature whereby tables can be partitioned using hash, range, and other methods.

Producer Operations

Operations that require the output of other operations are known as consumer operations. In Figure 24–2, the `GROUP BY SORT` operation is the producer of the `HASH JOIN` operation because `GROUP BY SORT` requires the `HASH JOIN` output.

Producer operations can begin consuming rows as soon as the producer operations have produced rows. In the previous example, while the parallel execution servers are producing rows in the `FULL SCAN departments` operation, another set of

parallel execution servers can begin to perform the HASH JOIN operation to consume the rows.

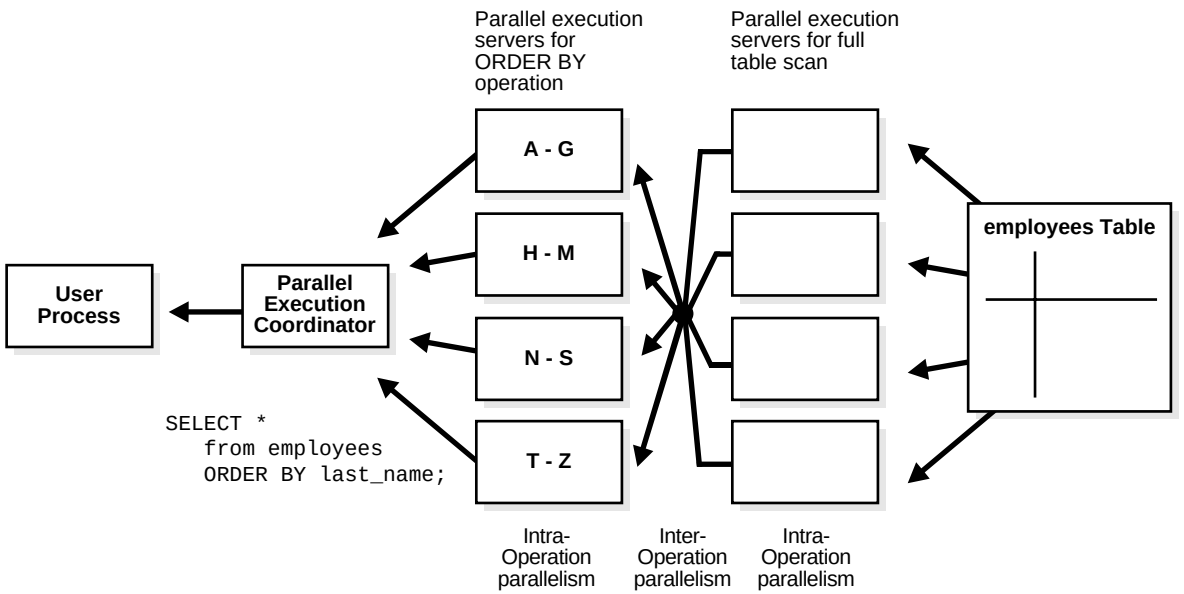
Each of the two operations performed concurrently is given its own set of parallel execution servers. Therefore, both query operations and the data flow tree itself have parallelism. The parallelism of an individual operation is called intraoperation parallelism and the parallelism between operations in a data flow tree is called interoperation parallelism. Due to the producer-consumer nature of the Oracle server's operations, only two operations in a given tree need to be performed simultaneously to minimize execution time. To illustrate intraoperation and interoperation parallelism, consider the following statement:

```
SELECT * FROM employees ORDER BY last_name;
```

The execution plan implements a full scan of the employees table. This operation is followed by a sorting of the retrieved rows, based on the value of the last_name column. For the sake of this example, assume the last_name column is not indexed. Also assume that the DOP for the query is set to 4, which means that four parallel execution servers can be active for any given operation.

Figure 24–3 illustrates the parallel execution of the example query.

Figure 24–3 Interoperation Parallelism and Dynamic Partitioning



As you can see from Figure 24–3, there are actually eight parallel execution servers involved in the query even though the DOP is 4. This is because a parent and child operator can be performed at the same time (interoperation parallelism).

Also note that all of the parallel execution servers involved in the scan operation send rows to the appropriate parallel execution server performing the SORT operation. If a row scanned by a parallel execution server contains a value for the last_name column between A and G, that row gets sent to the first ORDER BY parallel execution server. When the scan operation is complete, the sorting processes can return the sorted results to the coordinator, which, in turn, returns the complete query results to the user.

Types of Parallelism

The following types of parallelism are discussed in this section:

- Parallel Query
- Parallel DDL
- Parallel DML
- Parallel Execution of Functions
- Other Types of Parallelism

Parallel Query

You can parallelize queries and subqueries in SELECT statements. You can also parallelize the query portions of DDL statements and DML statements (INSERT, UPDATE, and DELETE). You can also query external tables in parallel.

See Also:

- "Operations That Can Be Parallelized" on page 24-3 for information on the query operations that Oracle can parallelize
- "Parallelizing SQL Statements" on page 24-8 for an explanation of how the processes perform parallel queries
- "Distributed Transaction Restrictions" on page 24-27 for examples of queries that reference a remote object
- "Rules for Parallelizing Queries" on page 24-37 for information on the conditions for parallelizing a query and the factors that determine the DOP

Parallel Queries on Index-Organized Tables

The following parallel scan methods are supported on index-organized tables:

- Parallel fast full scan of a nonpartitioned index-organized table
- Parallel fast full scan of a partitioned index-organized table
- Parallel index range scan of a partitioned index-organized table

These scan methods can be used for index-organized tables with overflow areas and for index-organized tables that contain LOBs.

Nonpartitioned Index-Organized Tables

Parallel query on a nonpartitioned index-organized table uses parallel fast full scan. The DOP is determined, in decreasing order of priority, by:

1. A `PARALLEL` hint (if present)
2. An `ALTER SESSION FORCE PARALLEL QUERY` statement
3. The parallel degree associated with the table, if the parallel degree is specified in the `CREATE TABLE` or `ALTER TABLE` statement

The allocation of work is done by dividing the index segment into a sufficiently large number of block ranges and then assigning the block ranges to parallel execution servers in a demand-driven manner. The overflow blocks corresponding to any row are accessed in a demand-driven manner only by the process which owns that row.

Partitioned Index-Organized Tables

Both index range scan and fast full scan can be performed in parallel. For parallel fast full scan, parallelization is exactly the same as for nonpartitioned index-organized tables. For parallel index range scan on partitioned index-organized tables, the DOP is the minimum of the degree picked up from the previous priority list (like in parallel fast full scan) and the number of partitions in the index-organized table. Depending on the DOP, each parallel execution server gets one or more partitions (assigned in a demand-driven manner), each of which contains the primary key index segment and the associated overflow segment, if any.

Parallel Queries on Object Types

Parallel queries can be performed on object type tables and tables containing object type columns. Parallel query for object types supports all of the features that are available for sequential queries on object types, including:

- Methods on object types
- Attribute access of object types
- Constructors to create object type instances
- Object views
- PL/SQL and OCI queries for object types

There are no limitations on the size of the object types for parallel queries.

The following restrictions apply to using parallel query for object types.

- A `MAP` function is needed to parallelize queries involving joins and sorts (through `ORDER BY`, `GROUP BY`, or set operations). In the absence of a `MAP` function, the query will automatically be executed serially.
- Parallel DML and parallel DDL are not supported with object types. DML and DDL statements are always performed serially.

In all cases where the query cannot execute in parallel because of any of these restrictions, the whole query executes serially without giving an error message.

Parallel DDL

This section includes the following topics on parallelism for DDL statements:

- DDL Statements That Can Be Parallelized
- CREATE TABLE ... AS SELECT in Parallel
- Recoverability and Parallel DDL
- Space Management for Parallel DDL

DDL Statements That Can Be Parallelized

You can parallelize DDL statements for tables and indexes that are nonpartitioned or partitioned. Table 24–3 on page 24-44 summarizes the operations that can be parallelized in DDL statements.

The parallel DDL statements for nonpartitioned tables and indexes are:

- CREATE INDEX
- CREATE TABLE ... AS SELECT
- ALTER INDEX ... REBUILD

The parallel DDL statements for partitioned tables and indexes are:

- CREATE INDEX
- CREATE TABLE ... AS SELECT
- ALTER TABLE ... [MOVE|SPLIT|COALESCE] PARTITION
- ALTER INDEX ... [REBUILD|SPLIT] PARTITION
  - This statement can be executed in parallel only if the (global) index partition being split is usable.

All of these DDL operations can be performed in no-logging mode for either parallel or serial execution.

CREATE TABLE for an index-organized table can be parallelized either with or without an AS SELECT clause.

Different parallelism is used for different operations (see Table 24–3 on page 24-44). Parallel CREATE TABLE ... AS SELECT statements on partitioned tables and parallel CREATE INDEX statements on partitioned indexes execute with a DOP equal to the number of partitions.

Partition parallel analyze table is made less necessary by the ANALYZE {TABLE, INDEX} PARTITION statements, since parallel analyze of an entire partitioned table can be constructed with multiple user sessions.

Parallel DDL cannot occur on tables with object columns. Parallel DDL cannot occur on non-partitioned tables with LOB columns.

CREATE TABLE ... AS SELECT in Parallel

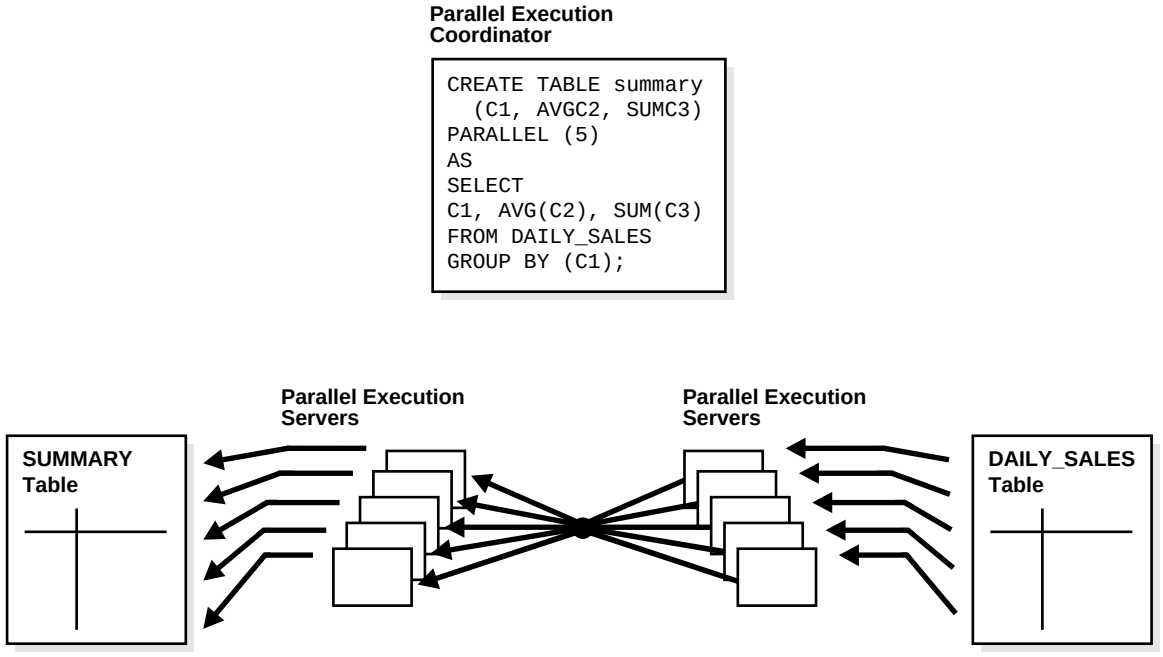
For performance reasons, decision support applications often require large amounts of data to be summarized or rolled up into smaller tables for use with ad hoc, decision support queries. Rollup occurs regularly (such as nightly or weekly) during a short period of system inactively.

Parallel execution lets you parallelize the query and create operations of creating a table as a subquery from another table or set of tables.

Clustered tables cannot be created and populated in parallel.

Figure 24–4 illustrates creating a table from a subquery in parallel.

Figure 24–4 Creating a Summary Table in Parallel



Recoverability and Parallel DDL

When summary table data is derived from other tables' data, recoverability from media failure for the smaller summary table may not be important and can be turned off during creation of the summary table.

If you disable logging during parallel table creation (or any other parallel DDL operation), you should back up the tablespace containing the table once the table is created to avoid loss of the table due to media failure.

Use the `NOLOGGING` clause of the `CREATE TABLE`, `CREATE INDEX`, `ALTER TABLE`, and `ALTER INDEX` statements to disable undo and redo log generation.

Space Management for Parallel DDL

Creating a table or index in parallel has space management implications that affect both the storage space required during a parallel operation and the free space available after a table or index has been created.

Storage Space When Using Dictionary-Managed Tablespaces

When creating a table or index in parallel, each parallel execution server uses the values in the STORAGE clause of the CREATE statement to create temporary segments to store the rows. Therefore, a table created with a NEXT setting of 5 MB and a PARALLEL DEGREE of 12 consumes at least 60 megabytes (MB) of storage during table creation because each process starts with an extent of 5 MB. When the parallel execution coordinator combines the segments, some of the segments may be trimmed, and the resulting table may be smaller than the requested 60 MB.

Free Space and Parallel DDL

When you create indexes and tables in parallel, each parallel execution server allocates a new extent and fills the extent with the table or index data. Thus, if you create an index with a DOP of 3, the index will have at least three extents initially. Allocation of extents is the same for rebuilding indexes in parallel and for moving, splitting, or rebuilding partitions in parallel.

Serial operations require the schema object to have at least one extent. Parallel creations require that tables or indexes have at least as many extents as there are parallel execution servers creating the schema object.

When you create a table or index in parallel, it is possible to create pockets of free space—either external or internal fragmentation. This occurs when the temporary segments used by the parallel execution servers are larger than what is needed to store the rows.

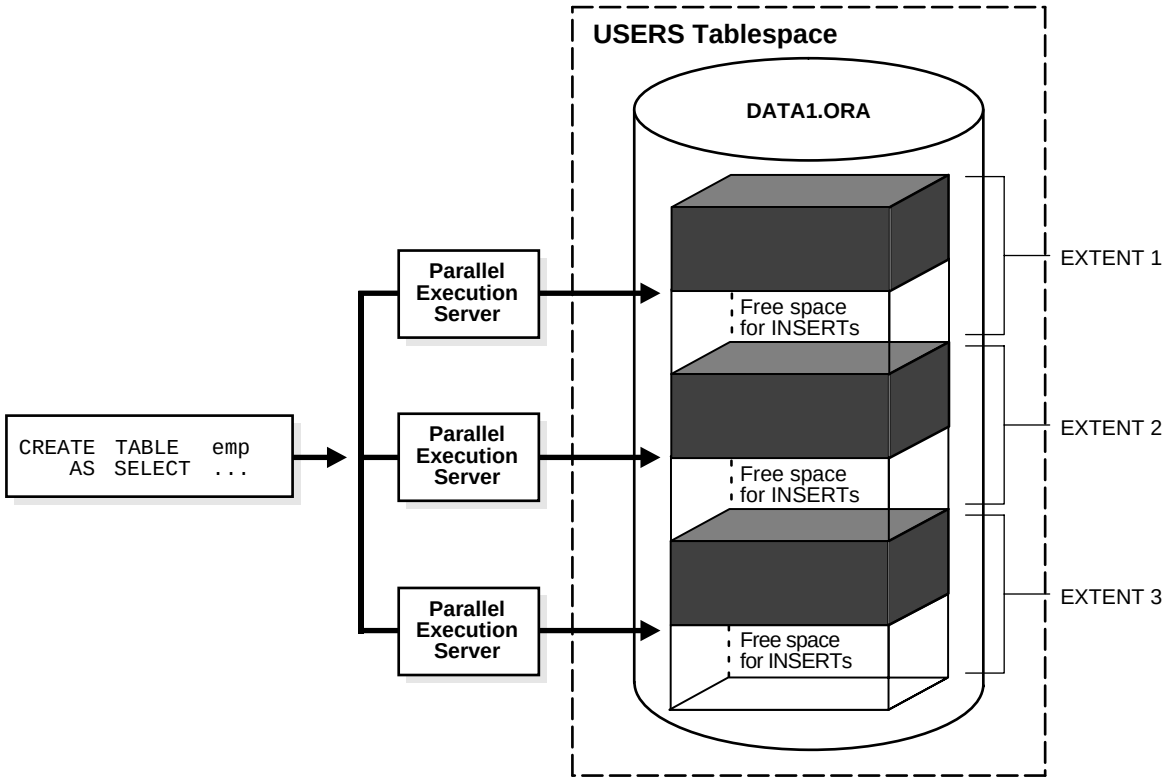
- If the unused space in each temporary segment is larger than the value of the MINIMUM EXTENT parameter set at the tablespace level, then Oracle trims the unused space when merging rows from all of the temporary segments into the table or index. The unused space is returned to the system free space and can be allocated for new extents, but it cannot be coalesced into a larger segment because it is not contiguous space (external fragmentation).
- If the unused space in each temporary segment is smaller than the value of the MINIMUM EXTENT parameter, then unused space cannot be trimmed when the rows in the temporary segments are merged. This unused space is not returned to the system free space; it becomes part of the table or index (internal fragmentation) and is available only for subsequent inserts or for updates that require additional space.

For example, if you specify a DOP of 3 for a CREATE TABLE ... AS SELECT statement, but there is only one datafile in the tablespace, then internal fragmentation may occur, as shown in Figure 24–5 on page 24-19. The pockets of

free space within the internal table extents of a datafile cannot be coalesced with other free space and cannot be allocated as extents.

See *Oracle Database Performance Tuning Guide* for more information about creating tables and indexes in parallel.

Figure 24–5 Unusable Free Space (Internal Fragmentation)



Parallel DML

Parallel DML (PARALLEL INSERT, UPDATE, DELETE, and MERGE) uses parallel execution mechanisms to speed up or scale up large DML operations against large database tables and indexes.

Note: Although DML generally includes queries, in this chapter the term DML refers only to inserts, updates, merges, and deletes.

This section discusses the following parallel DML topics:

- Advantages of Parallel DML over Manual Parallelism
- When to Use Parallel DML
- Enabling Parallel DML
- Transaction Restrictions for Parallel DML
- Rollback Segments
- Recovery for Parallel DML
- Space Considerations for Parallel DML
- Lock and Enqueue Resources for Parallel DML
- Restrictions on Parallel DML

Advantages of Parallel DML over Manual Parallelism

You can parallelize DML operations manually by issuing multiple DML statements simultaneously against different sets of data. For example, you can parallelize manually by:

- Issuing multiple `INSERT` statements to multiple instances of an Oracle Real Application Clusters to make use of free space from multiple free list blocks.
- Issuing multiple `UPDATE` and `DELETE` statements with different key value ranges or rowid ranges.

However, manual parallelism has the following disadvantages:

- It is difficult to use. You have to open multiple sessions (possibly on different instances) and issue multiple statements.
- There is a lack of transactional properties. The DML statements are issued at different times; and, as a result, the changes are done with inconsistent snapshots of the database. To get atomicity, the commit or rollback of the various statements must be coordinated manually (maybe across instances).
- The work division is complex. You may have to query the table in order to find out the rowid or key value ranges to correctly divide the work.

- The calculation is complex. The calculation of the degree of parallelism can be complex.
- There is a lack of affinity and resource information. You need to know affinity information to issue the right DML statement at the right instance when running an Oracle Real Application Clusters. You also have to find out about current resource usage to balance workload across instances.

Parallel DML removes these disadvantages by performing inserts, updates, and deletes in parallel automatically.

When to Use Parallel DML

Parallel DML operations are mainly used to speed up large DML operations against large database objects. Parallel DML is useful in a DSS environment where the performance and scalability of accessing large objects are important. Parallel DML complements parallel query in providing you with both querying and updating capabilities for your DSS databases.

The overhead of setting up parallelism makes parallel DML operations infeasible for short OLTP transactions. However, parallel DML operations can speed up batch jobs running in an OLTP database.

Some of the scenarios where parallel DML is used include:

- Refreshing Tables in a Data Warehouse System
- Creating Intermediate Summary Tables
- Using Scoring Tables
- Updating Historical Tables
- Running Batch Jobs

Refreshing Tables in a Data Warehouse System In a data warehouse system, large tables need to be refreshed (updated) periodically with new or modified data from the production system. You can do this efficiently by using parallel DML combined with updatable join views. You can also use the `MERGE` statement.

The data that needs to be refreshed is generally loaded into a temporary table before starting the refresh process. This table contains either new rows or rows that have been updated since the last refresh of the data warehouse. You can use an updatable join view with parallel `UPDATE` to refresh the updated rows, and you can use an anti-hash join with parallel `INSERT` to refresh the new rows.

Creating Intermediate Summary Tables In a DSS environment, many applications require complex computations that involve constructing and manipulating many large intermediate summary tables. These summary tables are often temporary and frequently do not need to be logged. Parallel DML can speed up the operations against these large intermediate tables. One benefit is that you can put incremental results in the intermediate tables and perform parallel update.

In addition, the summary tables may contain cumulative or comparison information which has to persist beyond application sessions; thus, temporary tables are not feasible. Parallel DML operations can speed up the changes to these large summary tables.

Using Scoring Tables Many DSS applications score customers periodically based on a set of criteria. The scores are usually stored in large DSS tables. The score information is then used in making a decision, for example, inclusion in a mailing list.

This scoring activity queries and updates a large number of rows in the large table. Parallel DML can speed up the operations against these large tables.

Updating Historical Tables Historical tables describe the business transactions of an enterprise over a recent time interval. Periodically, the DBA deletes the set of oldest rows and inserts a set of new rows into the table. Parallel `INSERT ... SELECT` and parallel `DELETE` operations can speed up this rollover task.

Although you can also use parallel direct loader (SQL*Loader) to insert bulk data from an external source, parallel `INSERT ... SELECT` is faster for inserting data that already exists in another table in the database.

Dropping a partition can also be used to delete old rows. However, to do this, the table has to be partitioned by date and with the appropriate time interval.

Running Batch Jobs Batch jobs executed in an OLTP database during off hours have a fixed time window in which the jobs must complete. A good way to ensure timely job completion is to parallelize their operations. As the work load increases, more machine resources can be added; the scaleup property of parallel operations ensures that the time constraint can be met.

Enabling Parallel DML

A DML statement can be parallelized only if you have explicitly enabled parallel DML in the session with the `ENABLE PARALLEL DML` clause of the `ALTER SESSION` statement. This mode is required because parallel DML and serial DML have different locking, transaction, and disk space requirements.

The default mode of a session is DISABLE PARALLEL DML. When parallel DML is disabled, no DML will be executed in parallel even if the PARALLEL hint is used.

When parallel DML is enabled in a session, all DML statements in this session will be considered for parallel execution. However, even if parallel DML is enabled, the DML operation may still execute serially if there are no parallel hints or no tables with a parallel attribute or if restrictions on parallel operations are violated.

The session's PARALLEL DML mode does not influence the parallelism of SELECT statements, DDL statements, and the query portions of DML statements. Thus, if this mode is not set, the DML operation is not parallelized, but scans or join operations within the DML statement may still be parallelized.

See Also:

- "Space Considerations for Parallel DML" on page 24-24
- "Lock and Enqueue Resources for Parallel DML" on page 24-25
- "Restrictions on Parallel DML" on page 24-25

Transaction Restrictions for Parallel DML

To execute a DML operation in parallel, the parallel execution coordinator acquires or spawns parallel execution servers, and each parallel execution server executes a portion of the work under its own parallel process transaction.

- Each parallel execution server creates a different parallel process transaction.
- If you use rollback segments instead of Automatic Undo Management, you may want to reduce contention on the rollback segments. To do this, only a few parallel process transactions should reside in the same rollback segment. See *Oracle Database SQL Reference* for more information.

The coordinator also has its own coordinator transaction, which can have its own rollback segment. In order to ensure user-level transactional atomicity, the coordinator uses a two-phase commit protocol to commit the changes performed by the parallel process transactions.

A session that is enabled for parallel DML may put transactions in the session in a special mode: If any DML statement in a transaction modifies a table in parallel, no subsequent serial or parallel query or DML statement can access the same table again in that transaction. This means that the results of parallel modifications cannot be seen during the transaction.

Serial or parallel statements that attempt to access a table that has already been modified in parallel within the same transaction are rejected with an error message.

If a PL/SQL procedure or block is executed in a parallel DML enabled session, then this rule applies to statements in the procedure or block.

Rollback Segments

If you use rollback segments instead of Automatic Undo Management, there are some restrictions when using parallel DML. See *Oracle Database SQL Reference* for information about restrictions for parallel DML and rollback segments.

Recovery for Parallel DML

The time required to roll back a parallel DML operation is roughly equal to the time it takes to perform the forward operation.

Oracle supports parallel rollback after transaction and process failures, and after instance and system failures. Oracle can parallelize both the rolling forward stage and the rolling back stage of transaction recovery.

See *Oracle Database Backup and Recovery Basics* for details about parallel rollback.

Transaction Recovery for User-Issued Rollback A user-issued rollback in a transaction failure due to statement error is performed in parallel by the parallel execution coordinator and the parallel execution servers. The rollback takes approximately the same amount of time as the forward transaction.

Process Recovery Recovery from the failure of a parallel execution coordinator or parallel execution server is performed by the PMON process. If a parallel execution server or a parallel execution coordinator fails, PMON rolls back the work from that process and all other processes in the transaction roll back their changes.

System Recovery Recovery from a system failure requires a new startup. Recovery is performed by the SMON process and any recovery server processes spawned by SMON. Parallel DML statements may be recovered using parallel rollback. If the initialization parameter `COMPATIBLE` is set to 8.1.3 or greater, Fast-Start On-Demand Rollback enables terminated transactions to be recovered, on demand one block at a time.

Space Considerations for Parallel DML

Parallel `UPDATE` uses the space in the existing object, while direct-path `INSERT` gets new segments for the data.

Space usage characteristics may be different in parallel than sequential execution because multiple concurrent child transactions modify the object.

Lock and Enqueue Resources for Parallel DML

A parallel DML operation's lock and enqueue resource requirements are very different from the serial DML requirements. Parallel DML holds many more locks, so you should increase the starting value of the ENQUEUE_RESOURCES and DML_LOCKS parameters. See "DML_LOCKS" on page 24-58 for more information.

Restrictions on Parallel DML

The following restrictions apply to parallel DML (including direct-path INSERT):

- Intra-partition parallelism for UPDATE, MERGE, and DELETE operations require that the COMPATIBLE initialization parameter be set to 9.2 or greater.
- INSERT, UPDATE, MERGE, and DELETE operations on nonpartitioned tables are not parallelized if there is a bitmap index on the table. If the table is partitioned and there is a bitmap index on the table, the degree of parallelism will be restricted to at most the number of partitions accessed.
- A transaction can contain multiple parallel DML statements that modify different tables, but after a parallel DML statement modifies a table, no subsequent serial or parallel statement (DML or query) can access the same table again in that transaction.
  - This restriction also exists after a serial direct-path INSERT statement: no subsequent SQL statement (DML or query) can access the modified table during that transaction.
  - Queries that access the same table are allowed before a parallel DML or direct-path INSERT statement, but not after.
  - Any serial or parallel statements attempting to access a table that has already been modified by a parallel UPDATE, DELETE, or MERGE, or a direct-path INSERT during the same transaction are rejected with an error message.
- Parallel DML operations cannot be done on tables with triggers.
- Replication functionality is not supported for parallel DML.
- Parallel DML cannot occur in the presence of certain constraints: self-referential integrity, delete cascade, and deferred integrity. In addition, for direct-path INSERT, there is no support for any referential integrity.
- Parallel DML can be done on tables with object columns provided you are not touching the object columns.

- Parallel DML can be done on tables with LOB columns provided the table is partitioned. However, intra-partition parallelism is not supported.
- A transaction involved in a parallel DML operation cannot be or become a distributed transaction.
- Clustered tables are not supported.

Violations of these restrictions cause the statement to execute serially without warnings or error messages (except for the restriction on statements accessing the same table in a transaction, which can cause error messages). For example, an update is serialized if it is on a nonpartitioned table.

Partitioning Key Restriction You can only update the partitioning key of a partitioned table to a new value if the update does not cause the row to move to a new partition. The update is possible if the table is defined with the row movement clause enabled.

Function Restrictions The function restrictions for parallel DML are the same as those for parallel DDL and parallel query. See "Parallel Execution of Functions" on page 24-28 for more information.

Data Integrity Restrictions

This section describes the interactions of integrity constraints and parallel DML statements.

NOT NULL and CHECK These types of integrity constraints are allowed. They are not a problem for parallel DML because they are enforced on the column and row level, respectively.

UNIQUE and PRIMARY KEY These types of integrity constraints are allowed.

FOREIGN KEY (Referential Integrity) Restrictions for referential integrity occur whenever a DML operation on one table could cause a recursive DML operation on another table. These restrictions also apply when, in order to perform an integrity check, it is necessary to see simultaneously all changes made to the object being modified.

Table 24–1 lists all of the operations that are possible on tables that are involved in referential integrity constraints.

Table 24–1 Referential Integrity Restrictions

DML Statement	Issued on Parent	Issued on Child	Self-Referential
INSERT	(Not applicable)	Not parallelized	Not parallelized
MERGE	(Not applicable)	Not parallelized	Not parallelized
UPDATE No Action	Supported	Supported	Not parallelized
DELETE No Action	Supported	Supported	Not parallelized
DELETE Cascade	Not parallelized	(Not applicable)	Not parallelized

Delete Cascade Delete on tables having a foreign key with delete cascade is not parallelized because parallel execution servers will try to delete rows from multiple partitions (parent and child tables).

Self-Referential Integrity DML on tables with self-referential integrity constraints is not parallelized if the referenced keys (primary keys) are involved. For DML on all other columns, parallelism is possible.

Deferrable Integrity Constraints If any deferrable constraints apply to the table being operated on, the DML operation will not be parallelized.

Trigger Restrictions

A DML operation will not be parallelized if the affected tables contain enabled triggers that may get fired as a result of the statement. This implies that DML statements on tables that are being replicated will not be parallelized.

Relevant triggers must be disabled in order to parallelize DML on the table. Note that, if you enable or disable triggers, the dependent shared cursors are invalidated.

Distributed Transaction Restrictions

A DML operation cannot be parallelized if it is in a distributed transaction or if the DML or the query operation is against a remote object.

Examples of Distributed Transaction Parallelization

This section contains several examples of distributed transaction processing.

Example 24–1 Distributed Transaction Parallelization

In this example, the DML statement queries a remote object:

```
INSERT /* APPEND PARALLEL (t3,2) */ INTO t3 SELECT * FROM t4@dblink;
```

The query operation is executed serially without notification because it references a remote object.

Example 24–2 Distributed Transaction Parallelization

In this example, the DML operation is applied to a remote object:

```
DELETE /*+ PARALLEL (t1, 2) */ FROM t1@dblink;
```

The DELETE operation is not parallelized because it references a remote object.

Example 24–3 Distributed Transaction Parallelization

In this example, the DML operation is in a distributed transaction:

```
SELECT * FROM t1@dblink;
DELETE /*+ PARALLEL (t2,2) */ FROM t2;
COMMIT;
```

The DELETE operation is not parallelized because it occurs in a distributed transaction (which is started by the SELECT statement).

Parallel Execution of Functions

SQL statements can contain user-defined functions written in PL/SQL, in Java, or as external procedures in C that can appear as part of the SELECT list, SET clause, or WHERE clause. When the SQL statement is parallelized, these functions are executed on a per-row basis by the parallel execution server. Any PL/SQL package variables or Java static attributes used by the function are entirely private to each individual parallel execution process and are newly initialized when each row is processed, rather than being copied from the original session. Because of this, not all functions will generate correct results if executed in parallel.

User-written table functions can appear in the statement's FROM list. These functions act like source tables in that they output rows. Table functions are initialized once during the statement at the start of each parallel execution process. All variables are entirely private to the parallel execution process.

Functions in Parallel Queries

In a SELECT statement or a subquery in a DML or DDL statement, a user-written function may be executed in parallel if it has been declared with the PARALLEL_ENABLE keyword, if it is declared in a package or type and has a PRAGMA RESTRICT_REFERENCES that indicates all of WNDS, RNPS, and WNPS, or if it is declared with CREATE FUNCTION and the system can analyze the body of the PL/SQL code and determine that the code neither writes to the database nor reads or modifies package variables.

Other parts of a query or subquery can sometimes execute in parallel even if a given function execution must remain serial.

See *Oracle Database Application Developer's Guide - Fundamentals* for information about the PRAGMA RESTRICT_REFERENCES and *Oracle Database SQL Reference* for information about CREATE FUNCTION.

Functions in Parallel DML and DDL Statements

In a parallel DML or DDL statement, as in a parallel query, a user-written function may be executed in parallel if it has been declared with the PARALLEL_ENABLE keyword, if it is declared in a package or type and has a PRAGMA RESTRICT_REFERENCES that indicates all of RNDS, WNDS, RNPS, and WNPS, or if it is declared with CREATE FUNCTION and the system can analyze the body of the PL/SQL code and determine that the code neither reads nor writes to the database or reads nor modifies package variables.

For a parallel DML statement, any function call that cannot be executed in parallel causes the entire DML statement to be executed serially.

For an INSERT ... SELECT or CREATE TABLE ... AS SELECT statement, function calls in the query portion are parallelized according to the parallel query rules in the prior paragraph. The query may be parallelized even if the remainder of the statement must execute serially, or vice versa.

Other Types of Parallelism

In addition to parallel SQL execution, Oracle can use parallelism for the following types of operations:

- Parallel recovery
- Parallel propagation (replication)
- Parallel load (the SQL*Loader utility)

Like parallel SQL, parallel recovery and propagation are performed by a parallel execution coordinator and multiple parallel execution servers. Parallel load, however, uses a different mechanism.

The behavior of the parallel execution coordinator and parallel execution servers may differ, depending on what kind of operation they perform (SQL, recovery, or propagation). For example, if all parallel execution servers in the pool are occupied and the maximum number of parallel execution servers has been started:

- In parallel SQL, the parallel execution coordinator switches to serial processing.
- In parallel propagation, the parallel execution coordinator returns an error.

For a given session, the parallel execution coordinator coordinates only one kind of operation. A parallel execution coordinator cannot coordinate, for example, parallel SQL and parallel recovery or propagation at the same time.

See Also:

- *Oracle Database Utilities* for information about parallel load and SQL*Loader
- *Oracle Database Backup and Recovery Basics* for information about parallel media recovery
- *Oracle Database Performance Tuning Guide* for information about parallel instance recovery
- *Oracle Database Advanced Replication* for information about parallel propagation

Initializing and Tuning Parameters for Parallel Execution

Parallel execution is enabled by default. The initial computed values of the parallel execution parameters should be acceptable for the majority of installations. These parameters affect memory usage and the degree of parallelism used for parallel operations.

Oracle Database computes defaults for these parameters based on the value at database startup of CPU_COUNT and PARALLEL_THREADS_PER_CPU. The parameters can also be manually tuned, increasing or decreasing their values to suit specific system configurations or performance goals.

For example:

- On systems where parallel execution will never be used, PARALLEL_MAX_SERVERS can be set to zero.

- On large systems with abundant SGA memory, PARALLEL_EXECUTION_MESSAGE_SIZE can be increased to improve throughput.

You can also manually tune parallel execution parameters; however, Oracle recommends using default settings for parallel execution. Manual tuning of parallel execution is more complex than using default settings for two reasons: manual parallel execution tuning requires more attentive administration than automated tuning, and manual tuning is prone to user-load and system-resource miscalculations.

Initializing and tuning parallel execution involves the following steps:

- Using Default Parameter Settings
- Setting the Degree of Parallelism for Parallel Execution
- How Oracle Determines the Degree of Parallelism for Operations
- Balancing the Workload
- Parallelization Rules for SQL Statements
- Enabling Parallelism for Tables and Queries
- Degree of Parallelism and Adaptive Multiuser: How They Interact
- Forcing Parallel Execution for a Session
- Controlling Performance with the Degree of Parallelism

Using Default Parameter Settings

By default, Oracle automatically sets parallel execution parameters, as shown in Table 24–2. For most systems, you do not need to make further adjustments to have an adequately tuned parallel execution environment.

Table 24–2 Parameters and Their Defaults

Parameter	Default	Comments
PARALLEL_ADAPTIVE_MULTI_USER	TRUE	Causes parallel execution SQL to throttle DOP requests to prevent system overload.
PARALLEL_MAX_SERVERS	CPU_COUNT x PARALLEL_THREADS_PRE_CPU x (2 if PGA_AGGREGATE_TARGET > 0; otherwise 1) x 5	Use this limit to maximize the number of processes that parallel execution uses.
PARALLEL_EXECUTION_MESSAGE_SIZE	2 KB (port specific)	Increase to 4k or 8k to improve parallel execution performance if sufficient SGA memory exists.

Note that you can set some parameters in such a way that Oracle will be constrained. For example, if you set PROCESSES to 20, you will not be able to get 25 slaves.

Setting the Degree of Parallelism for Parallel Execution

The parallel execution coordinator may enlist two or more of the instance's parallel execution servers to process a SQL statement. The number of parallel execution servers associated with a single operation is known as the degree of parallelism.

The DOP is specified in the following ways:

- At the statement level with hints and with the PARALLEL clause
- At the session level by issuing the ALTER SESSION FORCE PARALLEL statement
- At the table level in the table's definition
- At the index level in the index's definition

The following example shows a statement that sets the DOP to 4 on a table:

```
ALTER TABLE orders PARALLEL 4;
```

This next example sets the DOP on an index to 4:

```
ALTER INDEX iorders PARALLEL 4;
```

This last example sets a hint to 4 on a query:

```
SELECT /*+ PARALLEL(orders, 4) */ COUNT(*) FROM orders;
```

Note that the DOP applies directly only to intraoperation parallelism. If interoperation parallelism is possible, the total number of parallel execution servers for a statement can be twice the specified DOP. No more than two operations can be performed simultaneously.

Parallel execution is designed to effectively use multiple CPUs and disks to answer queries quickly. When multiple users employ parallel execution at the same time, available CPU, memory, and disk resources may be quickly exhausted. Oracle provides several ways to deal with resource utilization in conjunction with parallel execution, including:

- The adaptive multiuser algorithm, which reduces the DOP as the load on the system increases. By default, the adaptive multiuser algorithm is enabled,

which optimizes the performance of systems with concurrent parallel SQL execution operations.

- User resource limits and profiles, which allow you to set limits on the amount of various system resources available to each user as part of a user's security domain.
- The Database Resource Manager, which enables you to allocate resources to different groups of users.

How Oracle Determines the Degree of Parallelism for Operations

The parallel execution coordinator determines the DOP by considering several specifications. The coordinator:

- Checks for hints or a PARALLEL clause specified in the SQL statement itself.
- Checks for a session value set by the ALTER SESSION FORCE PARALLEL statement.
- Looks at the table's or index's definition.

After a DOP is found in one of these specifications, it becomes the DOP for the operation.

Hints, PARALLEL clauses, table or index definitions, and default values only determine the number of parallel execution servers that the coordinator requests for a given operation. The actual number of parallel execution servers used depends upon how many processes are available in the parallel execution server pool and whether interoperation parallelism is possible.

See Also:

- "The Parallel Execution Server Pool" on page 24-6
- "Parallelism Between Operations" on page 24-10
- "Default Degree of Parallelism" on page 24-34
- "Parallelization Rules for SQL Statements" on page 24-37

Hints and Degree of Parallelism

You can specify hints in a SQL statement to set the DOP for a table or index and for the caching behavior of the operation.

- The PARALLEL hint is used only for operations on tables. You can use it to parallelize queries and DML statements (INSERT, UPDATE, MERGE, and DELETE).
- The PARALLEL_INDEX hint parallelizes an index range scan of a partitioned index. (In an index operation, the PARALLEL hint is not valid and is ignored.)

See *Oracle Database Performance Tuning Guide* for information about using hints in SQL statements and the specific syntax for the PARALLEL, NO_PARALLEL, PARALLEL_INDEX, CACHE, and NOCACHE hints.

Table and Index Definitions

You can specify the DOP within a table or index definition by using one of the following statements: CREATE TABLE, ALTER TABLE, CREATE INDEX, or ALTER INDEX.

Default Degree of Parallelism

The default DOP is used when you ask to parallelize an operation but you do not specify a DOP in a hint or within the definition of a table or index. The default DOP is appropriate for most applications.

The default DOP for a SQL statement is determined by the following factors:

- The value of the parameter CPU_COUNT, which is, by default, the number of CPUs on the system, the number of RAC instances, and the value of the parameter PARALLEL_THREADS_PER_CPU.
- For parallelizing by partition, the number of partitions that will be accessed, based on partition pruning.
- For parallel DML operations with global index maintenance, the minimum number of transaction free lists among all the global indexes to be updated. The minimum number of transaction free lists for a partitioned global index is the minimum number across all index partitions. This is a requirement to prevent self-deadlock.

These factors determine the default number of parallel execution servers to use. However, the actual number of processes used is limited by their availability on the requested instances during run time. The initialization parameter PARALLEL_MAX_SERVERS sets an upper limit on the total number of parallel execution servers that an instance can have.

If a minimum fraction of the desired parallel execution servers is not available (specified by the initialization parameter PARALLEL_MIN_PERCENT), a user error is produced. You can retry the query when the system is less busy.

Adaptive Multiuser Algorithm

With the adaptive multiuser algorithm, the parallel execution coordinator varies the DOP according to the system load. Oracle determines the load by calculating the number of active Oracle Server processes. If the number of server processes currently allocated is larger than the optimal number of server processes, given the number of available CPUs, the algorithm reduces the DOP. This reduction improves total system throughput by avoiding overallocation of resources.

Minimum Number of Parallel Execution Servers

Oracle can perform an operation in parallel as long as at least two parallel execution servers are available. If too few parallel execution servers are available, your SQL statement may execute slower than expected. You can specify the minimum percentage of requested parallel execution servers that must be available in order for the operation to execute. This strategy ensures that your SQL statement executes with a minimum acceptable parallel performance. If the minimum percentage of requested parallel execution servers is not available, the SQL statement does not execute and returns an error ora 12827.

The initialization parameter PARALLEL_MIN_PERCENT specifies the desired minimum percentage of requested parallel execution servers. This parameter affects DML and DDL operations as well as queries.

For example, if you specify 50 for this parameter, then at least 50 percent of the parallel execution servers requested for any parallel operation must be available in order for the operation to succeed. If 20 parallel execution servers are requested, then at least 10 must be available or an error is returned to the user. If PARALLEL_MIN_PERCENT is set to null, then all parallel operations will proceed as long as at least two parallel execution servers are available for processing.

Limiting the Number of Available Instances

In Oracle Real Application Clusters, instance groups can be used to limit the number of instances that participate in a parallel operation. You can create any number of instance groups, each consisting of one or more instances. You can then specify which instance group is to be used for any or all parallel operations. Parallel execution servers will only be used on instances which are members of the specified instance group. See *Oracle Real Application Clusters Administrator's Guide* and *Oracle*

Real Application Clusters Deployment and Performance Guide for more information about instance groups.

Balancing the Workload

To optimize performance, all parallel execution servers should have equal workloads. For SQL statements parallelized by block range or by parallel execution servers, the workload is dynamically divided among the parallel execution servers. This minimizes workload skewing, which occurs when some parallel execution servers perform significantly more work than the other processes.

For the relatively few SQL statements parallelized by partitions, if the workload is evenly distributed among the partitions, you can optimize performance by matching the number of parallel execution servers to the number of partitions or by choosing a DOP in which the number of partitions is a multiple of the number of processes. This applies to partition-wise joins and PDML on tables created before Oracle9*i*. See "Limitation on the Degree of Parallelism" on page 24-79 for details regarding this topic.

For example, suppose a table has 10 partition, and a parallel operation divides the work evenly among them. You can use 10 parallel execution servers (DOP equals 10) to do the work in approximately one-tenth the time that one process would take. You might also use five processes to do the work in one-fifth the time, or two processes to do the work in one-half the time.

If, however, you use nine processes to work on 10 partitions, the first process to finish its work on one partition then begins work on the 10th partition; and as the other processes finish their work, they become idle. This configuration does not provide good performance when the work is evenly divided among partitions. When the work is unevenly divided, the performance varies depending on whether the partition that is left for last has more or less work than the other partitions.

Similarly, suppose you use four processes to work on 10 partitions and the work is evenly divided. In this case, each process works on a second partition after finishing its first partition, but only two of the processes work on a third partition while the other two remain idle.

In general, you cannot assume that the time taken to perform a parallel operation on a given number of partitions (N) with a given number of parallel execution servers (P) will be N / P. This formula does not take into account the possibility that some processes might have to wait while others finish working on the last partitions. By choosing an appropriate DOP, however, you can minimize the workload skew and optimize performance.

Parallelization Rules for SQL Statements

A SQL statement can be parallelized if it includes a parallel hint or if the table or index being operated on has been declared PARALLEL with a CREATE or ALTER statement. In addition, a DDL statement can be parallelized by using the PARALLEL clause. However, not all of these methods apply to all types of SQL statements.

Parallelization has two components: the decision to parallelize and the DOP. These components are determined differently for queries, DDL operations, and DML operations.

To determine the DOP, Oracle looks at the reference objects:

- Parallel query looks at each table and index, in the portion of the query being parallelized, to determine which is the reference table. The basic rule is to pick the table or index with the largest DOP.
- For parallel DML (INSERT, UPDATE, MERGE, and DELETE), the reference object that determines the DOP is the table being modified by an insert, update, or delete operation. Parallel DML also adds some limits to the DOP to prevent deadlock. If the parallel DML statement includes a subquery, the subquery's DOP is the same as the DML operation.
- For parallel DDL, the reference object that determines the DOP is the table, index, or partition being created, rebuilt, split, or moved. If the parallel DDL statement includes a subquery, the subquery's DOP is the same as the DDL operation.

Rules for Parallelizing Queries

This section discusses some rules for parallelizing queries.

Decision to Parallelize A SELECT statement can be parallelized only if the following conditions are satisfied:

- The query includes a parallel hint specification (PARALLEL or PARALLEL_INDEX) or the schema objects referred to in the query have a PARALLEL declaration associated with them.
- At least one of the tables specified in the query requires one of the following:
  - A full table scan
  - An index range scan spanning multiple partitions
- No scalar subqueries are in the SELECT list

Degree of Parallelism The DOP for a query is determined by the following rules:

- The query uses the maximum DOP taken from all of the table declarations involved in the query and all of the potential indexes that are candidates to satisfy the query (the reference objects). That is, the table or index that has the greatest DOP determines the query's DOP (maximum query directive).
- If a table has both a parallel hint specification in the query and a parallel declaration in its table specification, the hint specification takes precedence over parallel declaration specification. See Table 24–3 on page 24-44 for precedence rules.

Rules for UPDATE, MERGE, and DELETE

UPDATE, MERGE, and DELETE operations are parallelized by partition or subpartition. Update, merge, and delete parallelism are not possible within a partition, nor on a nonpartitioned table. See "Limitation on the Degree of Parallelism" on page 24-79 for a possible restriction.

You have two ways to specify parallel directives for UPDATE, MERGE, and DELETE operations (assuming that PARALLEL DML mode is enabled):

- Use a parallel clause in the definition of the table being updated or deleted (the reference object).
- Use an update, merge, or delete parallel hint in the statement.

Parallel hints are placed immediately after the UPDATE, MERGE, or DELETE keywords in UPDATE, MERGE, and DELETE statements. The hint also applies to the underlying scan of the table being changed.

You can use the ALTER SESSION FORCE PARALLEL DML statement to override parallel clauses for subsequent UPDATE, MERGE, and DELETE statements in a session. Parallel hints in UPDATE, MERGE, and DELETE statements override the ALTER SESSION FORCE PARALLEL DML statement.

Decision to Parallelize The following rule determines whether the UPDATE, MERGE, or DELETE operation should be parallelized:

The UPDATE or DELETE operation will be parallelized if and only if at least one of the following is true:

- The table being updated or deleted has a PARALLEL specification.
- The PARALLEL hint is specified in the DML statement.

- An ALTER SESSION FORCE PARALLEL DML statement has been issued previously during the session.

If the statement contains subqueries or updatable views, then they may have their own separate parallel hints or clauses. However, these parallel directives do not affect the decision to parallelize the UPDATE, MERGE, or DELETE.

The parallel hint or clause on the tables is used by both the query and the UPDATE, MERGE, DELETE portions to determine parallelism, the decision to parallelize the UPDATE, MERGE, or DELETE portion is made independently of the query portion, and vice versa.

Degree of Parallelism The DOP is determined by the same rules as for the queries. Note that in the case of UPDATE and DELETE operations, only the target table to be modified (the only reference object) is involved. Thus, the UPDATE or DELETE parallel hint specification takes precedence over the parallel declaration specification of the target table. In other words, the precedence order is: MERGE, UPDATE, DELETE hint > Session > Parallel declaration specification of target table. See Table 24–3 on page 24-44 for precedence rules.

A parallel execution server can update or merge into, or delete from multiple partitions, but each partition can only be updated or deleted by one parallel execution server.

If the DOP is less than the number of partitions, then the first process to finish work on one partition continues working on another partition, and so on until the work is finished on all partitions. If the DOP is greater than the number of partitions involved in the operation, then the excess parallel execution servers will have no work to do.

Example 24–4 Parallelization: Example 1

```
UPDATE tbl_1 SET c1=c1+1 WHERE c1>100;
```

If tbl_1 is a partitioned table and its table definition has a parallel clause, then the update operation is parallelized even if the scan on the table is serial (such as an index scan), assuming that the table has more than one partition with c1 greater than 100.

Example 24–5 Parallelization: Example 2

```
UPDATE /*+ PARALLEL(tbl_2,4) */ tbl_2 SET c1=c1+1;
```

Both the scan and update operations on tbl_2 will be parallelized with degree four.

Rules for INSERT ... SELECT

An INSERT ... SELECT statement parallelizes its INSERT and SELECT operations independently, except for the DOP.

You can specify a parallel hint after the INSERT keyword in an INSERT ... SELECT statement. Because the tables being queried are usually not the same as the table being inserted into, the hint enables you to specify parallel directives specifically for the insert operation.

You have the following ways to specify parallel directives for an INSERT ... SELECT statement (assuming that PARALLEL DML mode is enabled):

- SELECT parallel hints specified at the statement
- Parallel clauses specified in the definition of tables being selected
- INSERT parallel hint specified at the statement
- Parallel clause specified in the definition of tables being inserted into

You can use the ALTER SESSION FORCE PARALLEL DML statement to override parallel clauses for subsequent INSERT operations in a session. Parallel hints in insert operations override the ALTER SESSION FORCE PARALLEL DML statement.

Decision to Parallelize The following rule determines whether the INSERT operation should be parallelized in an INSERT ... SELECT statement:

The INSERT operation will be parallelized if and only if at least one of the following is true:

- The PARALLEL hint is specified after the INSERT in the DML statement.
- The table being inserted into (the reference object) has a PARALLEL declaration specification.
- An ALTER SESSION FORCE PARALLEL DML statement has been issued previously during the session.

The decision to parallelize the INSERT operation is made independently of the SELECT operation, and vice versa.

Degree of Parallelism Once the decision to parallelize the SELECT or INSERT operation is made, one parallel directive is picked for deciding the DOP of the whole statement, using the following precedence rule Insert hint directive >

Session> Parallel declaration specification of the inserting table > Maximum query directive.

In this context, maximum query directive means that among multiple tables and indexes, the table or index that has the maximum DOP determines the parallelism for the query operation.

The chosen parallel directive is applied to both the SELECT and INSERT operations.

Example 24–6 Parallelization: Example 3

The DOP used is 2, as specified in the INSERT hint:

```
INSERT /*+ PARALLEL(tbl_ins,2) */ INTO tbl_ins
SELECT /*+ PARALLEL(tbl_sel,4) */ * FROM tbl_sel;
```

Rules for DDL Statements

You need to keep the following in mind when parallelizing DDL statements.

Decision to Parallelize DDL operations can be parallelized if a PARALLEL clause (declaration) is specified in the syntax. In the case of CREATE INDEX and ALTER INDEX ... REBUILD or ALTER INDEX ... REBUILD PARTITION, the parallel declaration is stored in the data dictionary.

You can use the ALTER SESSION FORCE PARALLEL DDL statement to override the parallel clauses of subsequent DDL statements in a session.

Degree of Parallelism The DOP is determined by the specification in the PARALLEL clause, unless it is overridden by an ALTER SESSION FORCE PARALLEL DDL statement. A rebuild of a partitioned index is never parallelized.

Parallel clauses in CREATE TABLE and ALTER TABLE statements specify table parallelism. If a parallel clause exists in a table definition, it determines the parallelism of DDL statements as well as queries. If the DDL statement contains explicit parallel hints for a table, however, those hints override the effect of parallel clauses for that table. You can use the ALTER SESSION FORCE PARALLEL DDL statement to override parallel clauses.

Rules for [CREATE | REBUILD] INDEX or [MOVE | SPLIT] PARTITION

The following rules apply:

Parallel CREATE INDEX or ALTER INDEX ... REBUILD The CREATE INDEX and ALTER INDEX ... REBUILD statements can be parallelized only by a PARALLEL clause or an ALTER SESSION FORCE PARALLEL DDL statement.

ALTER INDEX ... REBUILD can be parallelized only for a nonpartitioned index, but ALTER INDEX ... REBUILD PARTITION can be parallelized by a PARALLEL clause or an ALTER SESSION FORCE PARALLEL DDL statement.

The scan operation for ALTER INDEX ... REBUILD (nonpartitioned), ALTER INDEX ... REBUILD PARTITION, and CREATE INDEX has the same parallelism as the REBUILD or CREATE operation and uses the same DOP. If the DOP is not specified for REBUILD or CREATE, the default is the number of CPUs.

Parallel MOVE PARTITION or SPLIT PARTITION The ALTERINDEX ... MOVE PARTITION and ALTERINDEX ...SPLIT PARTITION statements can be parallelized only by a PARALLEL clause or an ALTER SESSION FORCE PARALLEL DDL statement. Their scan operations have the same parallelism as the corresponding MOVE or SPLIT operations. If the DOP is not specified, the default is the number of CPUs.

Rules for CREATE TABLE AS SELECT

The CREATE TABLE ... AS SELECT statement contains two parts: a CREATE part (DDL) and a SELECT part (query). Oracle can parallelize both parts of the statement. The CREATE part follows the same rules as other DDL operations.

Decision to Parallelize (Query Part) The query part of a CREATE TABLE ... AS SELECT statement can be parallelized only if the following conditions are satisfied:

- The query includes a parallel hint specification (PARALLEL or PARALLEL_INDEX) or the CREATE part of the statement has a PARALLEL clause specification or the schema objects referred to in the query have a PARALLEL declaration associated with them.
- At least one of the tables specified in the query requires one of the following: a full table scan or an index range scan spanning multiple partitions.

Degree of Parallelism (Query Part) The DOP for the query part of a CREATE TABLE ... AS SELECT statement is determined by one of the following rules:

- The query part uses the values specified in the PARALLEL clause of the CREATE part.
- If the PARALLEL clause is not specified, the default DOP is the number of CPUs.
- If the CREATE is serial, then the DOP is determined by the query.

Note that any values specified in a hint for parallelism are ignored.

Decision to Parallelize (CREATE Part) The CREATE operation of CREATE TABLE ... AS SELECT can be parallelized only by a PARALLEL clause or an ALTER SESSION FORCE PARALLEL DDL statement.

When the CREATE operation of CREATE TABLE ... AS SELECT is parallelized, Oracle also parallelizes the scan operation if possible. The scan operation cannot be parallelized if, for example:

- The SELECT clause has a NO_PARALLEL hint
- The operation scans an index of a nonpartitioned table

When the CREATE operation is not parallelized, the SELECT can be parallelized if it has a PARALLEL hint or if the selected table (or partitioned index) has a parallel declaration.

Degree of Parallelism (CREATE Part) The DOP for the CREATE operation, and for the SELECT operation if it is parallelized, is specified by the PARALLEL clause of the CREATE statement, unless it is overridden by an ALTER SESSION FORCE PARALLEL DDL statement. If the PARALLEL clause does not specify the DOP, the default is the number of CPUs.

Summary of Parallelization Rules

Table 24–3 shows how various types of SQL statements can be parallelized and indicates which methods of specifying parallelism take precedence.

- The priority (1) specification overrides priority (2) and priority (3).
- The priority (2) specification overrides priority (3).

Table 24–3 Parallelization Rules

Parallel Operation	Parallelized by Clause, Hint, or Underlying Table/Index Declaration (priority order: 1, 2, 3)			
	PARALLEL Hint	**PARALLEL Clause**	**ALTER SESSION**	**Parallel Declaration**
Parallel query table scan (partitioned or nonpartitioned table)	(1) PARALLEL		(2) FORCE PARALLEL QUERY	(3) of table
Parallel query index range scan (partitioned index)	(1) PARALLEL_INDEX		(2) FORCE PARALLEL QUERY	(2) of index
Parallel UPDATE or DELETE (partitioned table only)	(1) PARALLEL		(2) FORCE PARALLEL DML	(3) of table being updated or deleted from
INSERT operation of parallel INSERT... SELECT (partitioned or nonpartitioned table)	(1) PARALLEL of insert		(2) FORCE PARALLEL DML	(3) of table being inserted into
SELECT operation of INSERT ... SELECT when INSERT is parallel	Takes degree from INSERT statement			
SELECT operation of INSERT ... SELECT when INSERT is serial	(1) PARALLEL			(2) of table being selected from
CREATE operation of parallel CREATE TABLE ... AS SELECT (partitioned or nonpartitioned table)	(Note: Hint in select clause does not affect the create operation.)	(2)	(1) FORCE PARALLEL DDL	
SELECT operation of CREATE TABLE ... AS SELECT when CREATE is parallel	Takes degree from CREATE statement			
SELECT operation of CREATE TABLE ... AS SELECT when CREATE is serial	(1) PARALLEL or PARALLEL_INDEX			(2) of querying tables or partitioned indexes
Parallel CREATE INDEX (partitioned or nonpartitioned index)		(2)	(1) FORCE PARALLEL DDL	
Parallel REBUILD INDEX (nonpartitioned index)		(2)	(1) FORCE PARALLEL DDL	

Table 24–3 (Cont.) Parallelization Rules

Parallel Operation	Parallelized by Clause, Hint, or Underlying Table/Index Declaration (priority order: 1, 2, 3)			
	PARALLEL Hint	PARALLEL Clause	ALTER SESSION	Parallel Declaration
REBUILD INDEX (partitioned index)—never parallelized				
Parallel REBUILD INDEX partition		(2)	(1) FORCE PARALLEL DDL	
Parallel MOVE or SPLIT partition		(2)	(1) FORCE PARALLEL DDL	

Enabling Parallelism for Tables and Queries

The DOP of tables involved in parallel operations affect the DOP for operations on those tables. Therefore, after setting parallel tuning parameters, you must also enable parallel execution for each table you want parallelized, using the PARALLEL clause of the CREATE TABLE or ALTER TABLE statements. You can also use the PARALLEL hint with SQL statements to enable parallelism for that operation only, or use the FORCE option of the ALTER SESSION statement to enable parallelism for all subsequent operations in the session.

When you parallelize tables, you can also specify the DOP or allow Oracle to use a default DOP. The value of the default DOP is derived automatically, based on the value of PARALLEL_THREADS_PER_CPU and the number of CPUs available to Oracle.

```
ALTER TABLE employees PARALLEL;    -- uses default DOP
ALTER TABLE employees PARALLEL 4;  -- users DOP of 4
```

Degree of Parallelism and Adaptive Multiuser: How They Interact

The DOP specifies the number of available processes, or threads, used in parallel operations. Each parallel thread can use one or two query processes, depending on the query's complexity.

The adaptive multiuser feature adjusts the DOP based on user load. For example, you might have a table with a DOP of 5. This DOP might be acceptable with 10 users. However, if 10 more users enter the system and you enable the PARALLEL_ADAPTIVE_MULTI_USER feature, Oracle reduces the DOP to spread resources more evenly according to the perceived Oracle load.

Once Oracle determines the DOP for a query, the DOP does not change for the duration of the query.

It is best to use the parallel adaptive multiuser feature when users process simultaneous parallel execution operations. By default, PARALLEL_ADAPTIVE_MULTI_USER is set to TRUE, which optimizes the performance of systems with concurrent parallel SQL execution operations. If PARALLEL_ADAPTIVE_MULTI_USER is set to FALSE, each parallel SQL execution operation receives the requested number of parallel execution server processes regardless of the impact to the performance of the system as long as sufficient resources have been configured.

How the Adaptive Multiuser Algorithm Works

The adaptive multiuser algorithm has several inputs. The algorithm first considers the number of active Oracle Server processes as calculated by Oracle. The algorithm then considers the default settings for parallelism as set in the initialization parameter file, as well as parallelism options used in CREATE TABLE and ALTER TABLE statements and SQL hints.

When a system is overloaded and the input DOP is larger than the default DOP, the algorithm uses the default degree as input. The system then calculates a reduction factor that it applies to the input DOP. For example, using a 16-CPU system, when the first user enters the system and it is idle, it will be granted a DOP of 32. The next user will be give a DOP of eight, the next four, and so on. If the system settles into a steady state of eight users issuing queries, all the users will eventually be given a DOP of 4, thus dividing the system evenly among all the parallel users.

Forcing Parallel Execution for a Session

If you are sure you want to execute in parallel and want to avoid setting the DOP for a table or modifying the queries involved, you can force parallelism with the following statement:

```
ALTER SESSION FORCE PARALLEL QUERY;
```

All subsequent queries will be executed in parallel provided no restrictions are violated. You can also force DML and DDL statements. This clause overrides any parallel clause specified in subsequent statements in the session, but is overridden by a parallel hint.

In typical OLTP environments, for example, the tables are not set parallel, but nightly batch scripts may want to collect data from these tables in parallel. By setting the DOP in the session, the user avoids altering each table in parallel and then altering it back to serial when finished.

Controlling Performance with the Degree of Parallelism

The initialization parameter PARALLEL_THREADS_PER_CPU affects algorithms controlling both the DOP and the adaptive multiuser feature. Oracle multiplies the value of PARALLEL_THREADS_PER_CPU by the number of CPUs for each instance to derive the number of threads to use in parallel operations.

The adaptive multiuser feature also uses the default DOP to compute the target number of query server processes that should exist in a system. When a system is running more processes than the target number, the adaptive algorithm reduces the DOP of new queries as required. Therefore, you can also use PARALLEL_THREADS_PER_CPU to control the adaptive algorithm.

PARALLEL_THREADS_PER_CPU enables you to adjust for hardware configurations with I/O subsystems that are slow relative to the CPU speed and for application workloads that perform few computations relative to the amount of data involved. If the system is neither CPU-bound nor I/O-bound, then the PARALLEL_THREADS_PER_CPU value should be increased. This increases the default DOP and allow better utilization of hardware resources. The default for PARALLEL_THREADS_PER_CPU on most platforms is 2. However, the default for machines with relatively slow I/O subsystems can be as high as eight.

Tuning General Parameters for Parallel Execution

This section discusses the following topics:

- Parameters Establishing Resource Limits for Parallel Operations
- Parameters Affecting Resource Consumption
- Parameters Related to I/O

Parameters Establishing Resource Limits for Parallel Operations

The parameters that establish resource limits are:

- PARALLEL_MAX_SERVERS
- PARALLEL_MIN_SERVERS
- SHARED_POOL_SIZE
- PARALLEL_MIN_PERCENT

PARALLEL_MAX_SERVERS

The PARALLEL_MAX_SEVERS parameter sets a resource limit on the maximum number of processes available for parallel execution. Most parallel operations need at most twice the number of query server processes as the maximum DOP attributed to any table in the operation.

Oracle sets PARALLEL_MAX_SERVERS to a default value that is sufficient for most systems. The default value for PARALLEL_MAX_SERVERS is as follows:

```
(CPU_COUNT x PARALLEL_THREADS_PER_CPU x (2 if PGA_AGGREGATE_TARGET > 0;
otherwise 1) x 5)
```

This might not be enough for parallel queries on tables with higher DOP attributes. We recommend users who expects to run queries of higher DOP to set PARALLEL_MAX_SERVERS as follows:

```
2 x DOP x NUMBER_OF_CONCURRENT_USERS
```

For example, setting PARALLEL_MAX_SERVERS to 64 will allow you to run four parallel queries simultaneously, assuming that each query is using two slave sets with a DOP of eight for each set.

If the hardware system is neither CPU bound nor I/O bound, then you can increase the number of concurrent parallel execution users on the system by adding more query server processes. When the system becomes CPU- or I/O-bound, however, adding more concurrent users becomes detrimental to the overall performance. Careful setting of PARALLEL_MAX_SERVERS is an effective method of restricting the number of concurrent parallel operations.

If users initiate too many concurrent operations, Oracle might not have enough query server processes. In this case, Oracle executes the operations sequentially or displays an error if PARALLEL_MIN_PERCENT is set to a value other than the default value of 0 (zero).

This condition can be verified through the GV$SYSSTAT view by comparing the statistics for parallel operations not downgraded and parallel operations downgraded to serial. For example:

```
SELECT * FROM GV$SYSSTAT WHERE name LIKE 'Parallel operation%';
```

When Users Have Too Many Processes When concurrent users have too many query server processes, memory contention (paging), I/O contention, or excessive context switching can occur. This contention can reduce system throughput to a level lower than if parallel execution were not used. Increase the PARALLEL_MAX_SERVERS

value only if the system has sufficient memory and I/O bandwidth for the resulting load.

You can use operating system performance monitoring tools to determine how much memory, swap space and I/O bandwidth are free. Look at the runq lengths for both your CPUs and disks, as well as the service time for I/Os on the system. Verify that the machine has sufficient swap space exists on the machine to add more processes. Limiting the total number of query server processes might restrict the number of concurrent users who can execute parallel operations, but system throughput tends to remain stable.

Increasing the Number of Concurrent Users

To increase the number of concurrent users, you must restrict the resource usage of each individual user. You can achieve this by using the parallel adaptive multiuser feature or by using resource consumer groups. See *Oracle Database Administrator's Guide* and *Oracle Database Concepts* for more information about resource consumer groups and the Database Resource Manager.

Limiting the Number of Resources for a User

You can limit the amount of parallelism available to a given user by establishing a resource consumer group for the user. Do this to limit the number of sessions, concurrent logons, and the number of parallel processes that any one user or group of users can have.

Each query server process working on a parallel execution statement is logged on with a session ID. Each process counts against the user's limit of concurrent sessions. For example, to limit a user to 10 parallel execution processes, set the user's limit to 11. One process is for the parallel coordinator and the other 10 consist of two sets of query server servers. This would allow one session for the parallel coordinator and 10 sessions for the parallel execution processes.

See *Oracle Database Administrator's Guide* for more information about managing resources with user profiles and *Oracle Real Application Clusters Administrator's Guide* for more information on querying `GV$` views.

PARALLEL_MIN_SERVERS

The recommended value for the `PARALLEL_MIN_SERVERS` parameter is 0 (zero), which is the default.

This parameter lets you specify in a single instance the number of processes to be started and reserved for parallel operations. The syntax is:

```
PARALLEL_MIN_SERVERS=n
```

The *n* variable is the number of processes you want to start and reserve for parallel operations.

Setting PARALLEL_MIN_SERVERS balances the startup cost against memory usage. Processes started using PARALLEL_MIN_SERVERS do not exit until the database is shut down. This way, when a query is issued the processes are likely to be available. It is desirable, however, to recycle query server processes periodically since the memory these processes use can become fragmented and cause the high water mark to slowly increase. When you do not set PARALLEL_MIN_SERVERS, processes exit after they are idle for five minutes.

SHARED_POOL_SIZE

Parallel execution requires memory resources in addition to those required by serial SQL execution. Additional memory is used for communication and passing data between query server processes and the query coordinator.

Oracle Database allocates memory for query server processes from the shared pool. Tune the shared pool as follows:

- Allow for other clients of the shared pool, such as shared cursors and stored procedures.
- Remember that larger values improve performance in multiuser systems, but smaller values use less memory.
- You must also take into account that using parallel execution generates more cursors. Look at statistics in the V$SQLAREA view to determine how often Oracle recompiles cursors. If the cursor hit ratio is poor, increase the size of the pool. This happens only when you have a large number of distinct queries.

  You can then monitor the number of buffers used by parallel execution and compare the `shared pool` PX `msg pool` to the current high water mark reported in output from the view V$PX_PROCESS_SYSSTAT.

  Note: If you do not have enough memory available, error message 12853 occurs (insufficient memory for PX buffers: current *string*K, max needed *string*K). This is caused by having insufficient SGA memory available for PX buffers. You need to reconfigure the SGA to have at least (MAX - CURRENT) bytes of additional memory.

By default, Oracle allocates parallel execution buffers from the shared pool.

If Oracle displays the following error on startup:

```
ORA-27102: out of memory
SVR4 Error: 12: Not enough space
```

You should reduce the value for SHARED_POOL_SIZE low enough so your database starts. After reducing the value of SHARED_POOL_SIZE, you might see the error:

```
ORA-04031: unable to allocate 16084 bytes of shared memory
   ("SHARED pool","unknown object","SHARED pool heap","PX msg pool")
```

If so, execute the following query to determine why Oracle could not allocate the 16,084 bytes:

```
SELECT NAME, SUM(BYTES) FROM V$SGASTAT WHERE POOL='SHARED POOL'
  GROUP BY ROLLUP (NAME);
```

Your output should resemble the following:

```
NAME                         SUM(BYTES)
-------------------------- ----------
PX msg pool                   1474572
free memory                    562132
                              2036704
```

If you specify SHARED_POOL_SIZE and the amount of memory you need to reserve is bigger than the pool, Oracle does not allocate all the memory it can get. Instead, it leaves some space. When the query runs, Oracle tries to get what it needs. Oracle uses the 560 KB and needs another 16KB when it fails. The error does not report the cumulative amount that is needed. The best way of determining how much more memory is needed is to use the formulas in "Adding Memory for Message Buffers" on page 24-52.

To resolve the problem in the current example, increase the value for SHARED_POOL_SIZE. As shown in the sample output, the SHARED_POOL_SIZE is about 2 MB. Depending on the amount of memory available, you could increase the value of SHARED_POOL_SIZE to 4 MB and attempt to start your database. If Oracle continues to display an ORA-4031 message, gradually increase the value for SHARED_POOL_SIZE until startup is successful.

Computing Additional Memory Requirements for Message Buffers

After you determine the initial setting for the shared pool, you must calculate additional memory requirements for message buffers and determine how much additional space you need for cursors.

Adding Memory for Message Buffers You must increase the value for the SHARED_POOL_SIZE parameter to accommodate message buffers. The message buffers allow query server processes to communicate with each other.

Oracle uses a fixed number of buffers for each virtual connection between producer query servers and consumer query servers. Connections increase as the square of the DOP increases. For this reason, the maximum amount of memory used by parallel execution is bound by the highest DOP allowed on your system. You can control this value by using either the PARALLEL_MAX_SERVERS parameter or by using policies and profiles.

To calculate the amount of memory required, use one of the following formulas:

- For SMP systems:

  ```
  mem in bytes = (3 x size x users x groups x connections)
  ```

- For SMP Real Application Clusters and MPP systems:

  ```
  mem in bytes = ((3 x local) + (2 x remote)) x (size x users x groups)
    / instances
  ```

Each instance uses the memory computed by the formula.

The terms are:

- SIZE = PARALLEL_EXECUTION_MESSAGE_SIZE
- USERS = the number of concurrent parallel execution users that you expect to have running with the optimal DOP
- GROUPS = the number of query server process groups used for each query

  A simple SQL statement requires only one group. However, if your queries involve subqueries which will be processed in parallel, then Oracle uses an additional group of query server processes.

- CONNECTIONS = (DOP^2 + 2 x DOP)

  If your system is a cluster or MPP, then you should account for the number of instances because this will increase the DOP. In other words, using a DOP of 4 on a two instance cluster results in a DOP of 8. A value of PARALLEL_MAX_SERVERS times the number of instances divided by four is a conservative estimate to use as a starting point.

- LOCAL = CONNECTIONS / INSTANCES
- REMOTE = CONNECTIONS - LOCAL

Add this amount to your original setting for the shared pool. However, before setting a value for either of these memory structures, you must also consider additional memory for cursors, as explained in the following section.

Calculating Additional Memory for Cursors Parallel execution plans consume more space in the SQL area than serial execution plans. You should regularly monitor shared pool resource use to ensure that the memory used by both messages and cursors can accommodate your system's processing requirements.

Adjusting Memory After Processing Begins

The formulas in this section are just starting points. Whether you are using automated or manual tuning, you should monitor usage on an on-going basis to make sure the size of memory is not too large or too small. To do this, tune the shared pool using the following query:

```
SELECT POOL, NAME, SUM(BYTES) FROM V$SGASTAT WHERE POOL LIKE '%pool%'
  GROUP BY ROLLUP (POOL, NAME);
```

Your output should resemble the following:

```
POOL        NAME                         SUM(BYTES)
----------- ------------------------- ----------
shared pool Checkpoint queue                38496
shared pool KGFF heap                        1964
shared pool KGK heap                         4372
shared pool KQLS heap                     1134432
shared pool LRMPD SGA Table                 23856
shared pool PLS non-lib hp                   2096
shared pool PX subheap                     186828
shared pool SYSTEM PARAMETERS               55756
shared pool State objects                 3907808
shared pool character set memory            30260
shared pool db_block_buffers               200000
shared pool db_block_hash_buckets           33132
shared pool db_files                       122984
shared pool db_handles                      52416
shared pool dictionary cache               198216
shared pool dlm shared memory             5387924
shared pool enqueue_resources               29016
shared pool event statistics per sess      264768
shared pool fixed allocation callback        1376
shared pool free memory                   26329104
shared pool gc_*                             64000
shared pool latch nowait fails or sle        34944
```

```
shared pool library cache                 2176808
shared pool log_buffer                      24576
shared pool log_checkpoint_timeout          24700
shared pool long op statistics array        30240
shared pool message pool freequeue         116232
shared pool miscellaneous                  267624
shared pool processes                       76896
shared pool session param values            41424
shared pool sessions                       170016
shared pool sql area                      9549116
shared pool table columns                  148104
shared pool trace_buffers_per_process     1476320
shared pool transactions                    18480
shared pool trigger inform                  24684
shared pool                             52248968
                                        90641768
```

Evaluate the memory used as shown in your output, and alter the setting for SHARED_POOL_SIZE based on your processing needs.

To obtain more memory usage statistics, execute the following query:

```
SELECT * FROM V$PX_PROCESS_SYSSTAT WHERE STATISTIC LIKE 'Buffers%';
```

Your output should resemble the following:

```
STATISTIC                           VALUE
-------------------                 -----
Buffers Allocated                   23225
Buffers Freed                       23225
Buffers Current                         0
Buffers HWM                          3620
```

The amount of memory used appears in the Buffers Current and Buffers HWM statistics. Calculate a value in bytes by multiplying the number of buffers by the value for PARALLEL_EXECUTION_MESSAGE_SIZE. Compare the high water mark to the parallel execution message pool size to determine if you allocated too much memory. For example, in the first output, the value for large pool as shown in px msg pool is 38,092,812 or 38 MB. The Buffers HWM from the second output is 3,620, which when multiplied by a parallel execution message size of 4,096 is 14,827,520, or approximately 15 MB. In this case, the high water mark has reached approximately 40 percent of its capacity.

PARALLEL_MIN_PERCENT

The recommended value for the PARALLEL_MIN_PERCENT parameter is 0 (zero).

This parameter enables users to wait for an acceptable DOP, depending on the application in use. Setting this parameter to values other than 0 (zero) causes Oracle to return an error when the requested DOP cannot be satisfied by the system at a given time. For example, if you set PARALLEL_MIN_PERCENT to 50, which translates to 50 percent, and the DOP is reduced by 50 percent or greater because of the adaptive algorithm or because of a resource limitation, then Oracle returns ORA-12827. For example:

```
SELECT /*+ PARALLEL(e, 8, 1) */ d.department_id, SUM(SALARY)
FROM employees e, departments d WHERE e.department_id = d.department_id
GROUP BY d.department_id ORDER BY d.department_id;
```

Oracle responds with this message:

```
ORA-12827: insufficient parallel query slaves available
```

Parameters Affecting Resource Consumption

The first group of parameters discussed in this section affects memory and resource consumption for all parallel operations, in particular, for parallel execution. These parameters are:

- PGA_AGGREGATE_TARGET
- PARALLEL_EXECUTION_MESSAGE_SIZE

A second subset of parameters discussed in this section explains parameters affecting parallel DML and DDL.

To control resource consumption, you should configure memory at two levels:

- At the Oracle level, so the system uses an appropriate amount of memory from the operating system.
- At the operating system level for consistency. On some platforms, you might need to set operating system parameters that control the total amount of virtual memory available, summed across all processes.

The SGA is typically part of real physical memory. The SGA is static and of fixed size; if you want to change its size, shut down the database, make the change, and restart the database. Oracle allocates the shared pool out of the SGA.

A large percentage of the memory used in data warehousing operations is more dynamic. This memory comes from process memory (PGA), and both the size of

process memory and the number of processes can vary greatly. Use the PGA_AGGREGATE_TARGET parameter to control both the process memory and the number of processes.

PGA_AGGREGATE_TARGET

You can simplify and improve the way PGA memory is allocated by enabling automatic PGA memory management. In this mode, Oracle dynamically adjusts the size of the portion of the PGA memory dedicated to work areas, based on an overall PGA memory target explicitly set by the DBA. To enable automatic PGA memory management, you have to set the initialization parameter PGA_AGGREGATE_TARGET. See *Oracle Database Performance Tuning Guide* for descriptions of how to use PGA_AGGREGATE_TARGET in different scenarios.

HASH_AREA_SIZE HASH_AREA_SIZE has been deprecated and you should use PGA_AGGREGATE_TARGET instead.

SORT_AREA_SIZE SORT_AREA_SIZE has been deprecated and you should use PGA_AGGREGATE_TARGET instead.

PARALLEL_EXECUTION_MESSAGE_SIZE

The PARALLEL_EXECUTION_MESSAGE_SIZE parameter specifies the size of the buffer used for parallel execution messages. The default value is os specific, but is typically 2K. This value should be adequate for most applications, however, increasing this value can improve performance. Consider increasing this value if you have adequate free memory in the shared pool or if you have sufficient operating system memory and can increase your shared pool size to accommodate the additional amount of memory required. Parameters Affecting Resource Consumption for Parallel DML and Parallel DDL

Parameters Affecting Resource Consumption for Parallel DML and Parallel DDL

The parameters that affect parallel DML and parallel DDL resource consumption are:

- TRANSACTIONS
- FAST_START_PARALLEL_ROLLBACK
- LOG_BUFFER
- DML_LOCKS
- ENQUEUE_RESOURCES

Parallel inserts, updates, and deletes require more resources than serial DML operations. Similarly, PARALLEL CREATE TABLE ... AS SELECT and PARALLEL CREATE INDEX can require more resources. For this reason, you may need to increase the value of several additional initialization parameters. These parameters do *not* affect resources for queries.

TRANSACTIONS For parallel DML and DDL, each query server process starts a transaction. The parallel coordinator uses the two-phase commit protocol to commit transactions; therefore, the number of transactions being processed increases by the DOP. As a result, you might need to increase the value of the TRANSACTIONS initialization parameter.

The TRANSACTIONS parameter specifies the maximum number of concurrent transactions. The default assumes no parallelism. For example, if you have a DOP of 20, you will have 20 more new server transactions (or 40, if you have two server sets) and 1 coordinator transaction. In this case, you should increase TRANSACTIONS by 21 (or 41) if the transactions are running in the same instance. If you do not set this parameter, Oracle sets it to a value equal to 1.1 x SESSIONS. This discussion does not apply if you are using server-managed undo.

FAST_START_PARALLEL_ROLLBACK If a system fails when there are uncommitted parallel DML or DDL transactions, you can speed up transaction recovery during startup by using the FAST_START_PARALLEL_ROLLBACK parameter.

This parameter controls the DOP used when recovering terminated transactions. Terminated transactions are transactions that are active before a system failure. By default, the DOP is chosen to be at most two times the value of the CPU_COUNT parameter.

If the default DOP is insufficient, set the parameter to the HIGH. This gives a maximum DOP of at most four times the value of the CPU_COUNT parameter. This feature is available by default.

LOG_BUFFER Check the statistic redo buffer allocation retries in the V$SYSSTAT view. If this value is high relative to redo blocks written, try to increase the LOG_BUFFER size. A common LOG_BUFFER size for a system generating numerous logs is 3 MB to 5 MB. If the number of retries is still high after increasing LOG_BUFFER size, a problem might exist with the disk on which the log files reside. In that case, tune the I/O subsystem to increase the I/O rates for redo. One way of doing this is to use fine-grained striping across multiple disks. For example, use a stripe size of 16 KB. A simpler approach is to isolate redo logs on their own disk.

DML_LOCKS This parameter specifies the maximum number of DML locks. Its value should equal the total number of locks on all tables referenced by all users. A parallel DML operation's lock and enqueue resource requirement is very different from serial DML. Parallel DML holds many more locks, so you should increase the value of the ENQUEUE_RESOURCES and DML_LOCKS parameters by equal amounts.

Table 24–4 shows the types of locks acquired by coordinator and parallel execution server processes for different types of parallel DML statements. Using this information, you can determine the value required for these parameters.

Table 24–4 Locks Acquired by Parallel DML Statements

Type of statement	Coordinator process acquires:	Each parallel execution server acquires:
Parallel UPDATE or DELETE into partitioned table; WHERE clause pruned to a subset of partitions or subpartitions	1 table lock SX	1 table lock SX
	1 partition lock X for each pruned (sub)partition	1 partition lock NULL for each pruned (sub)partition owned by the query server process
		1 partition-wait lock S for each pruned (sub)partition owned by the query server process
Parallel row-migrating UPDATE into partitioned table; WHERE clause pruned to a subset of (sub)partitions	1 table lock SX	1 table lock SX
	1 partition X lock for each pruned (sub)partition	1 partition lock NULL for each pruned (sub)partition owned by the query server process
		1 partition-wait lock S for each pruned partition owned by the query server process
	1 partition lock SX for all other (sub)partitions	1 partition lock SX for all other (sub)partitions
Parallel UPDATE, MERGE, DELETE, or INSERT into partitioned table	1 table lock SX	1 table lock SX
	Partition locks X for all (sub)partitions	1 partition lock NULL for each (sub)partition
		1 partition-wait lock S for each (sub)partition

Table 24–4 (Cont.) Locks Acquired by Parallel DML Statements

Type of statement	Coordinator process acquires:	Each parallel execution server acquires:
Parallel INSERT into partitioned table; destination table with partition or subpartition clause	1 table lock SX 1 partition lock X for each specified (sub)partition	1 table lock SX 1 partition lock NULL for each specified (sub)partition 1 partition-wait lock S for each specified (sub)partition
Parallel INSERT into nonpartitioned table	1 table lock X	None

Note: Table, partition, and partition-wait DML locks all appear as TM locks in the V$LOCK view.

Consider a table with 600 partitions running with a DOP of 100. Assume all partitions are involved in a parallel UPDATE or DELETE statement with no row-migrations.

The coordinator acquires:

- 1 table lock SX
- 600 partition locks X

Total server processes acquires:

- 100 table locks SX
- 600 partition locks NULL
- 600 partition-wait locks S

ENQUEUE_RESOURCES This parameter sets the number of resources that can be locked by the lock manager. Parallel DML operations require many more resources than serial DML. Oracle allocates more enqueue resources as needed.

Parameters Related to I/O

The parameters that affect I/O are:

- DB_CACHE_SIZE

- DB_BLOCK_SIZE
- DB_FILE_MULTIBLOCK_READ_COUNT
- DISK_ASYNCH_IO and TAPE_ASYNCH_IO

These parameters also affect the optimizer which ensures optimal performance for parallel execution I/O operations.

DB_CACHE_SIZE

When you perform parallel updates, merges, and deletes, the buffer cache behavior is very similar to any OLTP system running a high volume of updates.

DB_BLOCK_SIZE

The recommended value for this parameter is 8 KB or 16 KB.

Set the database block size when you create the database. If you are creating a new database, use a large block size such as 8 KB or 16 KB.

DB_FILE_MULTIBLOCK_READ_COUNT

The recommended value for this parameter is eight for 8 KB block size, or four for 16 KB block size. The default is 8.

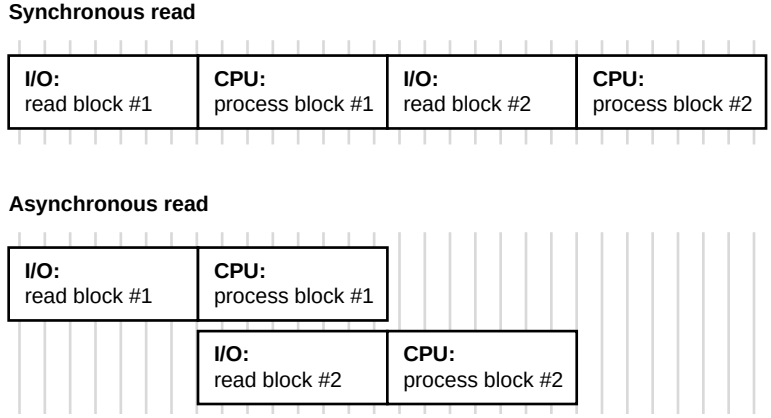
This parameter determines how many database blocks are read with a single operating system READ call. The upper limit for this parameter is platform-dependent. If you set DB_FILE_MULTIBLOCK_READ_COUNT to an excessively high value, your operating system will lower the value to the highest allowable level when you start your database. In this case, each platform uses the highest value possible. Maximum values generally range from 64 KB to 1 MB.

DISK_ASYNCH_IO and TAPE_ASYNCH_IO

The recommended value for both of these parameters is TRUE.

These parameters enable or disable the operating system's asynchronous I/O facility. They allow query server processes to overlap I/O requests with processing when performing table scans. If the operating system supports asynchronous I/O, leave these parameters at the default value of TRUE. Figure 24–6 illustrates how asynchronous read works.

Figure 24–6 Asynchronous Read



Asynchronous operations are currently supported for parallel table scans, hash joins, sorts, and serial table scans. However, this feature can require operating system specific configuration and may not be supported on all platforms. Check your Oracle operating system-specific documentation.

Monitoring and Diagnosing Parallel Execution Performance

You should do the following tasks when diagnosing parallel execution performance problems:

- Quantify your performance expectations to determine whether there is a problem.
- Determine whether a problem pertains to optimization, such as inefficient plans that might require reanalyzing tables or adding hints, or whether the problem pertains to execution, such as simple operations like scanning, loading, grouping, or indexing running much slower than published guidelines.
- Determine whether the problem occurs when running in parallel, such as load imbalance or resource bottlenecks, or whether the problem is also present for serial operations.

Performance expectations are based on either prior performance metrics (for example, the length of time a given query took last week or on the previous version of Oracle) or scaling and extrapolating from serial execution times (for example, serial execution took 10 minutes while parallel execution took 5 minutes). If the performance does not meet your expectations, consider the following questions:

- Did the execution plan change?

  If so, you should gather statistics and decide whether to use index-only access and a CREATE TABLE AS SELECT statement. You should use index hints if your system is CPU-bound. You should also study the EXPLAIN PLAN output.

- Did the data set change?

  If so, you should gather statistics to evaluate any differences.

- Is the hardware overtaxed?

  If so, you should check CPU, I/O, and swap memory.

After setting your basic goals and answering these questions, you need to consider the following topics:

- Is There Regression?
- Is There a Plan Change?
- Is There a Parallel Plan?
- Is There a Serial Plan?
- Is There Parallel Execution?
- Is the Workload Evenly Distributed?

Is There Regression?

Does parallel execution's actual performance deviate from what you expected? If performance is as you expected, could there be an underlying performance problem? Perhaps you have a desired outcome in mind to which you are comparing the current outcome. Perhaps you have justifiable performance expectations that the system does not achieve. You might have achieved this level of performance or a particular execution plan in the past, but now, with a similar environment and operation, the system is not meeting this goal.

If performance is not as you expected, can you quantify the deviation? For data warehousing operations, the execution plan is key. For critical data warehousing operations, save the EXPLAIN PLAN results. Then, as you analyze and reanalyze the data, upgrade Oracle, and load new data, over time you can compare new execution plans with old plans. Take this approach either proactively or reactively.

Alternatively, you might find that plan performance improves if you use hints. You might want to understand why hints are necessary and determine how to get the

optimizer to generate the desired plan without hints. Try increasing the statistical sample size: better statistics can give you a better plan.

See *Oracle Database Performance Tuning Guide* for information on preserving plans throughout changes to your system, using plan stability and outlines.

Is There a Plan Change?

If there has been a change in the execution plan, determine whether the plan is or should be parallel or serial.

Is There a Parallel Plan?

If the execution plan is or should be parallel, study the `EXPLAIN PLAN` output. Did you analyze all the tables? Perhaps you need to use hints in a few cases. Verify that the hint provides better performance. See `utlxplp.sql` in the `rdbms/admin` directory.

Is There a Serial Plan?

If the execution plan is or should be serial, consider the following strategies:

- Use an index. Sometimes adding an index can greatly improve performance. Consider adding an extra column to the index. Perhaps your operation could obtain all its data from the index, and not require a table scan. Perhaps you need to use hints in a few cases. Verify that the hint provides better results.
- Compute statistics. If you do not analyze often and you can spare the time, it is a good practice to compute statistics. This is particularly important if you are performing many joins, and it will result in better plans. Alternatively, you can estimate statistics. If you use different sample sizes, the plan may change. Generally, the higher the sample size, the better the plan.
- Use histograms for nonuniform distributions.
- Check initialization parameters to be sure the values are reasonable.
- Replace bind variables with literals unless `CURSOR_SHARING` is set to `force` or `similar`.
- Determine whether execution is I/O- or CPU-bound. Then check the optimizer cost model.
- Convert subqueries to joins.

- Use the CREATE TABLE ... AS SELECT statement to break a complex operation into smaller pieces. With a large query referencing five or six tables, it may be difficult to determine which part of the query is taking the most time. You can isolate bottlenecks in the query by breaking it into steps and analyzing each step.

Is There Parallel Execution?

If the cause of regression cannot be traced to problems in the plan, the problem must be an execution issue. For data warehousing operations, both serial and parallel, consider how the plan uses memory. Check the paging rate and make sure the system is using memory as effectively as possible. Check buffer, sort, and hash area sizing. After you run a query or DML operation, look at the V$SESSTAT, V$PX_SESSTAT, and V$PQ_SYSSTAT views to see the number of server processes used and other information for the session and system.

Is the Workload Evenly Distributed?

If you are using parallel execution, is there unevenness in workload distribution? For example, if there are 10 CPUs and a single user, you can see whether the workload is evenly distributed across CPUs. This can vary over time, with periods that are more or less I/O intensive, but in general each CPU should have roughly the same amount of activity.

The statistics in V$PQ_TQSTAT show rows produced and consumed for each parallel execution server. This is a good indication of skew and does not require single user operation.

Operating system statistics show you the per-processor CPU utilization and per-disk I/O activity. Concurrently running tasks make it harder to see what is going on, however. It may be useful to run in single-user mode and check operating system monitors that show system level CPU and I/O activity.

If I/O problems occur, you might need to reorganize your data by spreading it over more devices. If parallel execution problems occur, check to be sure you have followed the recommendation to spread data over at least as many devices as CPUs.

If there is no skew in workload distribution, check for the following conditions:

- Is there device contention?
- Is there controller contention?
- Is the system I/O-bound with too little parallelism? If so, consider increasing parallelism up to the number of devices.

- Is the system CPU-bound with too much parallelism? Check the operating system CPU monitor to see whether a lot of time is being spent in system calls. The resource might be overcommitted, and too much parallelism might cause processes to compete with themselves.
- Are there more concurrent users than the system can support?

Monitoring Parallel Execution Performance with Dynamic Performance Views

After your system has run for a few days, monitor parallel execution performance statistics to determine whether your parallel processing is optimal. Do this using any of the views discussed in this section.

In Oracle Real Application Clusters, global versions of the views described in this section aggregate statistics from multiple instances. The global views have names beginning with G, such as GV$FILESTAT for V$FILESTAT, and so on.

V$PX_BUFFER_ADVICE

The V$PX_BUFFER_ADVICE view provides statistics on historical and projected maximum buffer usage by all parallel queries. You can consult this view to reconfigure SGA size in response to insufficient memory problems for parallel queries.

V$PX_SESSION

The V$PX_SESSION view shows data about query server sessions, groups, sets, and server numbers. It also displays real-time data about the processes working on behalf of parallel execution. This table includes information about the requested DOP and the actual DOP granted to the operation.

V$PX_SESSTAT

The V$PX_SESSTAT view provides a join of the session information from V$PX_SESSION and the V$SESSTAT table. Thus, all session statistics available to a normal session are available for all sessions performed using parallel execution.

V$PX_PROCESS

The V$PX_PROCESS view contains information about the parallel processes, including status, session ID, process ID, and other information.

V$PX_PROCESS_SYSSTAT

The V$PX_PROCESS_SYSSTAT view shows the status of query servers and provides buffer allocation statistics.

V$PQ_SESSTAT

The V$PQ_SESSTAT view shows the status of all current server groups in the system such as data about how queries allocate processes and how the multiuser and load balancing algorithms are affecting the default and hinted values. V$PQ_SESSTAT will be obsolete in a future release.

You might need to adjust some parameter settings to improve performance after reviewing data from these views. In this case, refer to the discussion of "Tuning General Parameters for Parallel Execution" on page 24-47. Query these views periodically to monitor the progress of long-running parallel operations.

For many dynamic performance views, you must set the parameter TIMED_STATISTICS to TRUE in order for Oracle to collect statistics for each view. You can use the ALTER SYSTEM or ALTER SESSION statements to turn TIMED_STATISTICS on and off.

V$FILESTAT

The V$FILESTAT view sums read and write requests, the number of blocks, and service times for every datafile in every tablespace. Use V$FILESTAT to diagnose I/O and workload distribution problems.

You can join statistics from V$FILESTAT with statistics in the DBA_DATA_FILES view to group I/O by tablespace or to find the filename for a given file number. Using a ratio analysis, you can determine the percentage of the total tablespace activity used by each file in the tablespace. If you make a practice of putting just one large, heavily accessed object in a tablespace, you can use this technique to identify objects that have a poor physical layout.

You can further diagnose disk space allocation problems using the DBA_EXTENTS view. Ensure that space is allocated evenly from all files in the tablespace. Monitoring V$FILESTAT during a long-running operation and then correlating I/O activity to the EXPLAIN PLAN output is a good way to follow progress.

V$PARAMETER

The V$PARAMETER view lists the name, current value, and default value of all system parameters. In addition, the view shows whether a parameter is a session

parameter that you can modify online with an ALTER SYSTEM or ALTER SESSION statement.

V$PQ_TQSTAT

As a simple example, consider a hash join between two tables, with a join on a column with only two distinct values. At best, this hash function will have one hash value to parallel execution server A and the other to parallel execution server B. A DOP of two is fine, but, if it is four, then at least two parallel execution servers have no work. To discover this type of skew, use a query similar to the following example:

```
SELECT dfo_number, tq_id, server_type, process, num_rows
FROM V$PQ_TQSTAT ORDER BY dfo_number DESC, tq_id, server_type, process;
```

The best way to resolve this problem might be to choose a different join method; a nested loop join might be the best option. Alternatively, if one of the join tables is small relative to the other, a BROADCAST distribution method can be hinted using PQ_DISTRIBUTE hint. Note that the optimizer considers the BROADCAST distribution method, but requires OPTIMIZER_FEATURES_ENABLE set to 9.0.2 or higher.

Now, assume that you have a join key with high cardinality, but one of the values contains most of the data, for example, lava lamp sales by year. The only year that had big sales was 1968, and thus, the parallel execution server for the 1968 records will be overwhelmed. You should use the same corrective actions as described previously.

The V$PQ_TQSTAT view provides a detailed report of message traffic at the table queue level. V$PQ_TQSTAT data is valid only when queried from a session that is executing parallel SQL statements. A table queue is the pipeline between query server groups, between the parallel coordinator and a query server group, or between a query server group and the coordinator. The table queues are represented explicitly in the operation column by PX SEND <partitioning type> (for example, PX SEND HASH) and PX RECEIVE. For backward compatibility, the row labels of PARALLEL_TO_PARALLEL, SERIAL_TO_PARALLEL, or PARALLEL_TO_SERIAL will continue to have the same semantics as previous releases and can be used as before to infer the table queue allocation. In addition, the top of the parallel plan is marked by a new node with operation PX COORDINATOR.

V$PQ_TQSTAT has a row for each query server process that reads from or writes to in each table queue. A table queue connecting 10 consumer processes to 10 producer processes has 20 rows in the view. Sum the bytes column and group by TQ_ID, the table queue identifier, to obtain the total number of bytes sent through each table

queue. Compare this with the optimizer estimates; large variations might indicate a need to analyze the data using a larger sample.

Compute the variance of bytes grouped by TQ_ID. Large variances indicate workload imbalances. You should investigate large variances to determine whether the producers start out with unequal distributions of data, or whether the distribution itself is skewed. If the data itself is skewed, this might indicate a low cardinality, or low number of distinct values.

Note that the V$PQ_TQSTAT view will be renamed in a future release to V$PX_TQSTSAT.

V$SESSTAT and V$SYSSTAT

The V$SESSTAT view provides parallel execution statistics for each session. The statistics include total number of queries, DML and DDL statements executed in a session and the total number of intrainstance and interinstance messages exchanged during parallel execution during the session.

V$SYSSTAT provides the same statistics as V$SESSTAT, but for the entire system.

Monitoring Session Statistics

These examples use the dynamic performance views described in "Monitoring Parallel Execution Performance with Dynamic Performance Views" on page 24-65.

Use GV$PX_SESSION to determine the configuration of the server group executing in parallel. In this example, sessions 9 is the query coordinator, while sessions 7 and 21 are in the first group, first set. Sessions 18 and 20 are in the first group, second set. The requested and granted DOP for this query is 2, as shown by Oracle's response to the following query:

```
SELECT QCSID, SID, INST_ID "Inst", SERVER_GROUP "Group", SERVER_SET "Set",
  DEGREE "Degree", REQ_DEGREE "Req Degree"
FROM GV$PX_SESSION ORDER BY QCSID, QCINST_ID, SERVER_GROUP, SERVER_SET;
```

Your output should resemble the following:

```
QCSID      SID        Inst       Group      Set        Degree     Req Degree
---------- ---------- ---------- ---------- ---------- ---------- ----------
         9          9          1
         9          7          1          1          1          2          2
         9         21          1          1          1          2          2
         9         18          1          1          2          2          2
         9         20          1          1          2          2          2
```

For a single instance, use SELECT FROM V$PX_SESSION and do not include the column name Instance ID.

The processes shown in the output from the previous example using GV$PX_SESSION collaborate to complete the same task. The next example shows the execution of a join query to determine the progress of these processes in terms of physical reads. Use this query to track any specific statistic:

```
SELECT QCSID, SID, INST_ID "Inst", SERVER_GROUP "Group", SERVER_SET "Set",
  NAME "Stat Name", VALUE
FROM GV$PX_SESSTAT A, V$STATNAME B
WHERE A.STATISTIC# = B.STATISTIC# AND NAME LIKE 'PHYSICAL READS'
  AND VALUE > 0 ORDER BY QCSID, QCINST_ID, SERVER_GROUP, SERVER_SET;
```

Your output should resemble the following:

```
QCSID  SID   Inst   Group  Set    Stat Name          VALUE
------ ----- ------ ------ ------ ------------------ ----------
     9     9      1                physical reads           3863
     9     7      1      1       1 physical reads              2
     9    21      1      1       1 physical reads              2
     9    18      1      1       2 physical reads              2
     9    20      1      1       2 physical reads              2
```

Use the previous type of query to track statistics in V$STATNAME. Repeat this query as often as required to observe the progress of the query server processes.

The next query uses V$PX_PROCESS to check the status of the query servers.

```
SELECT * FROM V$PX_PROCESS;
```

Your output should resemble the following:

```
SERV STATUS    PID    SPID      SID    SERIAL
---- --------- ------ --------- ------ ------
P002 IN USE        16     16955     21   7729
P003 IN USE        17     16957     20   2921
P004 AVAILABLE     18     16959
P005 AVAILABLE     19     16962
P000 IN USE        12      6999     18   4720
P001 IN USE        13      7004      7    234
```

Monitoring System Statistics

The V$SYSSTAT and V$SESSTAT views contain several statistics for monitoring parallel execution. Use these statistics to track the number of parallel queries, DMLs, DDLs, data flow operators (DFOs), and operations. Each query, DML, or DDL can have multiple parallel operations and multiple DFOs.

In addition, statistics also count the number of query operations for which the DOP was reduced, or downgraded, due to either the adaptive multiuser algorithm or the depletion of available parallel execution servers.

Finally, statistics in these views also count the number of messages sent on behalf of parallel execution. The following syntax is an example of how to display these statistics:

```
SELECT NAME, VALUE FROM GV$SYSSTAT
WHERE UPPER (NAME) LIKE '%PARALLEL OPERATIONS%'
OR UPPER (NAME) LIKE '%PARALLELIZED%' OR UPPER (NAME) LIKE '%PX%';
```

Your output should resemble the following:

```
NAME                                                  VALUE
---------------------------------------------------- ----------
queries parallelized                                        347
DML statements parallelized                                   0
DDL statements parallelized                                   0
DFO trees parallelized                                      463
Parallel operations not downgraded                           28
Parallel operations downgraded to serial                     31
Parallel operations downgraded 75 to 99 pct                 252
Parallel operations downgraded 50 to 75 pct                 128
Parallel operations downgraded 25 to 50 pct                  43
Parallel operations downgraded 1 to 25 pct                   12
PX local messages sent                                    74548
PX local messages recv'd                                  74128
PX remote messages sent                                       0
PX remote messages recv'd                                     0
```

The following query shows the current wait state of each slave and QC process on the system:

```
SELECT px.SID "SID", p.PID, p.SPID "SPID", px.INST_ID "Inst",
       px.SERVER_GROUP "Group", px.SERVER_SET "Set",
       px.DEGREE "Degree", px.REQ_DEGREE "Req Degree", w.event "Wait Event"
FROM GV$SESSION s, GV$PX_SESSION px, GV$PROCESS p, GV$SESSION_WAIT w
WHERE s.sid (+) = px.sid AND s.inst_id (+) = px.inst_id AND
```

```
      s.sid = w.sid (+) AND s.inst_id = w.inst_id (+) AND
      s.paddr = p.addr (+) AND s.inst_id = p.inst_id (+)
ORDER BY DECODE(px.QCINST_ID,  NULL, px.INST_ID,  px.QCINST_ID), px.QCSID,
DECODE(px.SERVER_GROUP, NULL, 0, px.SERVER_GROUP), px.SERVER_SET, px.INST_ID;
```

Monitoring Operating System Statistics

There is considerable overlap between information available in Oracle and information available though operating system utilities (such as `sar` and `vmstat` on UNIX-based systems). Operating systems provide performance statistics on I/O, communication, CPU, memory and paging, scheduling, and synchronization primitives. The V$SESSTAT view provides the major categories of operating system statistics as well.

Typically, operating system information about I/O devices and semaphore operations is harder to map back to database objects and operations than is Oracle information. However, some operating systems have good visualization tools and efficient means of collecting the data.

Operating system information about CPU and memory usage is very important for assessing performance. Probably the most important statistic is CPU usage. The goal of low-level performance tuning is to become CPU bound on all CPUs. Once this is achieved, you can work at the SQL level to find an alternate plan that might be more I/O intensive but use less CPU.

Operating system memory and paging information is valuable for fine tuning the many system parameters that control how memory is divided among memory-intensive data warehouse subsystems like parallel communication, sort, and hash join.

Affinity and Parallel Operations

In a shared-disk cluster or MPP configuration, an instance of the Oracle Real Application Clusters is said to have affinity for a device if the device is directly accessed from the processors on which the instance is running. Similarly, an instance has affinity for a file if it has affinity for the devices on which the file is stored.

Determination of affinity may involve arbitrary determinations for files that are striped across multiple devices. Somewhat arbitrarily, an instance is said to have affinity for a tablespace (or a partition of a table or index within a tablespace) if the instance has affinity for the first file in the tablespace.

Oracle considers affinity when allocating work to parallel execution servers. The use of affinity for parallel execution of SQL statements is transparent to users.

Affinity and Parallel Queries

Affinity in parallel queries increases the speed of scanning data from disk by doing the scans on a processor that is near the data. This can provide a substantial performance increase for machines that do not naturally support shared disks.

The most common use of affinity is for a table or index partition to be stored in one file on one device. This configuration provides the highest availability by limiting the damage done by a device failure and makes the best use of partition-parallel index scans.

DSS customers might prefer to stripe table partitions over multiple devices (probably a subset of the total number of devices). This configuration enables some queries to prune the total amount of data being accessed using partitioning criteria and still obtain parallelism through rowid-range parallel table (partition) scans. If the devices are configured as a RAID, availability can still be very good. Even when used for DSS, indexes should probably be partitioned on individual devices.

Other configurations (for example, multiple partitions in one file striped over multiple devices) will yield correct query results, but you may need to use hints or explicitly set object attributes to select the correct DOP.

Affinity and Parallel DML

For parallel DML (inserts, updates, and deletes), affinity enhancements improve cache performance by routing the DML operation to the node that has affinity for the partition.

Affinity determines how to distribute the work among the set of instances or parallel execution servers to perform the DML operation in parallel. Affinity can improve performance of queries in several ways:

- For certain MPP architectures, Oracle uses device-to-node affinity information to determine on which nodes to spawn parallel execution servers (parallel process allocation) and which work granules (rowid ranges or partitions) to send to particular nodes (work assignment). Better performance is achieved by having nodes mainly access local devices, giving a better buffer cache hit ratio for every node and reducing the network overhead and I/O latency.
- For SMP, cluster, and MPP architectures, process-to-device affinity is used to achieve device isolation. This reduces the chances of having multiple parallel

execution servers accessing the same device simultaneously. This process-to-device affinity information is also used in implementing stealing between processes.

For partitioned tables and indexes, partition-to-node affinity information determines process allocation and work assignment. For shared-nothing MPP systems, Oracle Real Application Clusters tries to assign partitions to instances, taking the disk affinity of the partitions into account. For shared-disk MPP and cluster systems, partitions are assigned to instances in a round-robin manner.

Affinity is only available for parallel DML when running in an Oracle Real Application Clusters configuration. Affinity information which persists across statements improves buffer cache hit ratios and reduces block pings between instances.

Miscellaneous Parallel Execution Tuning Tips

This section contains some ideas for improving performance in a parallel execution environment and includes the following topics:

- Setting Buffer Cache Size for Parallel Operations
- Overriding the Default Degree of Parallelism
- Rewriting SQL Statements
- Creating and Populating Tables in Parallel
- Creating Temporary Tablespaces for Parallel Sort and Hash Join
- Executing Parallel SQL Statements
- Using EXPLAIN PLAN to Show Parallel Operations Plans
- Additional Considerations for Parallel DML
- Creating Indexes in Parallel
- Parallel DML Tips
- Incremental Data Loading in Parallel
- Using Hints with Query Optimization
- FIRST_ROWS(n) Hint
- Enabling Dynamic Sampling

Setting Buffer Cache Size for Parallel Operations

With the exception of parallel update and delete, parallel operations do not generally benefit from larger buffer cache sizes. Other parallel operations can benefit only if you increase the size of the buffer pool and thereby accommodate the inner table or index for a nested loop join.

Overriding the Default Degree of Parallelism

The default DOP is appropriate for reducing response time while guaranteeing use of CPU and I/O resources for any parallel operations.

If it is memory-bound, or if several concurrent parallel operations are running, you might want to decrease the default DOP.

Oracle uses the default DOP for tables that have PARALLEL attributed to them in the data dictionary or that have the PARALLEL hint specified. If a table does not have parallelism attributed to it, or has NO_PARALLEL (the default) attributed to it, and parallelism is not being forced through ALTER SESSION FORCE PARALLEL, then that table is never scanned in parallel. This override occurs regardless of the default DOP indicated by the number of CPUs, instances, and devices storing that table.

You can adjust the DOP by using the following guidelines:

- Modify the default DOP by changing the value for the PARALLEL_THREADS_PER_CPU parameter.
- Adjust the DOP either by using ALTER TABLE, ALTER SESSION, or by using hints.
- To increase the number of concurrent parallel operations, reduce the DOP, or set the parameter PARALLEL_ADAPTIVE_MULTI_USER to TRUE.

Rewriting SQL Statements

The most important issue for parallel execution is ensuring that all parts of the query plan that process a substantial amount of data execute in parallel. Use EXPLAIN PLAN to verify that all plan steps have an OTHER_TAG of PARALLEL_TO_PARALLEL, PARALLEL_TO_SERIAL, PARALLEL_COMBINED_WITH_PARENT, or PARALLEL_COMBINED_WITH_CHILD. Any other keyword (or null) indicates serial execution and a possible bottleneck. Also verify that such plan steps end in the operation PX SEND <partitioning type> node (for example, PX SEND HASH).

You can also use the `utlxplp.sql` script to present the EXPLAIN PLAN output with all relevant parallel information.

You can increase the optimizer's ability to generate parallel plans converting subqueries, especially correlated subqueries, into joins. Oracle can parallelize joins more efficiently than subqueries. This also applies to updates. See "Updating the Table in Parallel" on page 24-86 for more information.

Creating and Populating Tables in Parallel

Oracle cannot return results to a user process in parallel. If a query returns a large number of rows, execution of the query might indeed be faster. However, the user process can only receive the rows serially. To optimize parallel execution performance for queries that retrieve large result sets, use PARALLEL CREATE TABLE ... AS SELECT or direct-path INSERT to store the result set in the database. At a later time, users can view the result set serially.

Performing the SELECT in parallel does not influence the CREATE statement. If the CREATE is parallel, however, the optimizer tries to make the SELECT run in parallel also.

When combined with the NOLOGGING option, the parallel version of CREATE TABLE ... AS SELECT provides a very efficient intermediate table facility, for example:

```
CREATE TABLE summary PARALLEL NOLOGGING AS SELECT dim_1, dim_2 ...,
SUM (meas_1)
FROM facts GROUP BY dim_1, dim_2;
```

These tables can also be incrementally loaded with parallel INSERT. You can take advantage of intermediate tables using the following techniques:

- Common subqueries can be computed once and referenced many times. This can allow some queries against star schemas (in particular, queries without selective WHERE-clause predicates) to be better parallelized. Note that star queries with selective WHERE-clause predicates using the star-transformation technique can be effectively parallelized automatically without any modification to the SQL.
- Decompose complex queries into simpler steps in order to provide application-level checkpoint or restart. For example, a complex multitable join on a database 1 terabyte in size could run for dozens of hours. A failure during this query would mean starting over from the beginning. Using CREATE TABLE ... AS SELECT or PARALLEL INSERT AS SELECT, you can rewrite the query as a

sequence of simpler queries that run for a few hours each. If a system failure occurs, the query can be restarted from the last completed step.

- Implement manual parallel deletes efficiently by creating a new table that omits the unwanted rows from the original table, and then dropping the original table. Alternatively, you can use the convenient parallel delete feature, which directly deletes rows from the original table.
- Create summary tables for efficient multidimensional drill-down analysis. For example, a summary table might store the sum of revenue grouped by month, brand, region, and salesman.
- Reorganize tables, eliminating chained rows, compressing free space, and so on, by copying the old table to a new table. This is much faster than export/import and easier than reloading.

Be sure to use the DBMS_STATS package on newly created tables. Also consider creating indexes. To avoid I/O bottlenecks, specify a tablespace with at least as many devices as CPUs. To avoid fragmentation in allocating space, the number of files in a tablespace should be a multiple of the number of CPUs. See Chapter 4, "Hardware and I/O Considerations in Data Warehouses", for more information about bottlenecks.

Creating Temporary Tablespaces for Parallel Sort and Hash Join

For optimal space management performance, you should use locally managed temporary tablespaces. The following is an example:

```
CREATE TEMPORARY TABLESPACE TStemp TEMPFILE '/dev/D31'
SIZE 4096MB REUSE EXTENT MANAGEMENT LOCAL UNIFORM SIZE 10m;
```

You can associate temporary tablespaces to a database by issuing a statement such as:

```
ALTER DATABASE TEMPORARY TABLESPACE TStemp;
```

Once this is done, explicit assignment of users to tablespaces is not needed.

Size of Temporary Extents

When using a locally managed temporary tablespace, extents are all the same size because this helps avoid fragmentation. As a general rule, temporary extents should be smaller than permanent extents because there are more demands for temporary space, and parallel processes or other operations running concurrently must share the temporary tablespace. Normally, temporary extents should be in the range of

1MB to 10MB. Once you allocate an extent, it is available for the duration of an operation. If you allocate a large extent but only need to use a small amount of space, the unused space in the extent is unavailable.

At the same time, temporary extents should be large enough that processes do not have to wait for space. Temporary tablespaces use less overhead than permanent tablespaces when allocating and freeing a new extent. However, obtaining a new temporary extent still requires the overhead of acquiring a latch and searching through the SGA structures, as well as SGA space consumption for the sort extent pool.

See *Oracle Database Performance Tuning Guide* for information regarding locally-managed temporary tablespaces.

Executing Parallel SQL Statements

After analyzing your tables and indexes, you should see performance improvements based on the DOP used.

As a general process, you should start with simple parallel operations and evaluate their total I/O throughput with a `SELECT COUNT(*) FROM facts` statement. Then, evaluate total CPU power by adding a complex `WHERE` clause to the statement. An I/O imbalance might suggest a better physical database layout. After you understand how simple scans work, add aggregation, joins, and other operations that reflect individual aspects of the overall workload. In particular, you should look for bottlenecks.

Besides query performance, you should also monitor parallel load, parallel index creation, and parallel DML, and look for good utilization of I/O and CPU resources.

Using EXPLAIN PLAN to Show Parallel Operations Plans

Use the `EXPLAIN PLAN` statement to see the execution plans for parallel queries. `EXPLAIN PLAN` output shows optimizer information in the `COST`, `BYTES`, and `CARDINALITY` columns. You can also use the `utlxplp.sql` script to present the `EXPLAIN PLAN` output with all relevant parallel information.

There are several ways to optimize the parallel execution of join statements. You can alter system configuration, adjust parameters as discussed earlier in this chapter, or use hints, such as the `DISTRIBUTION` hint.

The key points when using `EXPLAIN PLAN` are to:

- Verify optimizer selectivity estimates. If the optimizer thinks that only one row will be produced from a query, it tends to favor using a nested loop. This could

be an indication that the tables are not analyzed or that the optimizer has made an incorrect estimate about the correlation of multiple predicates on the same table. A hint may be required to force the optimizer to use another join method. Consequently, if the plan says only one row is produced from any particular stage and this is incorrect, consider hints or gather statistics.

- Use hash join on low cardinality join keys. If a join key has few distinct values, then a hash join may not be optimal. If the number of distinct values is less than the DOP, then some parallel query servers may be unable to work on the particular query.
- Consider data skew. If a join key involves excessive data skew, a hash join may require some parallel query servers to work more than others. Consider using a hint to cause a `BROADCAST` distribution method if the optimizer did not choose it. Note that the optimizer will consider the `BROADCAST` distribution method only if the `OPTIMIZER_FEATURES_ENABLE` is set to 9.0.2 or higher. See "V$PQ_TQSTAT" on page 24-67 for further details.

Additional Considerations for Parallel DML

When you want to refresh your data warehouse database using parallel insert, update, or delete on a data warehouse, there are additional issues to consider when designing the physical database. These considerations do not affect parallel execution operations. These issues are:

- PDML and Direct-Path Restrictions
- Limitation on the Degree of Parallelism
- Using Local and Global Striping
- Increasing INITRANS
- Limitation on Available Number of Transaction Free Lists for Segments
- Using Multiple Archivers
- Database Writer Process (DBWn) Workload
- [NO]LOGGING Clause

PDML and Direct-Path Restrictions

If a parallel restriction is violated, the operation is simply performed serially. If a direct-path `INSERT` restriction is violated, then the `APPEND` hint is ignored and a conventional insert is performed. No error message is returned.

Limitation on the Degree of Parallelism

If you are performing parallel UPDATE, MERGE, or DELETE operations, the DOP is equal to or less than the number of partitions in the table.

For tables created prior to Oracle database release version 9.0.1 or tables that do not have the PDML itl invariant property, the previous PDML restriction applies to the calculation of the DOP. To see what tables do not have this property, issue the following statement:

```
SELECT u.name, o.name FROM obj$ o, tab$ t, user$ u
WHERE o.obj# = t.obj# AND o.owner# = u.user# AND
  bitand(t.property,536870912) != 536870912;
```

Using Local and Global Striping

Parallel updates and deletes work only on partitioned tables. They can generate a high number of random I/O requests during index maintenance.

For local index maintenance, local striping is most efficient in reducing I/O contention because one server process only goes to its own set of disks and disk controllers. Local striping also increases availability in the event of one disk failing.

For global index maintenance (partitioned or nonpartitioned), globally striping the index across many disks and disk controllers is the best way to distribute the number of I/Os.

Increasing INITRANS

If you have global indexes, a global index segment and global index blocks are shared by server processes of the same parallel DML statement. Even if the operations are not performed against the same row, the server processes can share the same index blocks. Each server transaction needs one transaction entry in the index block header before it can make changes to a block. Therefore, in the CREATE INDEX or ALTER INDEX statements, you should set INITRANS, the initial number of transactions allocated within each data block, to a large value, such as the maximum DOP against this index.

Limitation on Available Number of Transaction Free Lists for Segments

There is a limitation on the available number of transaction free lists for segments in dictionary-managed tablespaces. Once a segment has been created, the number of process and transaction free lists is fixed and cannot be altered. If you specify a large number of process free lists in the segment header, you might find that this limits the number of transaction free lists that are available. You can abate this

limitation the next time you re-create the segment header by decreasing the number of process free lists; this leaves more room for transaction free lists in the segment header.

For UPDATE and DELETE operations, each server process can require its own transaction free list. The parallel DML DOP is thus effectively limited by the smallest number of transaction free lists available on the table and on any of the global indexes the DML statement must maintain. For example, if the table has 25 transaction free lists and the table has two global indexes, one with 50 transaction free lists and one with 30 transaction free lists, the DOP is limited to 25. If the table had had 40 transaction free lists, the DOP would have been limited to 30.

The FREELISTS parameter of the STORAGE clause is used to set the number of process free lists. By default, no process free lists are created.

The default number of transaction free lists depends on the block size. For example, if the number of process free lists is not set explicitly, a 4 KB block has about 80 transaction free lists by default. The minimum number of transaction free lists is 25.

Using Multiple Archivers

Parallel DDL and parallel DML operations can generate a large amount of redo logs. A single ARCH process to archive these redo logs might not be able to keep up. To avoid this problem, you can spawn multiple archiver processes. This can be done manually or by using a job queue.

Database Writer Process (DBWn) Workload

Parallel DML operations dirty a large number of data, index, and undo blocks in the buffer cache during a short period of time. For example, suppose you see a high number of `free_buffer_waits` after querying the V$SYSTEM_EVENT view, as in the following syntax:

```
SELECT TOTAL_WAITS FROM V$SYSTEM_EVENT WHERE EVENT = 'FREE BUFFER WAITS';
```

In this case, you should consider increasing the DBW*n* processes. If there are no waits for free buffers, the query will not return any rows.

[NO]LOGGING Clause

The [NO]LOGGING clause applies to tables, partitions, tablespaces, and indexes. Virtually no log is generated for certain operations (such as direct-path INSERT) if the NOLOGGING clause is used. The NOLOGGING attribute is not specified at the INSERT statement level but is instead specified when using the ALTER or CREATE statement for a table, partition, index, or tablespace.

When a table or index has NOLOGGING set, neither parallel nor serial direct-path INSERT operations generate redo logs. Processes running with the NOLOGGING option set run faster because no redo is generated. However, after a NOLOGGING operation against a table, partition, or index, if a media failure occurs before a backup is taken, then all tables, partitions, and indexes that have been modified might be corrupted.

Direct-path INSERT operations (except for dictionary updates) never generate redo logs. The NOLOGGING attribute does not affect undo, only redo. To be precise, NOLOGGING allows the direct-path INSERT operation to generate a negligible amount of redo (range-invalidation redo, as opposed to full image redo).

For backward compatibility, [UN]RECOVERABLE is still supported as an alternate keyword with the CREATE TABLE statement. This alternate keyword might not be supported, however, in future releases.

At the tablespace level, the logging clause specifies the default logging attribute for all tables, indexes, and partitions created in the tablespace. When an existing tablespace logging attribute is changed by the ALTER TABLESPACE statement, then all tables, indexes, and partitions created after the ALTER statement will have the new logging attribute; existing ones will not change their logging attributes. The tablespace-level logging attribute can be overridden by the specifications at the table, index, or partition level.

The default logging attribute is LOGGING. However, if you have put the database in NOARCHIVELOG mode, by issuing ALTER DATABASE NOARCHIVELOG, then all operations that can be done without logging will not generate logs, regardless of the specified logging attribute.

Creating Indexes in Parallel

Multiple processes can work together simultaneously to create an index. By dividing the work necessary to create an index among multiple server processes, Oracle Database can create the index more quickly than if a single server process created the index sequentially.

Parallel index creation works in much the same way as a table scan with an ORDER BY clause. The table is randomly sampled and a set of index keys is found that equally divides the index into the same number of pieces as the DOP. A first set of query processes scans the table, extracts key-rowid pairs, and sends each pair to a process in a second set of query processes based on key. Each process in the second set sorts the keys and builds an index in the usual fashion. After all index pieces are built, the parallel coordinator simply concatenates the pieces (which are ordered) to form the final index.

Parallel local index creation uses a single server set. Each server process in the set is assigned a table partition to scan and for which to build an index partition. Because half as many server processes are used for a given DOP, parallel local index creation can be run with a higher DOP. However, the DOP is restricted to be less than or equal to the number of index partitions you wish to create. To avoid this, you can use the DBMS_PCLXUTIL package.

You can optionally specify that no redo and undo logging should occur during index creation. This can significantly improve performance but temporarily renders the index unrecoverable. Recoverability is restored after the new index is backed up. If your application can tolerate a window where recovery of the index requires it to be re-created, then you should consider using the NOLOGGING clause.

The PARALLEL clause in the CREATE INDEX statement is the only way in which you can specify the DOP for creating the index. If the DOP is not specified in the parallel clause of CREATE INDEX, then the number of CPUs is used as the DOP. If there is no PARALLEL clause, index creation is done serially.

When creating an index in parallel, the STORAGE clause refers to the storage of each of the subindexes created by the query server processes. Therefore, an index created with an INITIAL of 5 MB and a DOP of 12 consumes at least 60 MB of storage during index creation because each process starts with an extent of 5 MB. When the query coordinator process combines the sorted subindexes, some of the extents might be trimmed, and the resulting index might be smaller than the requested 60 MB.

When you add or enable a UNIQUE or PRIMARY KEY constraint on a table, you cannot automatically create the required index in parallel. Instead, manually create an index on the desired columns, using the CREATE INDEX statement and an appropriate PARALLEL clause, and then add or enable the constraint. Oracle then uses the existing index when enabling or adding the constraint.

Multiple constraints on the same table can be enabled concurrently and in parallel if all the constraints are already in the ENABLE NOVALIDATE state. In the following example, the ALTER TABLE ... ENABLE CONSTRAINT statement performs the table scan that checks the constraint in parallel:

```
CREATE TABLE a (a1 NUMBER CONSTRAINT ach CHECK (a1 > 0) ENABLE NOVALIDATE)
PARALLEL;
INSERT INTO a values (1);
COMMIT;
ALTER TABLE a ENABLE CONSTRAINT ach;
```

Parallel DML Tips

This section provides an overview of parallel DML functionality. The topics covered include:

- Parallel DML Tip 1: INSERT
- Parallel DML Tip 2: Direct-Path INSERT
- Parallel DML Tip 3: Parallelizing INSERT, MERGE, UPDATE, and DELETE

Parallel DML Tip 1: INSERT

Oracle INSERT functionality can be summarized as follows:

Table 24–5 Summary of INSERT Features

Insert Type	Parallel	Serial	NOLOGGING
Conventional	No	Yes	No
Direct-path INSERT (APPEND)	Yes, but requires: ALTER SESSION ENABLE PARALLEL DML Table PARALLEL attribute or PARALLEL hint APPEND hint (optional)	Yes, but requires: APPEND hint	Yes, but requires: NOLOGGING attribute set for partition or table

If parallel DML is enabled and there is a PARALLEL hint or PARALLEL attribute set for the table in the data dictionary, then inserts are parallel and appended, unless a restriction applies. If either the PARALLEL hint or PARALLEL attribute is missing, the insert is performed serially.

Parallel DML Tip 2: Direct-Path INSERT

The append mode is the default during a parallel insert: data is always inserted into a new block which is allocated to the table. Therefore the APPEND hint is optional. You should use append mode to increase the speed of INSERT operations, but not when space utilization needs to be optimized. You can use NOAPPEND to override append mode.

The APPEND hint applies to both serial and parallel insert: even serial inserts are faster if you use this hint. APPEND, however, does require more space and locking overhead.

You can use NOLOGGING with APPEND to make the process even faster. NOLOGGING means that no redo log is generated for the operation. NOLOGGING is never the default; use it when you wish to optimize performance. It should not normally be

used when recovery is needed for the table or partition. If recovery is needed, be sure to take a backup immediately after the operation. Use the ALTER TABLE [NO]LOGGING statement to set the appropriate value.

Parallel DML Tip 3: Parallelizing INSERT, MERGE, UPDATE, and DELETE

When the table or partition has the PARALLEL attribute in the data dictionary, that attribute setting is used to determine parallelism of INSERT, UPDATE, and DELETE statements as well as queries. An explicit PARALLEL hint for a table in a statement overrides the effect of the PARALLEL attribute in the data dictionary.

You can use the NO_PARALLEL hint to override a PARALLEL attribute for the table in the data dictionary. In general, hints take precedence over attributes.

DML operations are considered for parallelization only if the session is in a PARALLEL DML enabled mode. (Use ALTER SESSION ENABLE PARALLEL DML to enter this mode.) The mode does not affect parallelization of queries or of the query portions of a DML statement.

Parallelizing INSERT ... SELECT In the INSERT ... SELECT statement you can specify a PARALLEL hint after the INSERT keyword, in addition to the hint after the SELECT keyword. The PARALLEL hint after the INSERT keyword applies to the INSERT operation only, and the PARALLEL hint after the SELECT keyword applies to the SELECT operation only. Thus, parallelism of the INSERT and SELECT operations are independent of each other. If one operation cannot be performed in parallel, it has no effect on whether the other operation can be performed in parallel.

The ability to parallelize inserts causes a change in existing behavior if the user has explicitly enabled the session for parallel DML and if the table in question has a PARALLEL attribute set in the data dictionary entry. In that case, existing INSERT ... SELECT statements that have the select operation parallelized can also have their insert operation parallelized.

If you query multiple tables, you can specify multiple SELECT PARALLEL hints and multiple PARALLEL attributes.

Example 24–7 Parallelizing INSERT ... SELECT

Add the new employees who were hired after the acquisition of ACME.

```
INSERT /*+ PARALLEL(EMP) */ INTO employees
SELECT /*+ PARALLEL(ACME_EMP) */ *  FROM ACME_EMP;
```

The APPEND keyword is not required in this example because it is implied by the PARALLEL hint.

Parallelizing UPDATE and DELETE The PARALLEL hint (placed immediately after the UPDATE or DELETE keyword) applies not only to the underlying scan operation, but also to the UPDATE or DELETE operation. Alternatively, you can specify UPDATE or DELETE parallelism in the PARALLEL clause specified in the definition of the table to be modified.

If you have explicitly enabled parallel DML for the session or transaction, UPDATE or DELETE statements that have their query operation parallelized can also have their UPDATE or DELETE operation parallelized. Any subqueries or updatable views in the statement can have their own separate PARALLEL hints or clauses, but these parallel directives do not affect the decision to parallelize the update or delete. If these operations cannot be performed in parallel, it has no effect on whether the UPDATE or DELETE portion can be performed in parallel.

Tables must be partitioned in order to support parallel UPDATE and DELETE.

Example 24–8 Parallelizing UPDATE and DELETE

Give a 10 percent salary raise to all clerks in Dallas.

```
UPDATE /*+ PARALLEL(EMP) */ employees
SET SAL=SAL * 1.1 WHERE JOB='CLERK' AND DEPTNO IN
  (SELECT DEPTNO FROM DEPT WHERE LOCATION='DALLAS');
```

The PARALLEL hint is applied to the UPDATE operation as well as to the scan.

Example 24–9 Parallelizing UPDATE and DELETE

Remove all products in the grocery category because the grocery business line was recently spun off into a separate company.

```
DELETE /*+ PARALLEL(PRODUCTS) */ FROM PRODUCTS
WHERE PRODUCT_CATEGORY ='GROCERY';
```

Again, the parallelism is applied to the scan as well as UPDATE operation on table `employees`.

Incremental Data Loading in Parallel

Parallel DML combined with the updatable join views facility provides an efficient solution for refreshing the tables of a data warehouse system. To refresh tables is to update them with the differential data generated from the OLTP production system.

In the following example, assume that you want to refresh a table named `customer` that has columns `c_key`, `c_name`, and `c_addr`. The differential data

contains either new rows or rows that have been updated since the last refresh of the data warehouse. In this example, the updated data is shipped from the production system to the data warehouse system by means of ASCII files. These files must be loaded into a temporary table, named `diff_customer`, before starting the refresh process. You can use SQL*Loader with both the parallel and direct options to efficiently perform this task. You can use the APPEND hint when loading in parallel as well.

Once `diff_customer` is loaded, the refresh process can be started. It can be performed in two phases or by merging in parallel, as demonstrated in the following:

- Updating the Table in Parallel
- Inserting the New Rows into the Table in Parallel
- Merging in Parallel

Updating the Table in Parallel

The following statement is a straightforward SQL implementation of the update using subqueries:

```
UPDATE customers SET(c_name, c_addr) = (SELECT c_name, c_addr
FROM diff_customer WHERE diff_customer.c_key = customer.c_key)
WHERE c_key IN(SELECT c_key FROM diff_customer);
```

Unfortunately, the two subqueries in this statement affect performance.

An alternative is to rewrite this query using updatable join views. To do this, you must first add a primary key constraint to the `diff_customer` table to ensure that the modified columns map to a key-preserved table:

```
CREATE UNIQUE INDEX diff_pkey_ind ON diff_customer(c_key) PARALLEL NOLOGGING;
ALTER TABLE diff_customer ADD PRIMARY KEY (c_key);
```

You can then update the `customers` table with the following SQL statement:

```
UPDATE /*+ PARALLEL(cust_joinview) */
(SELECT /*+ PARALLEL(customers) PARALLEL(diff_customer) */
CUSTOMER.c_name AS c_name CUSTOMER.c_addr AS c_addr,
diff_customer.c_name AS c_newname, diff_customer.c_addr AS c_newaddr
   WHERE customers.c_key = diff_customer.c_key) cust_joinview
   SET c_name = c_newname, c_addr = c_newaddr;
```

The base scans feeding the join view cust_joinview are done in parallel. You can then parallelize the update to further improve performance, but only if the customers table is partitioned.

Inserting the New Rows into the Table in Parallel

The last phase of the refresh process consists of inserting the new rows from the diff_customer temporary table to the customer table. Unlike the update case, you cannot avoid having a subquery in the INSERT statement:

```
INSERT /*+PARALLEL(customers)*/ INTO customers SELECT * FROM diff_customer s);
```

However, you can guarantee that the subquery is transformed into an anti-hash join by using the HASH_AJ hint. Doing so enables you to use parallel INSERT to execute the preceding statement efficiently. Parallel INSERT is applicable even if the table is not partitioned.

Merging in Parallel

You can combine updates and inserts into one statement, commonly known as a **merge**. The following statement achieves the same result as all of the statements in "Updating the Table in Parallel" on page 24-86 and "Inserting the New Rows into the Table in Parallel" on page 24-87:

```
MERGE INTO customers USING diff_customer
ON (diff_customer.c_key = customer.c_key) WHEN MATCHED THEN
  UPDATE SET (c_name, c_addr) = (SELECT c_name, c_addr
  FROM diff_customer WHERE diff_customer.c_key = customers.c_key)
WHEN NOT MATCHED THEN
  INSERT VALUES (diff_customer.c_key,diff_customer.c_data);
```

Using Hints with Query Optimization

Query optimization is a sophisticated approach to finding the best execution plan for SQL statements. Oracle automatically uses query optimization with parallel execution.

You must use the DBMS_STATS package to gather current statistics for cost-based optimization. In particular, tables used in parallel should always be analyzed. Always keep your statistics current by using the DBMS_STATS package. Failure to do so may result in degraded execution performance due to non-optimal execution plans.

Use discretion in employing hints. If used, hints should come as a final step in tuning and only when they demonstrate a necessary and significant performance

advantage. In such cases, begin with the execution plan recommended by query optimization, and go on to test the effect of hints only after you have quantified your performance expectations. Remember that hints are powerful. If you use them and the underlying data changes, you might need to change the hints. Otherwise, the effectiveness of your execution plans might deteriorate.

FIRST_ROWS(n) Hint

The FIRST_ROWS(n) hint enables the optimizer to use a new optimization mode to optimize the query to return *n* rows in the shortest amount of time. Oracle Corporation recommends that you use this new hint in place of the old FIRST_ROWS hint for online queries because the new optimization mode may improve the response time compared to the old optimization mode.

Use the FIRST_ROWS(n) hint in cases where you want the first *n* number of rows in the shortest possible time. For example, to obtain the first 10 rows in the shortest possible time, use the hint as follows:

```
SELECT /*+ FIRST_ROWS(10) */ article_id
FROM articles_tab WHERE CONTAINS(article, 'Oracle')>0 ORDER BY pub_date DESC;
```

Enabling Dynamic Sampling

Dynamic sampling allows Oracle to derive more accurate statistics and thereby improve query performance when statistics do not exist or are out of date. This is particularly useful in data warehousing environments or when you expect long transactions or queries. In these situations, making sure that Oracle uses the best execution plan is important. Dynamic sampling does, however, have a small cost, so you should use it when that cost is likely to be a small fraction of the total execution time.

If you enable dynamic statistic sampling, Oracle determines at compile time whether a query would benefit from dynamic sampling. If so, a recursive SQL statement is issued to scan a small, random sample of the table's blocks, and to apply the relevant single table predicates to estimate predicate selectivities. Relevant table, index and column statistics are also estimated. More accurate selectivity and statistics estimates allow the optimizer to produce better performing plans.

Dynamic sampling is controlled with the initialization parameter OPTIMIZER_DYNAMIC_SAMPLING, which can be set to a value between 0 and 10, inclusive. Increasing the value of the parameter will result in more aggressive application of

dynamic sampling, in terms of both the type (unanalyzed / analyzed) of tables sampled and the amount of I/O spent on sampling.

Oracle also provides the table-specific hint `DYNAMIC_SAMPLING`. If the table name is omitted, the hint is considered cursor-level. The table-level hint forces dynamic sampling for the table.

See *Oracle Database Performance Tuning Guide* for more information regarding dynamic sampling.

Glossary

additive

Describes a fact (or **measure**) that can be summarized through addition. An additive fact is the most common type of fact. Examples include sales, cost, and profit. Contrast with **nonadditive** and **semi-additive**.

See Also: fact

advisor

See: SQLAccess Advisor.

aggregate

Summarized data. For example, unit sales of a particular product could be aggregated by day, month, quarter and yearly sales.

aggregation

The process of consolidating data values into a single value. For example, sales data could be collected on a daily basis and then be aggregated to the week level, the week data could be aggregated to the month level, and so on. The data can then be referred to as aggregate data. **Aggregation** is synonymous with **summarization**, and aggregate data is synonymous with summary data.

ancestor

A value at any level higher than a given value in a hierarchy. For example, in a Time dimension, the value 1999 might be the ancestor of the values Q1-99 and Jan-99.

See Also: hierarchy and level

attribute

A descriptive characteristic of one or more levels. For example, the product dimension for a clothing manufacturer might contain a level called item, one of whose attributes is color. Attributes represent logical groupings that enable end users to select data based on like characteristics.

Note that in relational modeling, an attribute is defined as a characteristic of an entity. In Oracle Database 10*g*, an attribute is a column in a dimension that characterizes elements of a single level.

cardinality

From an OLTP perspective, this refers to the number of rows in a table. From a data warehousing perspective, this typically refers to the number of distinct values in a column. For most data warehouse DBAs, a more important issue is the **degree of cardinality**.

See Also: degree of cardinality

change set

A set of logically grouped change data that is transactionally consistent. It contains one or more change tables.

change table

A relational table that contains change data for a single source table. To Change Data Capture subscribers, a change table is known as a publication.

child

A value at the level under a given value in a hierarchy. For example, in a Time dimension, the value Jan-99 might be the child of the value Q1-99. A value can be a child for more than one parent if the child value belongs to multiple hierarchies.

See Also:

- hierarchy
- level
- parent

cleansing

The process of resolving inconsistencies and fixing the anomalies in source data, typically as part of the ETL process.

See Also: ETL

Common Warehouse Metadata (CWM)

A repository standard used by Oracle data warehousing, and decision support. The CWM repository schema is a standalone product that other products can share—each product owns only the objects within the CWM repository that it creates.

cross product

A procedure for combining the elements in multiple sets. For example, given two columns, each element of the first column is matched with every element of the second column. A simple example is illustrated as follows:

```
Col1   Col2   Cross Product
----   ----   -------------
a      c      ac
b      d      ad
              bc
              bd
```

Cross products are performed when grouping sets are concatenated, as described in Chapter 20, "SQL for Aggregation in Data Warehouses".

data mart

A data warehouse that is designed for a particular line of business, such as sales, marketing, or finance. In a dependent data mart, the data can be derived from an enterprise-wide data warehouse. In an independent data mart, data can be collected directly from sources.

See Also: data warehouse

data source

A database, application, repository, or file that contributes data to a warehouse.

data warehouse

A relational database that is designed for query and analysis rather than transaction processing. A data warehouse usually contains historical data that is derived from transaction data, but it can include data from other sources. It separates analysis workload from transaction workload and enables a business to consolidate data from several sources.

In addition to a relational database, a data warehouse environment often consists of an ETL solution, an OLAP engine, client analysis tools, and other applications that manage the process of gathering data and delivering it to business users.

See Also: ETL and online analytical processing (OLAP)

degree of cardinality

The number of unique values of a column divided by the total number of rows in the table. This is particularly important when deciding which indexes to build. You typically want to use bitmap indexes on low degree of cardinality columns and B-tree indexes on high degree of cardinality columns. As a general rule, a cardinality of under 1% makes a good candidate for a bitmap index.

denormalize

The process of allowing redundancy in a table. Contrast with **normalize**.

derived fact (or measure)

A fact (or measure) that is generated from existing data using a mathematical operation or a data transformation. Examples include averages, totals, percentages, and differences.

detail

See: fact table.

detail table

See: fact table.

dimension

The term dimension is commonly used in two ways:

- A general term for any characteristic that is used to specify the members of a data set. The 3 most common dimensions in sales-oriented data warehouses are time, geography, and product. Most dimensions have hierarchies.
- An object defined in a database to enable queries to navigate dimensions. In Oracle Database 10*g*, a dimension is a database object that defines hierarchical (parent/child) relationships between pairs of column sets. In Oracle Express, a dimension is a database object that consists of a list of values.

dimension table

Dimension tables describe the business entities of an enterprise, represented as hierarchical, categorical information such as time, departments, locations, and products. Dimension tables are sometimes called lookup or reference tables.

dimension value

One element in the list that makes up a dimension. For example, a computer company might have dimension values in the product dimension called LAPPC and DESKPC. Values in the geography dimension might include Boston and Paris. Values in the time dimension might include MAY96 and JAN97.

drill

To navigate from one item to a set of related items. Drilling typically involves navigating up and down through the levels in a hierarchy. When selecting data, you can expand or collapse a hierarchy by drilling down or up in it, respectively.

See Also: drill down and drill up

drill down

To expand the view to include child values that are associated with parent values in the hierarchy.

See Also: drill and drill up

drill up

To collapse the list of descendant values that are associated with a parent value in the hierarchy.

element

An object or process. For example, a dimension is an object, a mapping is a process, and both are elements.

entity

Entity is used in database modeling. In relational databases, it typically maps to a table.

ETL

Extraction, transformation, and loading. ETL refers to the methods involved in accessing and manipulating source data and loading it into a data warehouse. The order in which these processes are performed varies.

Note that ETT (extraction, transformation, transportation) and ETM (extraction, transformation, move) are sometimes used instead of ETL.

See Also:

- data warehouse
- extraction
- transformation
- transportation

extraction

The process of taking data out of a source as part of an initial phase of ETL.

See Also: ETL

fact

Data, usually numeric and additive, that can be examined and analyzed. Examples include sales, cost, and profit. **Fact** and **measure** are synonymous; fact is more commonly used with relational environments, measure is more commonly used with multidimensional environments.

See Also: derived fact (or measure)

fact table

A table in a star schema that contains facts. A fact table typically has two types of columns: those that contain facts and those that are foreign keys to dimension tables. The primary key of a fact table is usually a composite key that is made up of all of its foreign keys.

A fact table might contain either detail level facts or facts that have been aggregated (fact tables that contain aggregated facts are often instead called summary tables). A fact table usually contains facts with the same level of aggregation.

fast refresh

An operation that applies only the data changes to a materialized view, thus eliminating the need to rebuild the materialized view from scratch.

file-to-table mapping

Maps data from flat files to tables in the warehouse.

hierarchy

A logical structure that uses ordered levels as a means of organizing data. A hierarchy can be used to define data aggregation; for example, in a time dimension, a hierarchy might be used to aggregate data from the Month level to the Quarter level to the Year level. Hierarchies can be defined in Oracle as part of the dimension object. A hierarchy can also be used to define a navigational drill path, regardless of whether the levels in the hierarchy represent aggregated totals.

See Also: dimension and level

high boundary

The newest row in a subscription window.

level

A position in a hierarchy. For example, a time dimension might have a hierarchy that represents data at the Month, Quarter, and Year levels.

See Also: hierarchy

level value table

A database table that stores the values or data for the levels you created as part of your dimensions and hierarchies.

low boundary

The oldest row in a subscription window.

mapping

The definition of the relationship and data flow between source and target objects.

materialized view

A pre-computed table comprising aggregated or joined data from fact and possibly dimension tables. Also known as a summary or aggregate table.

measure

See: fact.

metadata

Data that describes data and other structures, such as objects, business rules, and processes. For example, the schema design of a data warehouse is typically stored in a repository as metadata, which is used to generate scripts used to build and populate the data warehouse. A repository contains metadata.

Examples include: for data, the definition of a source to target transformation that is used to generate and populate the data warehouse; for information, definitions of tables, columns and associations that are stored inside a relational modeling tool; for business rules, discount by 10 percent after selling 1,000 items.

model

An object that represents something to be made. A representative style, plan, or design. Metadata that defines the structure of the data warehouse.

nonadditive

Describes a fact (or measure) that cannot be summarized through addition. An example includes Average. Contrast with **additive** and **semi-additive**.

normalize

In a relational database, the process of removing redundancy in data by separating the data into multiple tables. Contrast with denormalize.

The process of removing redundancy in data by separating the data into multiple tables.

OLAP

See: online analytical processing (OLAP).

online analytical processing (OLAP)

OLAP functionality is characterized by dynamic, multidimensional analysis of historical data, which supports activities such as the following:

- Calculating across dimensions and through hierarchies
- Analyzing trends
- Drilling up and down through hierarchies
- Rotating to change the dimensional orientation

OLAP tools can run against a multidimensional database or interact directly with a relational database.

OLTP

See: online transaction processing (OLTP).

online transaction processing (OLTP)

Online transaction processing. OLTP systems are optimized for fast and reliable transaction handling. Compared to data warehouse systems, most OLTP interactions will involve a relatively small number of rows, but a larger group of tables.

parallelism

Breaking down a task so that several processes do part of the work. When multiple CPUs each do their portion simultaneously, very large performance gains are possible.

parallel execution

Breaking down a task so that several processes do part of the work. When multiple CPUs each do their portion simultaneously, very large performance gains are possible.

parent

A value at the level above a given value in a hierarchy. For example, in a Time dimension, the value `Q1-99` might be the parent of the value `Jan-99`.

See Also:

- child
- hierarchy
- level

partition

Very large tables and indexes can be difficult and time-consuming to work with. To improve manageability, you can break your tables and indexes into smaller pieces called partitions.

pivoting

A transformation where each record in an input stream is converted to many records in the appropriate table in the data warehouse. This is particularly important when taking data from nonrelational databases.

publication

A relational table that contains change data for a single source table. Change Data Capture publishers refer to a publication as a change table.

publication ID

A publication ID is a unique numeric value that Change Data Capture assigns to each change table defined by a publisher.

publisher

Usually a database administrator who is in charge of creating and maintaining schema objects that make up the Change Data Capture system.

refresh

The mechanism whereby materialized views are changed to reflect new data.

schema

A collection of related database objects. Relational schemas are grouped by database user ID and include tables, views, and other objects. The sample schemas sh are used throughout this Guide.

See Also: snowflake schema and star schema

semi-additive

Describes a fact (or measure) that can be summarized through addition along some, but not all, dimensions. Examples include headcount and on hand stock. Contrast with **additive** and **nonadditive**.

slice and dice

This is an informal term referring to data retrieval and manipulation. We can picture a data warehouse as a cube of data, where each axis of the cube represents a dimension. To "slice" the data is to retrieve a piece (a slice) of the cube by specifying measures and values for some or all of the dimensions. When we retrieve a data slice, we may also move and reorder its columns and rows as if we had diced the slice into many small pieces. A system with good slicing and dicing makes it easy to navigate through large amounts of data.

snowflake schema

A type of star schema in which the dimension tables are partly or fully normalized.

See Also: schema and star schema

source

A database, application, file, or other storage facility from which the data in a data warehouse is derived.

source system

A database, application, file, or other storage facility from which the data in a data warehouse is derived.

source tables

The tables in a source database.

SQLAccess Advisor

The SQLAccess Advisor helps you achieve your performance goals by recommending the proper set of materialized views, materialized view logs, and indexes for a given workload. It is a GUI in Oracle Enterprise Manager, and has similar capabilities to the `DBMS_ADVISOR` package.

staging area

A place where data is processed before entering the warehouse.

staging file

A file used when data is processed before entering the warehouse.

star query

A join between a fact table and a number of dimension tables. Each dimension table is joined to the fact table using a primary key to foreign key join, but the dimension tables are not joined to each other.

star schema

A relational schema whose design represents a multidimensional data model. The star schema consists of one or more fact tables and one or more dimension tables that are related through foreign keys.

See Also: schema and snowflake schema

subject area

A classification system that represents or distinguishes parts of an organization or areas of knowledge. A data mart is often developed to support a subject area such as sales, marketing, or geography.

See Also: data mart

subscribers

Consumers of the published change data. These are normally applications.

subscription

A mechanism for Change Data Capture subscribers that controls access to the change data from one or more source tables of interest within a single change set. A subscription contains one or more subscriber views.

subscription window

A mechanism that defines the range of rows in a Change Data Capture publication that the subscriber can currently see in subscriber views.

summary

See: materialized view.

Summary Advisor

Replaced by the SQLAccess Advisor. See: SQLAccess Advisor.

target

Holds the intermediate or final results of any part of the ETL process. The target of the entire ETL process is the data warehouse.

See Also: data warehouse and ETL

third normal form (3NF)

A classical relational database modeling technique that minimizes data redundancy through normalization.

third normal form schema

A schema that uses the same kind of normalization as typically found in an OLTP system. Third normal form schemas are sometimes chosen for large data

warehouses, especially environments with significant data loading requirements that are used to feed data marts and execute long-running queries.

See Also: snowflake schema and star schema

transformation

The process of manipulating data. Any manipulation beyond copying is a transformation. Examples include cleansing, aggregating, and integrating data from multiple sources.

transportation

The process of moving copied or transformed data from a source to a data warehouse.

See Also: transformation

unique identifier

An identifier whose purpose is to differentiate between the same item when it appears in more than one place.

update window

The length of time available for updating a warehouse. For example, you might have 8 hours at night to update your warehouse.

update frequency

How often a data warehouse is updated with new information. For example, a warehouse might be updated nightly from an OLTP system.

validation

The process of verifying metadata definitions and configuration parameters.

versioning

The ability to create new versions of a data warehouse project for new requirements and changes.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X