Oracle® Text

Application Developer's Guide

10*g* Release 1 (10.1)

Part No. B10729-01

December 2003

ORACLE®

Oracle Text Application Developer's Guide, 10*g* Release 1 (10.1)

Part No. B10729-01

Primary Author: Colin McGregor

Contributors: Omar Alonso, Shamim Alpha, Steve Buxton, Chung-Ho Chen, Jack Chen, Yun Cheng, Michele Cyran, Paul Dixon, Mohammad Faisal, Roger Ford, Elena Huang, Garrett Kaminaga, Ji Sun Kang, Ciya Liao, Wesley Lin, Bryn Llewellyn, Yasuhiro Matsuda, Valarie Moore, Takeshi Okawa, Gerda Shank, Qunong Xiao, Steve Yang

Contents

2 Getting Started with Oracle Text

3 Indexing

4 Querying

6 Document Classification

7 Performance Tuning

8 Document Section Searching

9 Working With a Thesaurus

10 Administration

11 Migrating Applications from Earlier Releases

A CONTEXT Query Application

B CATSEARCH Query Application

Index

Send Us Your Comments

Oracle Text Application Developer's Guide, 10*g* Release 1 (10.1)

Part No. B10729-01

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
- 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 guide explains how to build query applications with Oracle Text. This preface contains these topics:

Audience

Oracle Text Application Developer's Guide is intended for users who perform the following tasks:

- Develop Oracle Text applications
- Administer Oracle Text installations

To use this document, you need to have experience with the Oracle object relational database management system, SQL, SQL*Plus, and PL/SQL.

Organization

This document contains:

Chapter 1, "Oracle Text Application Development"

This chapter explains the basic features of the query, catalog, and classification applications that you can build with Oracle Text.

Chapter 2, "Getting Started with Oracle Text"

This chapter explains how to get started on building a simple query applications using Oracle Text.

Chapter 3, "Indexing"

This chapter describes how to index your document set. It discusses considerations for indexing as well as how to create CONTEXT, CTXCAT, and CTXRULE indexes.

Chapter 4, "Querying"

This chapter describes how to query your document set. It gives examples for how to use the CONTAINS, CATSEARCH, and MATCHES operators.

Chapter 5, "Document Presentation"

This chapter describes how to present documents to the user of your query application.

Chapter 6, "Document Classification"

This chapter describes how to build classification applications.

Chapter 7, "Performance Tuning"

This chapter describes how to tune your queries to improve response time and throughput.

Chapter 8, "Document Section Searching"

This chapter describes how to enable section searching in HTML and XML.

Chapter 9, "Working With a Thesaurus"

This chapter describes how to work with a thesaurus in your application. It also describes how to augment your knowledge base with a thesaurus.

Chapter 10, "Administration"

This chapter describes Oracle Text administration.

Chapter 11, "Migrating Applications from Earlier Releases"

This chapter describes how to migrate your applications from earlier versions of Oracle Text.

Appendix A, "CONTEXT Query Application"

This appendix describes a sample Oracle Text CONTEXT Web application and the wizard used to produce it.

Appendix B, "CATSEARCH Query Application"

This appendix describes an Oracle Text CATSEARCH example Web application.

Related Documentation

For more information about Oracle Text, refer to:

- *Oracle Text Reference*

For more information about Oracle Database, refer to:

- *Oracle Database Concepts*
- *Oracle Database Administrator's Guide*
- *Oracle Database Utilities*
- *Oracle Database Performance Tuning Guide*
- *Oracle Database SQL Reference*
- *Oracle Database Reference*
- *Oracle Database Application Developer's Guide - Fundamentals*

For more information about PL/SQL, refer to:

- *PL/SQL User's Guide and Reference*

You can obtain Oracle Text technical information, collateral, code samples, training slides and other material at:

`http://otn.oracle.com/products/text/`

Many books in the documentation set use the sample schemas of the seed database, which is installed by default when you install Oracle Database. 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/`

Conventions

This section describes the conventions used in the text and code examples of the 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.	The C datatypes such as **ub4**, **sword**, or **OCINumber** are valid. When you specify this clause, you create an **index-organized table**.
Italics	Italic typeface indicates query terms, book titles, emphasis, syntax clauses, or placeholders.	*Oracle9i Concepts* You can specify the *parallel_clause*. Run U*old_release*.SQL where *old_release* refers to the release you installed prior to upgrading.

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, user names, and roles.	You can specify this clause only for a `NUMBER` column. You can back up the database using the `BACKUP` command. Query the `TABLE_NAME` column in the `USER_TABLES` table in the data dictionary view. Specify the `ROLLBACK_SEGMENTS` parameter. Use the `DBMS_STATS.GENERATE_STATS` procedure.
lowercase monospace (fixed-width font)	Lowercase monospace typeface indicates executables 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, user names and roles, program units, and parameter values.	Enter `sqlplus` to open SQL*Plus. 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.

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]`

Convention	Meaning	Example
...	Horizontal ellipsis points indicate either:	
	That we have omitted parts of the code that are not directly related to the example	`CREATE TABLE ... AS subquery;`
	That you can repeat a portion of the code	`SELECT col1, col2, ... , coln FROM employees;`
. . .	Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example.	
Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as it is shown.	`acctbal NUMBER(11,2);` `acct CONSTANT NUMBER(4) := 3;`
Italics	Italicized text indicates variables for which you must supply particular values.	`CONNECT SYSTEM/system_password`

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;`

lowercase	Lowercase typeface indicates programmatic elements that you supply. For example, lowercase indicates names of tables, columns, or files.	`SELECT last_name, employee_id FROM employees;` `sqlplus hr/hr`

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.

1

Oracle Text Application Development

This chapter discuses the following topics:

What is Oracle Text?

Oracle Text is a technology that enables you to build text query applications and document classification applications. Oracle Text provides indexing, word and theme searching, and viewing capabilities for text.

Designing Your Application

To design your Oracle Text application, you must determine the type of queries you expect to execute. Doing so enables you to choose the most suitable index for the task. We can divide application queries into three different categories:

Text Queries on Document Collections

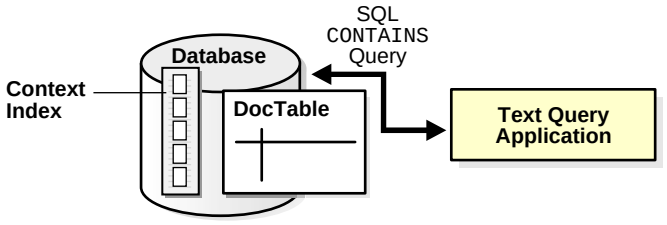
A text query application enables users to search document collections such as Web sites, digital libraries, or document warehouses. Searching is enabled by first indexing the document collection. The collection is typically static with no significant change in content after the initial indexing run. Documents can be of any size and of different formats such as HTML, PDF, or Microsoft Word. These documents are stored in a document table.

Queries usually consist of words or phrases. Application users can specify logical combinations of words and phrases using operators such as `OR` and `AND`. Other query operations such as stemming, proximity searching, and wildcarding can be used to improve the search results.

An important factor for this type of application is retrieving documents that are relevant to a user query while retrieving as few non-relevant documents as possible. The most relevant documents must be ranked high in the result list.

The queries for this type of application are best served with a `CONTEXT` index on your document table. To query this index, your application uses the `SQL CONTAINS` operator in the `WHERE` clause of a `SELECT` statement

Figure 1–1 Overview of Text Query Application



Flowchart of Text Query Application

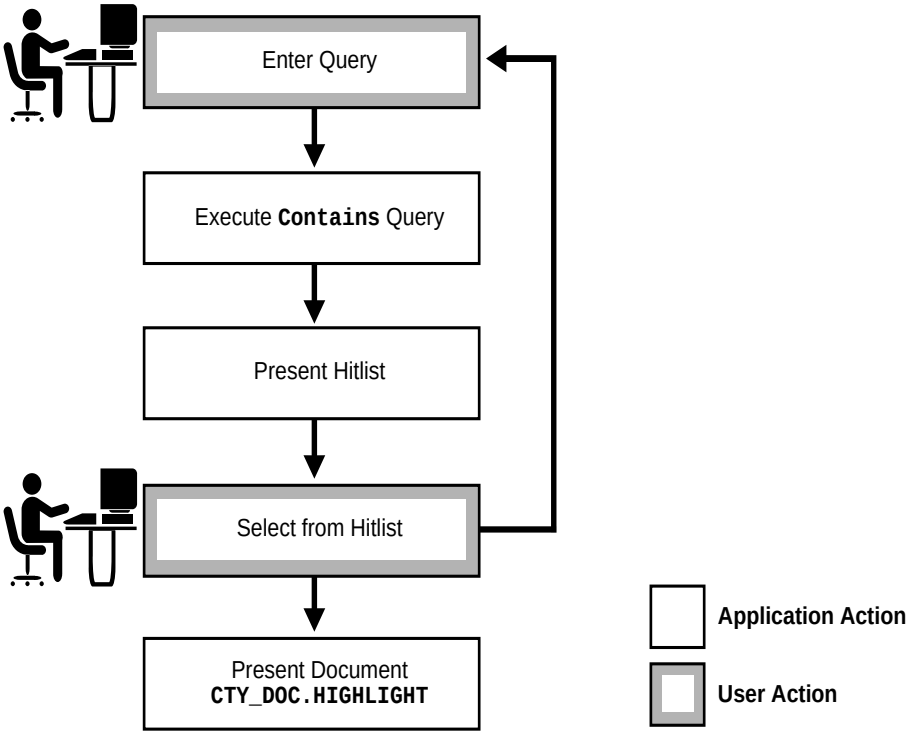
A typical text query application on a document collection enables the user to enter a query. The application issues a CONTAINS query and returns a list, called a *hitlist*, of documents that satisfy the query. The results are usually ranked by relevance. The application enables the user to view one or more documents in the hitlist.

For example, an application might index URLs (HTML files) on the World Wide Web and provide query capabilities across the set of indexed URLs. Hitlists returned by the query application are composed of URLs that the user can visit.

Figure 1–2 illustrates the flowchart of how a user interacts with a simple query application. The figure shows the steps required to enter the query through to viewing the results. A query application can be modeled according to the following steps:

1. The user enters a query.
2. The application executes a CONTAINS query.
3. The application presents a hitlist.
4. The user selects document from hitlist.
5. The application presents a document to the user for viewing.

Figure 1–2 Flowchart of a query application



Queries on Catalog Information

Catalog information consists of inventory type information such as that of an online book store or auction site. The stored information consists of text information such as book titles and related structured information such as price. The information is usually updated regularly to keep the online catalog up to date with the inventory.

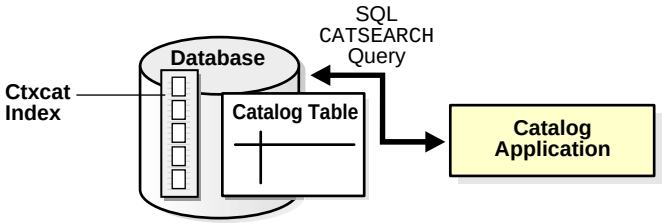
Queries are usually a combination of a text component and a structured component, such as price or author. Results are almost always sorted by a structured component such as date or price.

Good response time is always an important factor with this type of query application.

Catalog applications are best served by a CTXCAT index. You query this index with the CATSEARCH operator in the WHERE clause of a SELECT statement.

Figure 1–3 illustrates the relation of the catalog table, its CTXCAT index, and the catalog application which uses the CATSEARCH operator to query the index.

Figure 1–3 A Catalog Query Application



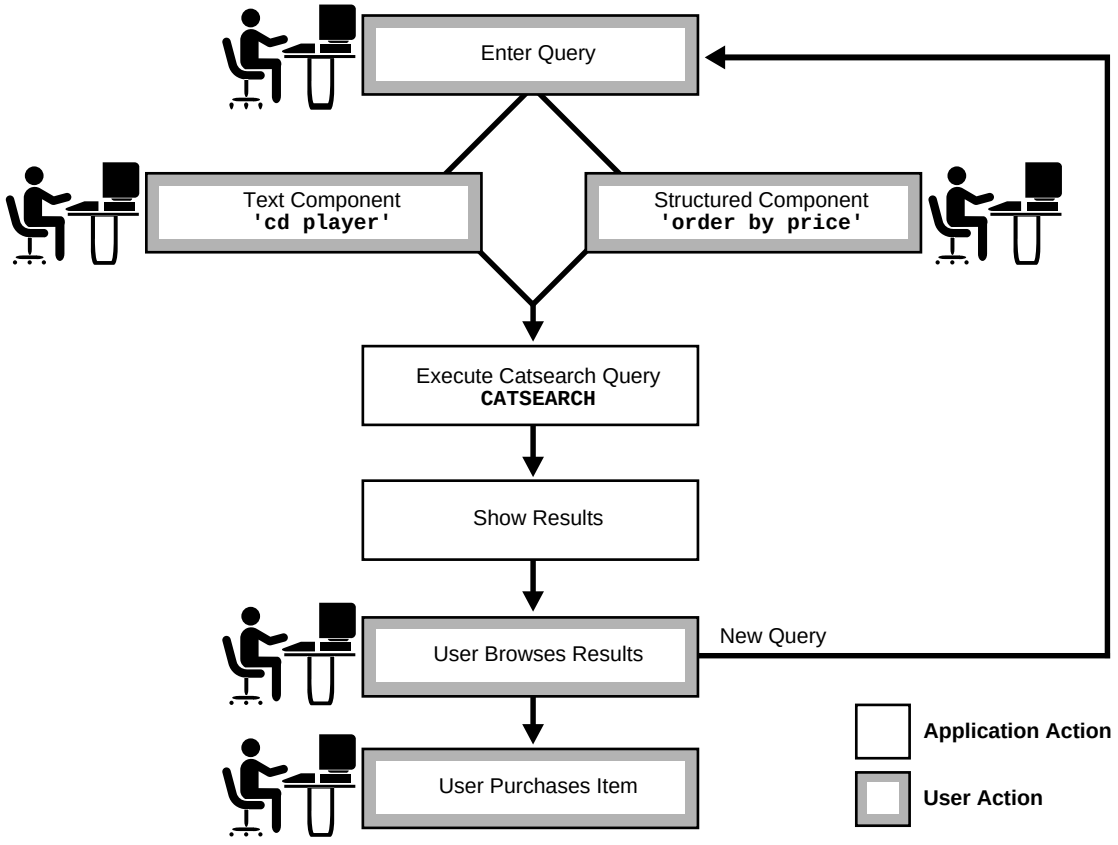
Flowchart for Catalog Query Application

A catalog application enables users to search for specific items in catalogs. For example, an online store application enables users to search for and purchase items in inventory. Typically, the user query consists of a text component that searches across the textual descriptions plus some other ordering criteria, such as price or date.

Figure 1–4 illustrates the flowchart of a catalog query application for an online electronics store.

1. The user enters the query, consisting of a text component (for example *cd player*) and a structured component (for example *order by price*).
2. The application executes the CATSEARCH query.
3. The application shows the results ordered accordingly.
4. The user browses the results.
5. The user then either issues another query or performs an action, such as purchasing the item.

Figure 1–4 Flowchart of a catalog query application



Document Classification

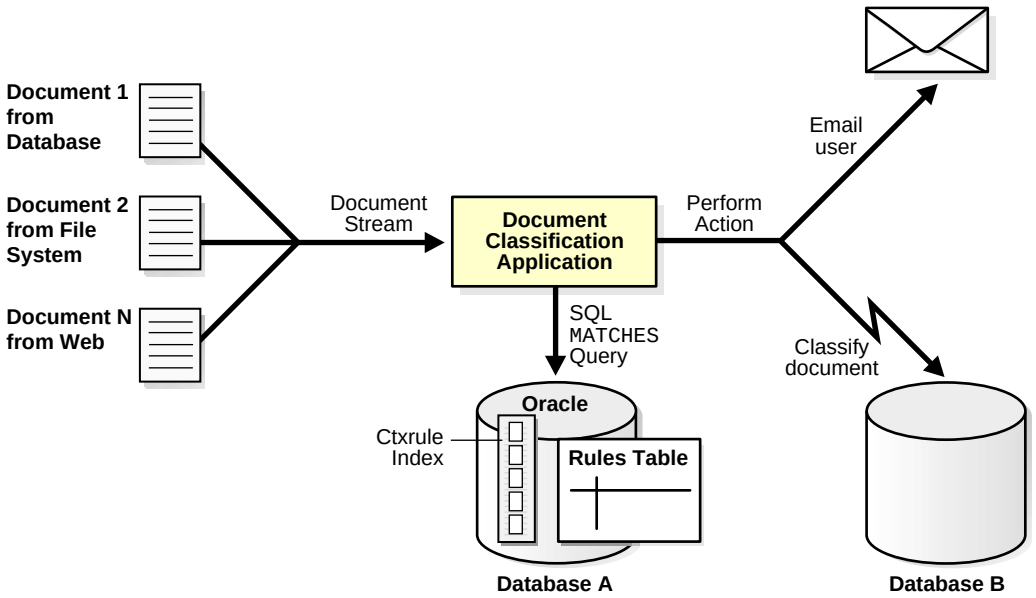
In a document classification application, an incoming stream or a set of documents is compared to a pre-defined set of rules. When a document matches one or more rules, the application performs some action.

For example, assume we have an incoming stream of news articles. We can define a rule to represent the category of Finance. The rule is essentially one or more queries that select document about the subject of Finance. The rule might have the form *'stocks or bonds or earnings'*.

When a document arrives about a Wall Street earnings forecast and satisfies the rules for this category, the application takes an action such as tagging the document as Finance or emailing one or more users.

To create a document classification application, you create a table of rules and then create a CTXRULE index. To classify an incoming stream of text, use the MATCHES operator in the WHERE clause of a SELECT statement. Refer to Figure 1–5 for the general flow of a classification application.

Figure 1–5 Overview of a Document Classification Application



XML Searching

An XML search application performs searches over XML documents. In a regular document search, you usually search across a set of documents to return documents that satisfy a text predicate; in an XML search, you often use the structure of the XML document to restrict the search. Typically, only that part of the document that satisfies the search is returned. For example, instead of finding all purchase orders

that contain the word *electric*, the user might need only purchase orders in which the comment field contains *electric*.

Oracle Text enables you to perform XML searching using the following approaches:

- Using Oracle Text
- Using the Oracle XML DB Framework
- Combining Oracle Text features with Oracle XML DB

Using Oracle Text

The `CONTAINS` operator is well suited to structured searching, enabling you to perform restrictive searches with the `WITHIN`, `HASPATH`, and `INPATH` operators. If you use a CONTEXT index, you can also benefit from the following characteristics of Oracle Text searches:

- searches are token-based, whitespace-normalized
- hit lists are ranked by relevance
- you can enable case-sensitive searching
- you can utilize section searching
- you can leverage linguistic features such as stemming and fuzzy searching
- queries are performance-optimized for large document sets

See Also: "XML Section Searching" on page 8-14

Using the Oracle XML DB Framework

With Oracle XML DB, you load your XML documents in an `XMLType` column. XML searching with Oracle XML DB usually consists of an `XPATH` expression within an `existsNode()`, `extract()`, or `extractValue()` query. This type of search can be characterized as follows:

- non-text search with equality and range on dates and numbers
- string search that is character-based where all characters are treated the same
- has the ability to leverage the `ora:contains()` function with a `CTXXPATH` index to speed up `existsNode()` queries.

This type of search has the following disadvantages:

- no special linguistic processing
- uses exact matching so there is no notion of relevance
- can be very slow for some searches, such as wildcarding, as with:

```
WHERE col1 like '%dog%'
```

See Also: The *Oracle XML DB Developer's Guide*

Combining Oracle Text features with Oracle XML DB

You can combine the features of Oracle Text and Oracle XML DB for applications in which you want to do a full-text retrieval, leveraging the XML structure by issuing queries such as "find all nodes that contain the word Pentium." You do so in one of two ways:

- Using the Text-on-XML Method
- Using the XML-on-Text Method

See Also: The *Oracle XML DB Developer's Guide* and "XML Section Searching" on page 8-14

Using the Text-on-XML Method

With Oracle Text, you can create a CONTEXT index on a column that contains your XML data. Your column type can be `XMLType`, but can also be any supported type provided you use the correct index preference for XML data.

With the Text-on-XML method, you use the standard `CONTAINS` query and add a structured constraint to limit the scope of a search to a particular section, field, tag, or attribute. This amounts to specifying the structure inside text operators such as `WITHIN`, `HASPATH`, and `INPATH`.

For example, you can set up your CONTEXT index to create sections with XML documents. Consider the following XML document that defines a purchase order.

```
<?xml version="1.0"?>
<PURCHASEORDER pono="1">
   <PNAME>Po_1</PNAME>
   <CUSTNAME>John</CUSTNAME>
   <SHIPADDR>
      <STREET>1033 Main Street</STREET>
      <CITY>Sunnyvalue</CITY>
      <STATE>CA</STATE>
```

```
   </SHIPADDR>
   <ITEMS>
     <ITEM>
       <ITEM_NAME> Dell Computer </ITEM_NAME>
       <DESC> Pentium 2.0 Ghz 500MB RAM  </DESC>
     </ITEM>
     <ITEM>
       <ITEM_NAME> Norelco R100 </ITEM_NAME>
       <DESC>Electric Razor </DESC>
     </ITEM>
   </ITEMS>
</PURCHASEORDER>
```

To query all purchase orders that contain *Pentium* within the item description section, you might use the `WITHIN` operator as follows:

```
SELECT id from po_tab where CONTAINS( doc, 'Pentium WITHIN desc') > 0;
```

You can specify more complex criteria with `XPATH` expressions using `INPATH` operator:

```
SELECT id from po_tab where  CONTAINS(doc, 'Pentium INPATH
(/purchaseOrder/items/item/desc') > 0;
```

Using the XML-on-Text Method

With the XML-on-Text method, you add text operations to an XML search. This includes using the `ora:contains()` function in the `XPATH` expression with `existsNode()`, `extract()`, and `extractValue()` queries. This amounts to including the full-text predicate inside the structure. For example:

```
SELECT
 Extract(doc, '/purchaseOrder//desc{ora:contains(.,"pentium")>0]',
                    'xmlns:ora=http://xmlns.oracle.com/xdb')
"Item Comment" FROM po_tab_xmltype
/
```

Additionally you can improve the performance of `existsNode()`, `extract()`, and `extractValue()` queries using the `CTXXPATH` Text domain index.

2

Getting Started with Oracle Text

This chapter discuses the following topics:

Overview of Getting Started with Oracle Text

This chapter describes how to get started with creating an Oracle Text developer and building simple text query and catalog applications. For each type of application, this chapter steps you through the basic SQL statements for loading, indexing and querying your tables.

More complete application examples are given in the Appendices. To learn more about building document classification applications, see Chapter 6.

> **Note:** The `SQL>` prompt has been omitted in this chapter, in part to improve readability and in part to make it easier for you to cut and paste text.

Creating an Oracle Text User

Before you can create Oracle Text indexes and use Oracle Text PL/SQL packages, you need to create a user with the CTXAPP role. This role enables you to do the following:

- Create and delete Oracle Text indexing preferences
- Use the Oracle Text PL/SQL packages

To create an Oracle Text application developer user, execute the following SQL statements as the system administrator user:

Step 1 Create User

The following SQL command creates a user called MYUSER with a password of myuser_password:

```
CREATE USER myuser IDENTIFIED BY myuser_password;
```

Step 2 Grant Roles

The following SQL command grants the required roles of RESOURCE,CONNECT, and CTXAPP to MYUSER:

```
GRANT RESOURCE, CONNECT, CTXAPP TO MYUSER;
```

Step 3 Grant EXECUTE Privileges on CTX PL/SQL Packages

There are ten Oracle Text packages that enable you to perform actions ranging from synchronizing an Oracle Text index to highlighting documents. For example, the CTX_DDL.SYNC_INDEX package enables you to synchronize your index.

To call any of these procedures from a stored procedure, your application requires execute privileges on the packages.

For example, to grant to MYUSER execute privileges on all Oracle Text packages, issue the following SQL commands:

```
GRANT EXECUTE ON CTX_CLS TO myuser;
GRANT EXECUTE ON CTX_DDL TO myuser;
GRANT EXECUTE ON CTX_DOC TO myuser;
GRANT EXECUTE ON CTX_OUTPUT TO myuser;
GRANT EXECUTE ON CTX_QUERY TO myuser;
GRANT EXECUTE ON CTX_REPORT TO myuser;
GRANT EXECUTE ON CTX_THES TO myuser;
```

Query Application Quick Tour

In a basic text query application, users enter query words or phrases and expect the application to return a list of documents that best match the query. Such an application involves creating a CONTEXT index and querying it with CONTAINS.

This example steps you through the basic SQL statements you use to load your text table, index your documents, and query your index.

Typically, query applications require a user interface. An example of how to build such a query application using the CONTEXT index type is given in Appendix A.

Step 1 Connect as the New User

Before creating any tables, assume the identity of the user you just created.

```
CONNECT myuser;
```

Step 2 Create your Text Table

The following example creates a table called `docs` with two columns, `id` and `text`, by using the `CREATE TABLE` statement. This example makes the `id` column the primary key. The `text` column is `VARCHAR2`.

```
CREATE TABLE docs (id NUMBER PRIMARY KEY, text VARCHAR2(200));
```

Step 3 Load Documents into Table

You can use the SQL `INSERT` statement to load text to a table.

To populate the `docs` table, use the `INSERT` statement as follows:

```
INSERT INTO docs VALUES(1, '<HTML>California is a state in the US.</HTML>');
INSERT INTO docs VALUES(2, '<HTML>Paris is a city in France.</HTML>');
INSERT INTO docs VALUES(3, '<HTML>France is in Europe.</HTML>');
```

Using SQL*Loader

You can also load your table in batch with SQL*Loader.

> **See Also:** "Building the Web Application" in Appendix A, "CONTEXT Query Application" for an example on how to use SQL*Loader to load a text table from a data file.

Step 1 Create the CONTEXT index

Index the HTML files by creating a `CONTEXT` index on the text column as follows. Since you are indexing HTML, this example uses the `NULL_FILTER` preference type for no filtering and uses the `HTML_SECTION_GROUP` type:

```
CREATE INDEX idx_docs ON docs(text)
     INDEXTYPE IS CTXSYS.CONTEXT PARAMETERS
     ('FILTER CTXSYS.NULL_FILTER SECTION GROUP CTXSYS.HTML_SECTION_GROUP');
```

Use the NULL_FILTER because you do not need to filter HTML documents during indexing. However, if you index PDF, Microsoft Word, or other formatted documents, use the CTXSYS.INSO_FILTER (the default) as your FILTER preference.

This example also uses the HTML_SECTION_GROUP section group which is recommended for indexing HTML documents. Using HTML_SECTION_GROUP enables you to search within specific HTML tags, and eliminates from the index unwanted markup such as font information.

Step 2 Querying Your Table with CONTAINS

You query the table with the SELECT statement with CONTAINS to retrieve the document ids that satisfy the query.

Before doing so, set the format of the SELECT statement's output so that it is easily readable. To do so, set the width of the `text` column to 40 characters:

```
COLUMN text FORMAT a40;
```

Now use SELECT. The following query looks for all documents that contain the word *France*:

```
SELECT SCORE(1), id, text FROM docs WHERE CONTAINS(text, 'France', 1) > 0;

  SCORE(1)         ID TEXT
---------- ---------- ----------------------------------------
         4          3 <HTML>France is in Europe.</HTML>
         4          2 <HTML>Paris is a city in France.</HTML>
```

Step 3 Present the Document

In a real application, you might want to present the selected document to the user with query terms highlighted. Oracle Text enables you to mark up documents with the CTX_DOC package.

We can demonstrate HTML document markup with an anonymous PL/SQL block in SQL*Plus. However, in a real application you might present the document in a browser.

This PL/SQL example uses the in-memory version of CTX_DOC.MARKUP to highlight the word *France* in document 3. It allocates a temporary CLOB (Character

Large Object datatype) to store the markup text and reads it back to the standard output. The CLOB is then de-allocated before exiting:

```
SET SERVEROUTPUT ON;
DECLARE
  2  mklob CLOB;
  3  amt NUMBER := 40;
  4  line VARCHAR2(80);
  5  BEGIN
  6    CTX_DOC.MARKUP('idx_docs','3','France', mklob);
  7    DBMS_LOB.READ(mklob, amt, 1, line);
  8    DBMS_OUTPUT.PUT_LINE('FIRST 40 CHARS ARE:'||line);
  9    DBMS_LOB.FREETEMPORARY(mklob);
 10    END;
 11  /
FIRST 40 CHARS ARE:<HTML><<<France>>> is in Europe.</HTML>

PL/SQL procedure successfully completed.
```

Step 4 Synchronize the Index After Data Manipulation

When you create a CONTEXT index, you need to explicitly synchronize your index to keep it up to date with any inserts, updates, or deletes to the text table.

Oracle Text enables you to do so with the CTX_DDL.SYNC_INDEX procedure.

Add some rows to the docs table:

```
INSERT INTO docs VALUES(4, '<HTML>Los Angeles is a city in California.</HTML>');
INSERT INTO docs VALUES(5, '<HTML>Mexico City is big.</HTML>');
```

Since the index is not synchronized, these new rows are not returned with a query on *city*:

```
SELECT SCORE(1), id, text FROM docs WHERE CONTAINS(text, 'city', 1) > 0;

  SCORE(1)         ID TEXT
---------- ---------- --------------------------------------------------
         4          2 <HTML>Paris is a city in France.</HTML>
```

Therefore, synchronize the index with 2Mb of memory, and reexecute the query:

```
EXEC CTX_DDL.SYNC_INDEX('idx_docs', '2M');

PL/SQL procedure successfully completed.
```

```
COLUMN text FORMAT a50;
SELECT SCORE(1), id, text FROM docs WHERE CONTAINS(text, 'city', 1) > 0;

  SCORE(1)         ID TEXT
---------- ---------- --------------------------------------------------
         4          5 <HTML>Mexico City is big.</HTML>
         4          4 <HTML>Los Angeles is a city in California.</HTML>
         4          2 <HTML>Paris is a city in France.</HTML>
```

Building Web Applications with the Oracle Text Wizard

Oracle Text enables you to build simple Text and Catalog Web applications with the Oracle Text Wizard addin for Oracle JDeveloper. The wizard automatically generates Java Server Pages or PL/SQL server scripts you can use with the Oracle-configured Apache Web server.

Both JDeveloper and the Text Wizard can be downloaded for free from the following Oracle Technology Network (OTN) sites. Note that you need to register with OTN before you can access these pages.

Oracle JDeveloper

You can obtain the latest JDeveloper software from:

http://otn.oracle.com/software/products/jdev/content.html

See "Building the JSP Web Application" on page B-2 for an example.

Oracle Text Wizard Addins

You can obtain the Text, Catalog, and Classification Wizard addins from:

http://otn.oracle.com/software/products/text/content.html

Oracle Text Wizard Instructions

You can find instructions on using the Oracle Text Wizard and setting up your JSP files to run in a Web server environment from:

http://otn.oracle.com/software/products/text/content.html

Follow the "Text Search Wizard for JDeveloper" link.

Catalog Application Quick Tour

This example creates a catalog index for an auction site that sells electronic equipment such as cameras and CD players. New inventory is added everyday and item descriptions, bid dates, and prices must be stored together.

The application requires good response time for mixed queries. The key is to determine what columns users frequently search so that we can create a suitable CTXCAT index. Queries on this type of index are issued with the CATSEARCH operator.

> **Note:** Typically, query applications require a user interface. An example of how to build such a query application using the CATSEARCH index type is given in Appendix B.

Step 1 Connect as the Appropriate User

In this case, we connect as the user `myuser`, whom we created in section "Create User".

```
CONNECT myuser;
```

Step 2 Create Your Table

Set up an auction table to store your inventory:

```
CREATE TABLE auction(
item_id NUMBER,
title VARCHAR2(100),
category_id NUMBER,
price NUMBER,
bid_close DATE);
```

Figure 2–1 illustrates this table.

Step 3 Populate Your Table

Now populate the table with various items, each with an `id, title, price` and `bid_date`:

```
INSERT INTO AUCTION VALUES(1, 'NIKON CAMERA', 1, 400, '24-OCT-2002');
INSERT INTO AUCTION VALUES(2, 'OLYMPUS CAMERA', 1, 300, '25-OCT-2002');
INSERT INTO AUCTION VALUES(3, 'PENTAX CAMERA', 1, 200, '26-OCT-2002');
INSERT INTO AUCTION VALUES(4, 'CANON CAMERA', 1, 250, '27-OCT-2002');
```

Using SQL*Loader

You can also load your table in batch with SQL*Loader.

> **See Also:** "Building the Web Application" in Appendix A, "CONTEXT Query Application" for an example on how to use SQL*Loader to load a text table from a data file.[

Step 1 Determine your Queries

You need to determine what criteria are likely to be retrieved. In this example, you determine that all queries search the title column for item descriptions, and most queries order by price. When using the CATSEARCH operator later, we'll specify the terms for the text column and the criteria for the structured clause.

Step 2 Create the Sub-Index to Order by Price

For Oracle Text to serve these queries efficiently, we need a sub-index for the price column, since our queries will order by price.

Therefore, create an index set called `auction_set` and add a sub-index for the price column:

```
EXEC CTX_DDL.CREATE.INDEXT_SET('auction_iset');
EXEC CTX_DDL.ADD_INDEX('auction_iset','price'); /* sub-index A*/
```

Figure 2–1 shows how the sub-index relates to the columns.

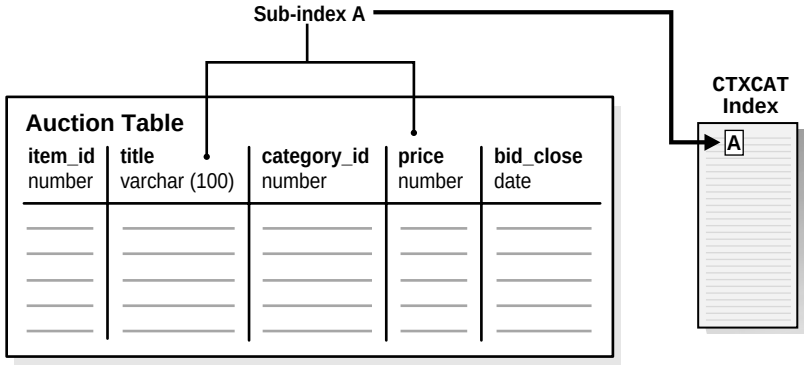
Step 3 Create the CTXCAT Index

Create the combined catalog index on the AUCTION table with CREATE INDEX as follows:

```
CREATE INDEX auction_titlex ON AUCTION(title) INDEXTYPE IS CTXSYS.CTXCAT
PARAMETERS ('index set auction_iset');
```

Figure 2–1 shows how the CTXCAT index and its sub-index relates to the columns.

Figure 2–1 Auction table schema and CTXCAT index



Step 1 Querying Your Table with CATSEARCH

When you have created the CTXCAT index on the AUCTION table, you can query this index with the CATSEARCH operator.

First set the output format to make the output readable:

```
COLUMN title FORMAT a40;
```

Now execute the query:

```
SELECT title, price FROM auction WHERE CATSEARCH(title, 'CAMERA', 'order by
price')> 0;

TITLE                PRICE
-------------- ----------
PENTAX CAMERA          200
CANON CAMERA           250
OLYMPUS CAMERA         300
NIKON CAMERA           400

SELECT title, price FROM auction WHERE CATSEARCH(title, 'CAMERA',
     'price <= 300')>0;

TITLE                PRICE
-------------- ----------
PENTAX CAMERA          200
```

```
CANON CAMERA           250
OLYMPUS CAMERA         300
```

Step 2 Update Your Table

You can update your catalog table by adding new rows. When you do so, the CTXCAT index is automatically synchronized to reflect the change.

For example, add the following new rows to our table and then reexecute the query:

```
INSERT INTO AUCTION VALUES(5, 'FUJI CAMERA', 1, 350, '28-OCT-2002');
INSERT INTO AUCTION VALUES(6, 'SONY CAMERA', 1, 310, '28-OCT-2002');

SELECT title, price FROM auction WHERE CATSEARCH(title, 'CAMERA', 'order by
price')> 0;

TITLE                                    PRICE
---------------------------------- ----------
PENTAX CAMERA                             200
CANON CAMERA                              250
OLYMPUS CAMERA                            300
SONY CAMERA                               310
FUJI CAMERA                               350
NIKON CAMERA                              400

6 rows selected.
```

Note how the added rows show up immediately in the query.

Classification Application Quick Tour

The function of a classification application is to perform some action based on document content. These actions can include assigning a category id to a document or sending the document to a user. The result is classification of a document.

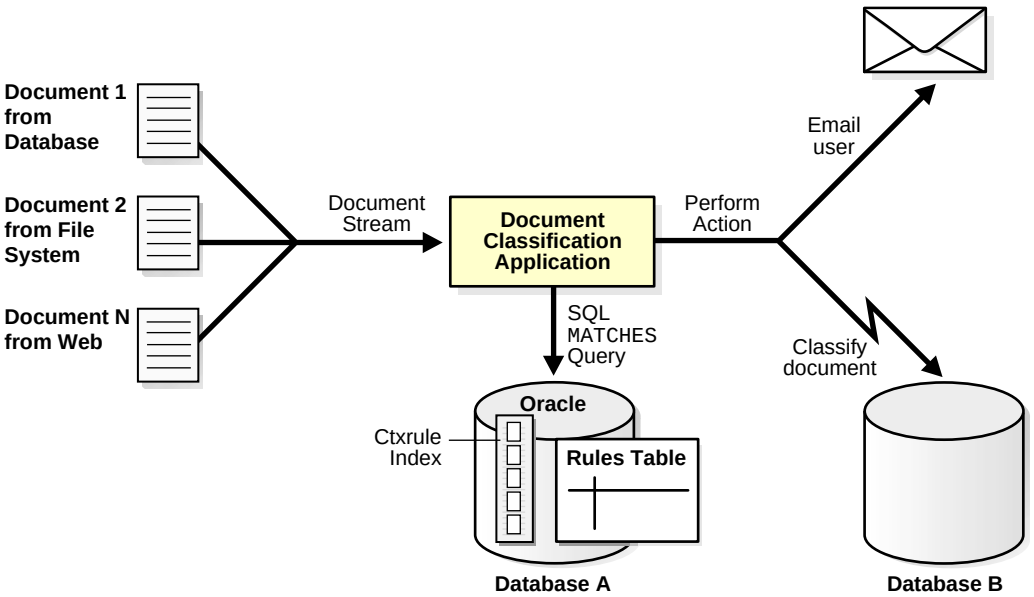
Documents are classified according to pre-defined rules. These rules select for a category. For instance, a query rule of *'presidential elections'* might select documents for a category about politics.

Oracle Text provides several types of classification. One type is *simple*, or *rule-based* classification, discussed here, in which you create both document categories and the rules for categorizing documents. With *supervised classification*, Oracle Text derives the rules from a set of training documents you provide. With *clustering*, Oracle Text

does all the work for you, deriving both rules and categories. (For more on classification, see Chapter 6, "Document Classification" in this book.)

To achieve simple classification in Oracle Text, you create rules, which are essentially a table of queries. You index these rules in a CTXRULE index. To classify an incoming stream of text, use the MATCHES operator in the WHERE clause of a SELECT statement. Refer to Figure 2–2 for the general flow of a classification application.

Figure 2–2 Overview of a Document Classification Application



Steps for Creating a Classification Application

The following example steps you through defining simple categories, creating a CTXRULE index, and using MATCHES to classify documents.

Step 1 Connect As the Appropriate User

In this case, we connect as the user myuser, which we created in section "Create User".

```
CONNECT myuser;
```

Step 2 Create the Rule Table

We must create a rule table and populate it with query rules. In this example, we create a table called queries. Each row defines a category with an id, and a rule which is a query string:

```
CREATE TABLE queries (
      query_id      NUMBER,
      query_string  VARCHAR2(80)
    );

    INSERT INTO queries VALUES (1, 'oracle');
    INSERT INTO queries VALUES (2, 'larry or ellison');
    INSERT INTO queries VALUES (3, 'oracle and text');
    INSERT INTO queries VALUES (4, 'market share');
```

Step 3 Create Your CTXRULE Index

Create a CTXRULE index as follows:

```
 CREATE INDEX queryx ON queries(query_string) INDEXTYPE IS CTXRULE;
```

Step 4 Classify with MATCHES

Use the MATCHES operator in the WHERE clause of a SELECT statement to match documents to queries and hence classify.

```
    COLUMN query_string FORMAT a35;
    SELECT query_id,query_string FROM queries
     WHERE MATCHES(query_string,
                   'Oracle announced that its market share in databases
                    increased over the last year.')>0;

  QUERY_ID QUERY_STRING
---------- -----------------------------------
         1 oracle
         4 market share
```

As shown, the document string matches categories 1 and 4. With this classification you can perform an action, such as writing the document to a specific table or emailing a user.

> **See Also:** Chapter 6, "Document Classification" for more extended classification examples.

3

Indexing

The chapter is an introduction to Oracle Text indexing. The following topics are covered:

- About Oracle Text Indexes
- Considerations For Indexing
- Index Creation
- Index Maintenance
- Managing DML Operations for a CONTEXT Index

About Oracle Text Indexes

The following sections discuss the different types of Oracle Text indexes, their structure, the indexing process, and limitations.

Type of Index

With Oracle Text, you can create one of four index types with `CREATE INDEX`. The following table describes each type, its purpose, and what features it supports:

Index Type	Description	Supported Preferences and Parameters	Query Operator	Notes
CONTEXT	Use this index to build a text retrieval application when your text consists of large coherent documents. You can index documents of different formats such as MS Word, HTML or plain text. With a context index, you can customize your index in a variety of ways. This index type requires CTX_DDL.SYNC_INDEX after DML on base table.	All CREATE INDEX preferences and parameters supported except for INDEX SET. These supported parameters include the index partition clause, and the format, charset, and language columns.	CONTAINS Grammar is called the CONTEXT grammar, which supports a rich set of operations. The CTXCAT grammar can be used with query templating.	Supports all documents services and query services. Supports indexing of partitioned text tables.

Index Type	Description	Supported Preferences and Parameters	Query Operator	Notes
CTXCAT	Use this index type for better mixed query performance. Typically, with this index type, you index small documents or text fragments. Other columns in the base table, such as item names, prices and descriptions can be included in the index to improve mixed query performance. This index type is transactional, automatically updating itself after DML to base table. No CTX_DDL.SYNC_INDEX is necessary.	INDEX SET LEXER STOPLIST STORAGE WORDLIST (only prefix_index attribute supported for Japanese data) Format, charset, and language columns not supported. Table and index partitioning not supported.	CATSEARCH Grammar is called CTXCAT, which supports logical operations, phrase queries, and wildcarding. The CONTEXT grammar can be used with query templating. Theme querying is supported.	This index is larger and takes longer to build than a CONTEXT index. The size of a CTXCAT index is related to the total amount of text to be indexed, number of indexes in the index set, and number of columns indexed. Carefully consider your queries and your resources before adding indexes to the index set. The CTXCAT index does not support table and index partitioning, documents services (highlighting, markup, themes, and gists) or query services (explain, query feedback, and browse words.)
CTXRULE	Use CTXRULE index to build a document classification or routing application. The CTXRULE index is an index created on a table of queries, where the queries define the classification or routing criteria.	See "CTXRULE Parameters and Limitations" on page 6-8.	MATCHES	Single documents (plain text, HTML, or XML) can be classified using the MATCHES operator, which turns a document into a set of queries and finds the matching rows in the CTXRULE index.

Index Type	Description	Supported Preferences and Parameters	Query Operator	Notes
CTXXPATH	Create this index when you need to speed up existsNode() queries on an XMLType column.	STORAGE	Use with existsNode()	Can only create this index on XMLType column. Although this index type can be helpful for `existsNode()` queries, it is not required for XML searching. See "XML Searching" on page 1-7

See Also: Index Creation in this chapter.

An Oracle Text index is an Oracle Database domain index.To build your query application, you can create an index of type CONTEXT and query it with the CONTAINS operator.

You create an index from a populated text table. In a query application, the table must contain the text or pointers to where the text is stored. Text is usually a collection of documents, but can also be small text fragments.

For better performance for mixed queries, you can create a CTXCAT index. Use this index type when your application relies heavily on mixed queries to search small documents or descriptive text fragments based on related criteria such as dates or prices. You query this index with the CATSEARCH operator.

To build a document classification application using *simple* or *rule-based* classification, you create an index of type CTXRULE. With such an index, you can classify plain text, HTML, or XML documents using the MATCHES operator. You store your defining query set in the text table you index.

If you are working with XMLtype columns, you can create a CTXXPATH index to speed up queries with existsNode.

You create a text index as a type of extensible index to Oracle Database using standard SQL. This means that an Oracle Text index operates like an Oracle Database index. It has a name by which it is referenced and can be manipulated with standard SQL statements.

The benefits of a creating an Oracle Text index include fast response time for text queries with the CONTAINS, CATSEARCH, and MATCHES Oracle Text operators.

These operators query the CONTEXT, CTXCAT, and CTXRULE index types respectively.

> **See Also:** "Index Creation" in this chapter.

Structure of the Oracle Text CONTEXT Index

Oracle Text indexes text by converting all words into tokens. The general structure of an Oracle Text CONTEXT index is an inverted index where each token contains the list of documents (rows) that contain that token.

For example, after a single initial indexing operation, the word DOG might have an entry as follows:

Word	Appears in Document
DOG	DOC1 DOC3 DOC5

This means that the word DOG is contained in the rows that store documents one, three and five.

For more information, see optimizing the index in this chapter.

Merged Word and Theme Index

By default in English and French, Oracle Text indexes theme information with word information. You can query theme information with the ABOUT operator. You can optionally enable and disable theme indexing.

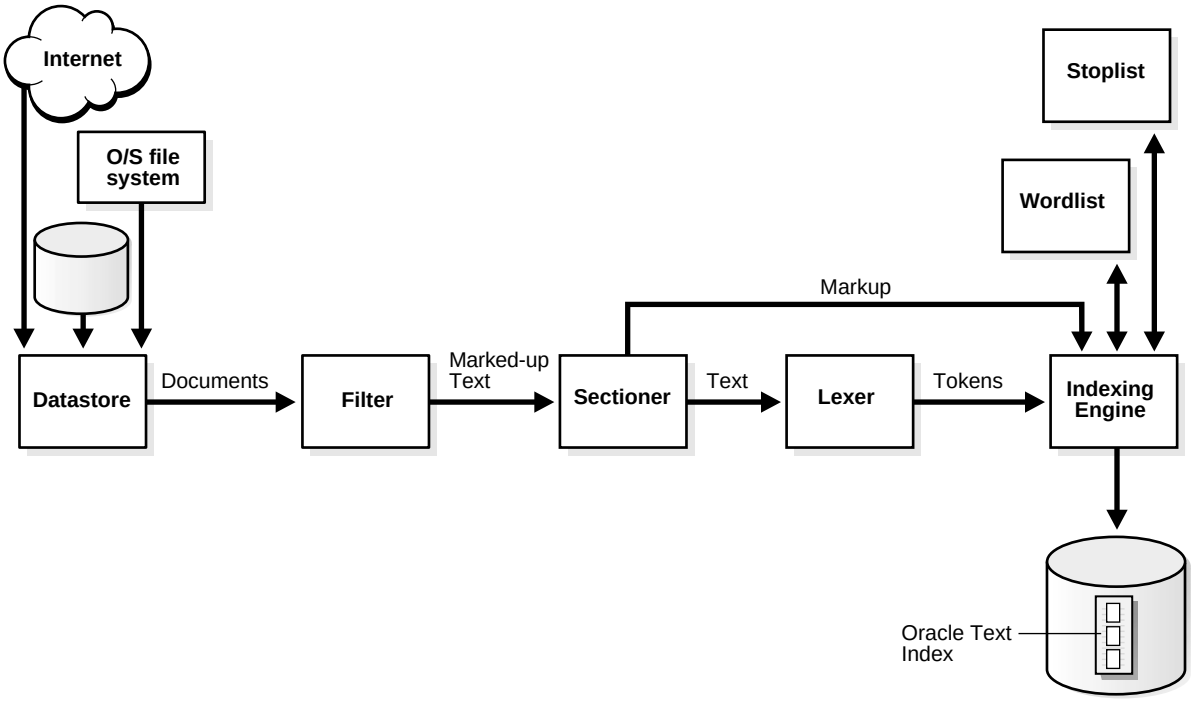
> **See Also:** To learn more about indexing theme information, see "Creating Preferences" in this chapter.

The Oracle Text Indexing Process

This section describes the Oracle Text indexing process.You initiate the indexing process with the CREATE INDEX statement. The goal is to create an Oracle Text index of tokens according to the parameters and preferences you specify.

Figure 3–1 shows the indexing process. This process is a data stream that is acted upon by the different indexing objects. Each object corresponds to an indexing preference type or section group you can specify in the parameter string of CREATE INDEX or ALTER INDEX. The sections that follow describe these objects.

Figure 3–1 Oracle Text Indexing Process



Datastore Object

The stream starts with the datastore reading in the documents as they are stored in the system according to your datastore preference. For example, if you have defined your datastore as `FILE_DATASTORE`, the stream starts by reading the files from the operating system. You can also store your documents on the internet or in the Oracle Database. Wherever your files reside physically, you must always have a text table in the Oracle Database that points to the file.

Filter Object

The stream then passes through the filter. What happens here is determined by your FILTER preference. The stream can be acted upon in one of the following ways:

- No filtering takes place. This happens when you specify the `NULL_FILTER` preference type or when the value of the format column is `IGNORE`. Documents that are plain text, HTML, or XML need no filtering.

- Formatted documents (binary) are filtered to marked-up text. This happens when you specify the INSO_FILTER preference type or when the value of the format column is BINARY.
- Text is converted from a non-database character set to the database character set. This happens when you specify CHARSET_FILTER preference type.

Sectioner Object

After being filtered, the marked-up text passes through the sectioner that separates the stream into text and section information. Section information includes where sections begin and end in the text stream. The type of sections extracted is determined by your section group type.

The section information is passed directly to the indexing engine which uses it later. The text is passed to the lexer.

Lexer Object

The lexer breaks the text into tokens according to your language. These tokens are usually words. To extract tokens, the lexer uses the parameters as defined in your lexer preference. These parameters include the definitions for the characters that separate tokens such as whitespace, and whether to convert the text to all uppercase or to leave it in mixed case.

When theme indexing is enabled, the lexer analyses your text to create theme tokens for indexing.

Indexing Engine

The indexing engine creates the inverted index that maps tokens to the documents that contain them. In this phase, Oracle Text uses the stoplist you specify to exclude stopwords or stopthemes from the index. Oracle Text also uses the parameters defined in your WORDLIST preference, which tell the system how to create a prefix index or substring index, if enabled.

Partitioned Tables and Indexes

You can create a partitioned CONTEXT index on a partitioned text table. The table must be partitioned by range. Hash, composite and list partitions are not supported.

You might create a partitioned text table to partition your data by date. For example, if your application maintains a large library of dated news articles, you can partition your information by month or year. Partitioning simplifies the

manageability of large databases since querying, DML, and backup and recovery can act on single partitions.

> **See Also:** *Oracle Database Concepts* for more information about partitioning.

Querying Partitioned Tables

To query a partitioned table, you use `CONTAINS` in the `WHERE` clause of a `SELECT` statement as you query a regular table. You can query the entire table or a single partition. However, if you are using the `ORDER BY SCORE` clause, Oracle recommends that you query single partitions unless you include a range predicate that limits the query to a single partition.

Creating an Index Online

When it is not practical to lock up your base table for indexing because of ongoing updates, you can create your index online with the ONLINE parameter of CREATE INDEX. This way an application with heavy DML need not stop updating the base table for indexing.

There are short periods, however, when the base table is locked at the beginning and end of the indexing process.

> **See Also:** *Oracle Text Reference* to learn more about creating an index online.

Parallel Indexing

Oracle Text supports parallel indexing with `CREATE INDEX`.

When you issue a parallel indexing command on a non-partitioned table, Oracle Text splits the base table into temporary partitions, spawns slave processes, and assigns a slave to a partition. Each slave indexes the rows in its partition. The method of slicing the base table into partitions is determined by Oracle Text and is not under your direct control. This is true as well for the number of slave processes actually spawned, which depends on machine capabilities, system load, your init.ora settings, and other factors. The actual parallel degree may not match the degree of parallelism requested.

Since indexing is an I/O intensive operation, parallel indexing is most effective in decreasing your indexing time when you have distributed disk access and multiple CPUs. Parallel indexing can only affect the performance of an initial index with

CREATE INDEX. It does not affect DML performance with ALTER INDEX, and has minimal impact on query performance.

Since parallel indexing decreases the *initial* indexing time, it is useful for

- data staging, when your product includes an Oracle Text index
- rapid initial startup of applications based on large data collections
- application testing, when you need to test different index parameters and schemas while developing your application

See Also:

"Frequently Asked Questions About Indexing Performance" in Chapter 7, "Performance Tuning" to learn more about creating an index in parallel.

Oracle Text Reference

Indexing and Views

Oracle SQL standards do not support creating indexes on views. If you need to index documents whose contents are in different tables, you can create a data storage preference using the USER_DATASTORE object. With this object, you can define a procedure that synthesizes documents from different tables at index time.

See Also: *Oracle Text Reference* to learn more about USER_DATASTORE.

Oracle Text does support the creation of CONTEXT, CTXCAT, CTXRULE, and CTXXPATH indexes on materialized views (MVIEW).

Considerations For Indexing

You use the CREATE INDEX statement to create an Oracle Text index. When you create an index and specify no parameter string, an index is created with default parameters. You can create either a CONTEXT, CTXCAT, or CTXRULE index.

You can also override the defaults and customize your index to suit your query application. The parameters and preference types you use to customize your index with CREATE INDEX fall into the following general categories.

Location of Text

The basic prerequisite for an Oracle Text query application is to have a populated text table. The text table is where you store information about your document collection and is required for indexing.

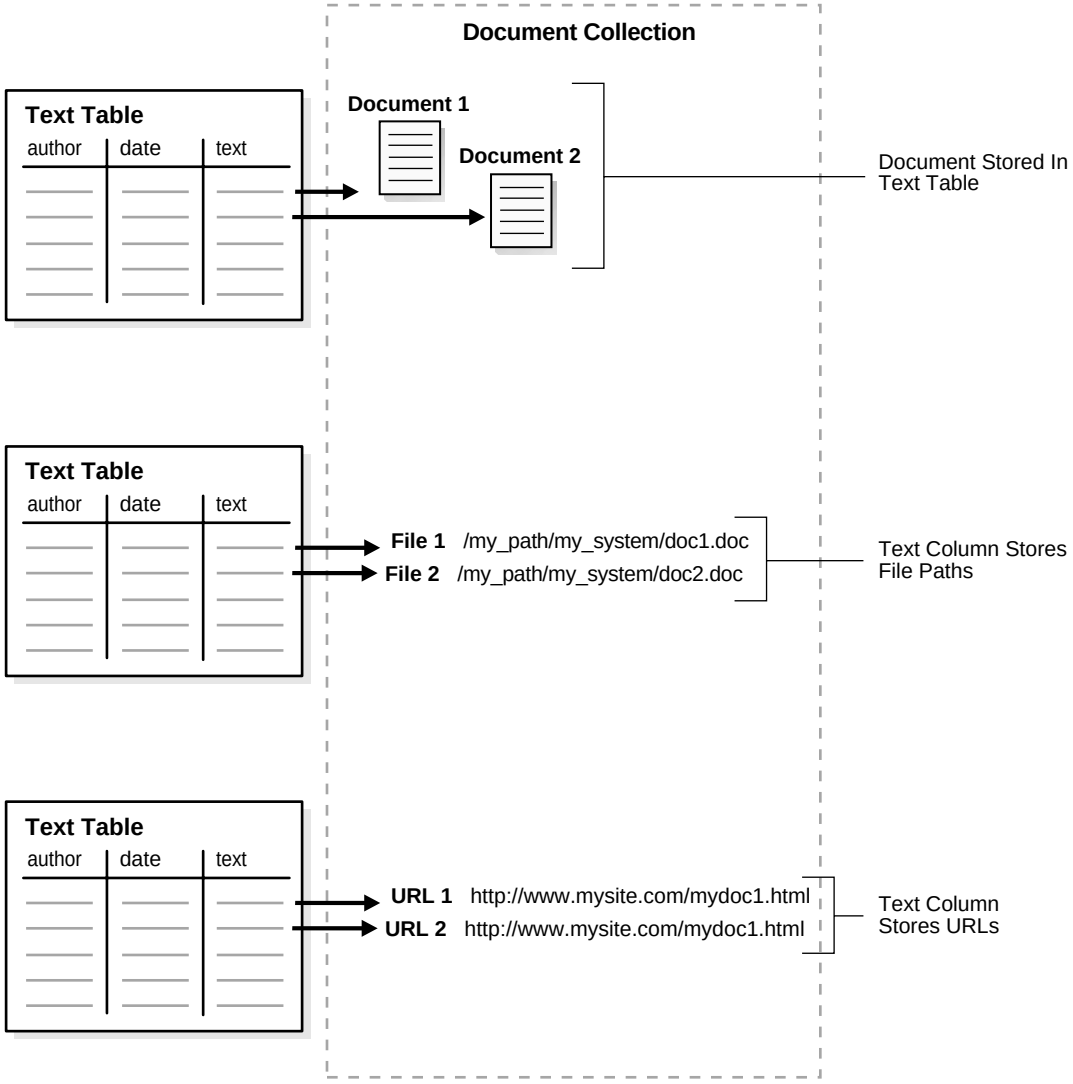
When you create a CONTEXT index, you can populate rows in your text table with one of the following elements:

- text information (can be documents or text fragments)
- path names of documents in your file system
- URLs that specify World Wide Web documents

Figure 3–2 illustrates these different methods.

When creating a CTXCAT or CTXRULE index, only the first method shown is supported.

Figure 3–2 Different Ways of Storing Text



By default, the indexing operation expects your document text to be directly loaded in your text table, which is the first method shown previously.

However, when you create a CONTEXT index, you can specify the other ways of identifying your documents such as with filenames or with URLs by using the corresponding data storage indexing preference.

Supported Column Types

With Oracle Text, you can create a CONTEXT index with columns of type VARCHAR2, CLOB, BLOB, CHAR, BFILE, XMLType, and URIType.

Note: The column types NCLOB, DATE and NUMBER cannot be indexed.

Storing Text in the Text Table

This section discusses how you can store text in directly in your table with the different indexes.

CONTEXT Data Storage You can store documents in your text table in different ways.

You can store documents in one column using the DIRECT_DATASTORE data storage type or over a number of columns using the MULTI_COLUMN_DATASTORE type. When your text is stored over a number of columns, Oracle Text concatenates the columns into a virtual document for indexing.

You can also create master-detail relationships for your documents, where one document can be stored across a number of rows. To create master-detail index, use the DETAIL_DATASTORE data storage type.

You can also store your text in a nested table using the NESTED_DATASTORE type.

Oracle Text supports the indexing of the XMLType datatype which you use to store XML documents.

CTXCAT Data Storage In your text table, you can also store short text fragments such as names, descriptions, and addresses over a number of columns and create a CTXCAT index. A CTXCAT index improves performance for mixed queries.

Storing File Path Names

In your text table, you can store path names to files stored in your file system. When you do so, use the FILE_DATASTORE preference type during indexing. This method of data storage is supported for CONTEXT indexes only.

Storing URLs

You can store URL names to index Web sites. When you do so, use the URL_DATASTORE preference type during indexing. This method of data storage is supported for CONTEXT indexes only.

Storing Associated Document Information

In your text table, you can create additional columns to store structured information that your query application might need, such as primary key, date, description, or author.

Format and Character Set Columns

If your documents are of mixed formats or of mixed character sets, you can create the following additional columns:

- A format column to record the format (TEXT or BINARY) to help filtering during indexing. You can also use the format column to ignore rows for indexing by setting the format column to IGNORE. This is useful for bypassing rows that contain data incompatible with text indexing such as images.
- A character set column to record the document character set on a per-row basis.

When you create your index, you must specify the name of the format or character set column in the parameter clause of CREATE INDEX.

For all rows containing the keywords 'AUTO' or 'AUTOMATIC' in character set or language columns, Oracle Text will apply statistical techniques to determine the character set and language respectively of the documents and modify document indexing appropriately.

Supported Document Formats

Because the system can index most document formats including HTML, PDF, Microsoft Word, and plain text, you can load any supported type into the text column.

When you have mixed formats in your text column, you can optionally include a format column to help filtering during indexing. With the format column you can specify whether a document is binary (formatted) or text (non-formatted such as HTML).

> **See Also:** *Oracle Text Reference* for more information about the supported document formats.

Summary of DATASTORE Types

When you index with CREATE INDEX, you specify the location using the datastore preference. Use the appropriate datastore according to your application.

The following table summarizes the different ways you can store your text with the datastore preference type.

Datastore Type	Use When
DIRECT_DATASTORE	Data is stored internally in a text column. Each row is indexed as a single document. Your text column can be VARCHAR2, CLOB, BLOB, CHAR, or BFILE. XMLType columns are supported for the context index type.
MULTI_COLUMN_DATASTORE	Data is stored in a text table in more than one column. Columns are concatenated to create a virtual document, one document for each row.
DETAIL_DATASTORE	Data is stored internally in a text column. Document consists of one or more rows stored in a text column in a detail table, with header information stored in a master table.
FILE_DATASTORE	Data is stored externally in operating system files. Filenames are stored in the text column, one for each row.
NESTED_DATASTORE	Data is stored in a nested table.
URL_DATASTORE	Data is stored externally in files located on an intranet or the Internet. Uniform Resource Locators (URLs) are stored in the text column.
USER_DATASTORE	Documents are synthesized at index time by a user-defined stored procedure.

Indexing time and document retrieval time will be increased for indexing URLs since the system must retrieve the document from the network.

See Also: Datastore Examples in this chapter.

Document Formats and Filtering

Formatted documents such as Microsoft Word and PDF must be filtered to text to be indexed. The type of filtering the system uses is determined by the FILTER

preference type. By default the system uses the INSO_FILTER filter type, which automatically detects the format of your documents and filters them to text.

Oracle Text can index most formats. Oracle Text can also index columns that contain documents with mixed formats.

No Filtering for HTML

If you are indexing HTML or plain text files, do not use the INSO_FILTER type. For best results, use the NULL_FILTER preference type.

> **See Also:** NULL_FILTER Example: Indexing HTML Documents in this chapter.

Filtering Mixed-Format Columns

If you have a mixed-format column such as one that contains Microsoft Word, plain text, and HTML documents, you can bypass filtering for plain text or HTML by including a format column in your text table. In the format column, you tag each row TEXT or BINARY. Rows that are tagged TEXT are not filtered.

For example, you can tag the HTML and plain text rows as TEXT and the Microsoft Word rows as BINARY. You specify the format column in the CREATE INDEX parameter clause.

Custom Filtering

You can create your own custom filter to filter documents for indexing. You can create either an external filter that is executed from the file system or an internal filter as a PL/SQL or Java stored procedure.

For external custom filtering, use the USER_FILTER filter preference type.

For internal filtering, use the PROCEDURE_FILTER filter type.

> **See Also:** PROCEDURE_FILTER Example on page 3-25.

Bypassing Rows for Indexing

You can bypass rows in your text table that are not to be indexed, such as rows that contain image data. To do so, create a format column in your table and set it to IGNORE. You name the format column in the parameter clause of CREATE INDEX.

Document Character Set

The indexing engine expects filtered text to be in the database character set. When you use the INSO_FILTER filter type, formatted documents are converted to text in the database character set.

If your source is text and your document character set is not the database character set, you can use the INSO_FILTER or CHARSET_FILTER filter type to convert your text for indexing.

Mixed Character Set Columns

If your document set contains documents with different character sets, such as JA16EUC and JA16SJIS, you can index the documents provided you create a charset column. You populate this column with the name of the document character set on a per-row basis. You name the column in the parameter clause of the CREATE INDEX statement.

Document Language

Oracle Text can index most languages. By default, Oracle Text assumes the language of text to index is the language you specify in your database setup.

You use the BASIC_LEXER preference type to index whitespace-delimited languages such as English, French, German, and Spanish. For some of these languages you can enable alternate spelling, composite word indexing, and base letter conversion.

You can also index Japanese, Chinese, and Korean.

> **See Also:** *Oracle Text Reference* to learn more about indexing these languages.

Languages Features Outside BASIC_LEXER

With the BASIC_LEXER, Japanese, Chinese and Korean lexers, Oracle Text provides a lexing solution for most languages. For other languages such as Thai and Arabic, you can create your own lexing solution using the user-defined lexer interface. This interface enables you to create a PL/SQL or Java procedure to process your documents during indexing and querying.

You can also use the user-defined lexer to create your own theme lexing solution or linguistic processing engine.

> **See Also:** *Oracle Text Reference* to learn more about this lexer.

Indexing Multi-language Columns

Oracle Text can index text columns that contain documents of different languages, such as a column that contains documents written in English, German, and Japanese. To index a multi-language column, you need a language column in your text table. Use the `MULTI_LEXER` preference type.

You can also incorporate a multi-language stoplist when you index multi-language columns.

> **See Also:** MULTI_LEXER Example: Indexing a Multi-Language Table in this chapter.

Indexing Special Characters

When you use the `BASIC_LEXER` preference type, you can specify how non-alphanumeric characters such as hyphens and periods are indexed in relation to the tokens that contain them. For example, you can specify that Oracle Text include or exclude hyphen character (-) when indexing a word such as *web-site*.

These characters fall into `BASIC_LEXER` categories according to the behavior you require during indexing. The way the you set the lexer to behave for indexing is the way it behaves for query parsing.

Some of the special characters you can set are as follows:

Printjoins Character

Define a non-alphanumeric character as printjoin when you want this character to be included in the token during indexing.

For example, if you want your index to include hyphens and underscore characters, define them as printjoins. This means that words such as *web-site* are indexed as *web-site*. A query on *website* does not find *web-site*.

> **See Also:** BASIC_LEXER Example: Setting Printjoins Characters in this chapter.

Skipjoins Character

Define a non-alphanumeric character as a skipjoin when you do not want this character to be indexed with the token that contains it.

For example, with the hyphen (-) character defined as a skipjoin, the word *web-site* is indexed as *website*. A query on *web-site* finds documents containing *website* and *web-site*.

Other Characters

Other characters can be specified to control other tokenization behavior such as token separation (startjoins, endjoins, whitespace), punctuation identification (punctuations), number tokenization (numjoins), and word continuation after line-breaks (continuation). These categories of characters have defaults, which you can modify.

> **See Also:** *Oracle Text Reference* to learn more about the BASIC_LEXER.

Case-Sensitive Indexing and Querying

By default, all text tokens are converted to uppercase and then indexed. This results in case-insensitive queries. For example, separate queries on each of the three words *cat*, *CAT*, and *Cat* all return the same documents.

You can change the default and have the index record tokens as they appear in the text. When you create a case-sensitive index, you must specify your queries with exact case to match documents. For example, if a document contains *Cat*, you must specify your query as *Cat* to match this document. Specifying *cat* or *CAT* does not return the document.

To enable or disable case-sensitive indexing, use the mixed_case attribute of the BASIC_LEXER preference.

> **See Also:** *Oracle Text Reference* to learn more about the BASIC_LEXER.

Language Specific Features

You can enable the following language specific features at index time:

Indexing Themes

For English and French, you can index document theme information. A document themes is a concept that is sufficiently developed in the document. Themes can be queried with the ABOUT operator.

You can index theme information in other languages provided you have loaded and compiled a knowledge base for the language.

By default themes are indexed in English and French. You can enable and disable theme indexing with the index_themes attribute of the BASIC_LEXER preference type.

See Also: *Oracle Text Reference* to learn more about the BASIC_LEXER.

ABOUT Queries and Themes in Chapter 4, "Querying".

Base-Letter Conversion for Characters with Diacritical Marks

Some languages contain characters with diacritical marks such as tildes, umlauts, and accents. When your indexing operation converts words containing diacritical marks to their base letter form, queries need not contain diacritical marks to score matches. For example in Spanish with a base-letter index, a query of *energía* matches *energía* and *energia* in the index.

However, with base-letter indexing disabled, a query of *energía* matches only *energía*.

You can enable and disable base-letter indexing for your language with the base_letter attribute of the `BASIC_LEXER` preference type.

See Also: *Oracle Text Reference* to learn more about the `BASIC_LEXER`.

Alternate Spelling

Languages such as German, Danish, and Swedish contain words that have more than one accepted spelling. For instance, in German, *ae* can be substituted for *ä*. The *ae* character pair is known as the alternate form.

By default, Oracle Text indexes words in their alternate forms for these languages. Query terms are also converted to their alternate forms. The result is that these words can be queried with either spelling.

You can enable and disable alternate spelling for your language using the alternate_spelling attribute in the `BASIC_LEXER` preference type.

See Also: *Oracle Text Reference* to learn more about the `BASIC_LEXER`.

Composite Words

German and Dutch text contain composite words. By default, Oracle Text creates composite indexes for these languages. The result is that a query on a term returns words that contain the term as a sub-composite.

For example, in German, a query on the term *Bahnhof* (train station) returns documents that contain *Bahnhof* or any word containing *Bahnhof* as a sub-composite, such as *Hauptbahnhof*, *Nordbahnhof*, or *Ostbahnhof*.

You can enable and disable the creation of composite indexes with the composite attribute of the BASIC_LEXER preference.

> **See Also:** *Oracle Text Reference* to learn more about the BASIC_LEXER.

Korean, Japanese, and Chinese Indexing

You index these languages with specific lexers:

Language	Lexer
Korean	KOREAN_MORPH_LEXER
Japanese	JAPANESE_LEXER
Chinese	CHINESE_VGRAM_LEXER

The KOREAN_MORPH_LEXER has its own set of attributes to control indexing. Features include composite word indexing.

> **See Also:** *Oracle Text Reference* to learn more about these lexers.

Fuzzy Matching and Stemming

Fuzzy matching enables you to match similarly spelled words in queries.

Stemming enables you to match words with the same linguistic root. For example a query on *$speak*, expands to search for all documents that contain *speak, speaks, spoke, and spoken*.

Fuzzy matching and stemming are automatically enabled in your index if Oracle Text supports this feature for your language.

Fuzzy matching is enabled with default parameters for its similarity score lower limit and for its maximum number of expanded terms. At index time you can change these default parameters.

To improve the performance of stem queries, you can create a stem index by enabling the index_stems attribute of the BASIC_LEXER.

> **See Also:** *Oracle Text Reference*.

Better Wildcard Query Performance

Wildcard queries enable you to issue left-truncated, right-truncated and doubly truncated queries, such as *%ing*, *cos%*, or *%benz%*. With normal indexing, these queries can sometimes expand into large word lists, degrading your query performance.

Wildcard queries have better response time when token prefixes and substrings are recorded in the index.

By default, token prefixes and substrings are not recorded in the Oracle Text index. If your query application makes heavy use of wildcard queries, consider indexing token prefixes and substrings. To do so, use the wordlist preference type. The trade-off is a bigger index for improved wildcard searching.

> **See Also:** BASIC_WORDLIST Example: Enabling Substring and Prefix Indexing in this chapter.

Document Section Searching

For documents that have internal structure such as HTML and XML, you can define and index document sections. Indexing document sections enables you to narrow the scope of your queries to within pre-defined sections. For example, you can specify a query to find all documents that contain the term *dog* within a section you define as *Headings*.

Sections must be defined prior to indexing and specified with the section group preference.

Oracle Text provides section groups with system-defined section definitions for HTML and XML. You can also specify that the system automatically create sections from XML documents during indexing.

> **See Also:** Chapter 8, "Document Section Searching"

Stopwords and Stopthemes

A stopword is a word that is not to be indexed. Usually stopwords are low information words in a given language such as *this* and *that* in English.

By default, Oracle Text provides a list of stopwords called a stoplist for indexing a given language. You can modify this list or create your own with the `CTX_DDL` package. You specify the stoplist in the parameter string of `CREATE INDEX`.

A stoptheme is a word that is prevented from being theme-indexed or prevented from contributing to a theme. You can add stopthemes with the CTX_DDL package.

You can search document themes with the ABOUT operator. You can retrieve document themes programatically with the CTX_DOC PL/SQL package.

Multi-Language Stoplists

You can also create multi-language stoplists to hold language-specific stopwords. A multi-language stoplist is useful when you use the MULTI_LEXER to index a table that contains documents in different languages, such as English, German, and Japanese.

At indexing time, the language column of each document is examined, and only the stopwords for that language are eliminated. At query time, the session language setting determines the active stopwords, like it determines the active lexer when using the multi-lexer.

Index Performance

There are factors that influence indexing performance including memory allocation, document format, degree of parallelism, and partitioned tables.

> **See Also:** "Frequently Asked Questions About Indexing Performance" in Chapter 7, "Performance Tuning"

Query Performance and Storage of LOB Columns

If your table contains LOB structured columns that are frequently accessed in queries but rarely updated, you can improve query performance by storing these columns out of line.

> **See Also:** "Does out of line LOB storage of wide base table columns improve performance?" in Chapter 7, "Performance Tuning"

Index Creation

You can create four types of indexes with Oracle Text: CONTEXT, CTXCAT, and CTXRULE, and CTXXPATH

Procedure for Creating a CONTEXT Index

By default, the system expects your documents to be stored in a text column. Once this requirement is satisfied, you can create a text index using the `CREATE INDEX` SQL command as an extensible index of type `CONTEXT`, without explicitly specifying any preferences. The system automatically detects your language, the datatype of the text column, format of documents, and sets indexing preferences accordingly.

> **See Also:** For more information about the out-of-box defaults, see Default CONTEXT Index Example in this chapter.

To create an Oracle Text index, do the following:

1. Optionally, determine your custom indexing preferences, section groups, or stoplists if not using defaults. The following table describes these indexing classes:

Class	Description
Datastore	How are your documents stored?
Filter	How can the documents be converted to plaintext?
Lexer	What language is being indexed?
Wordlist	How should stem and fuzzy queries be expanded?
Storage	How should the index data be stored?
Stop List	What words or themes are not to be indexed?
Section Group	How are documents sections defined?

> **See Also:** Considerations For Indexing in this chapter and *Oracle Text Reference*.

1. Optionally, create your own custom preferences, section groups, or stoplists. See "Creating Preferences" in this chapter.
2. Create the Text index with the SQL command `CREATE INDEX`, naming your index and optionally specifying preferences. See "Creating an Index" in this chapter.

Creating Preferences

You can optionally create your own custom index preferences to override the defaults. Use the preferences to specify index information such as where your files are stored and how to filter your documents. You create the preferences then set the attributes.

Datastore Examples

The following sections give examples for setting direct, multi-column, URL, and file datastores.

> **See Also:** *Oracle Text Reference* for more information about data storage.

Specifying DIRECT_DATASTORE The following example creates a table with a CLOB column to store text data. It then populates two rows with text data and indexes the table using the system-defined preference CTXSYS.DEFAULT_DATASTORE which uses the DIRECT_DATASTORE preference type.

```
create table mytable(id number primary key, docs clob);

insert into mytable values(111555,'this text will be indexed');
insert into mytable values(111556,'this is a default datastore example');
commit;

create index myindex on mytable(docs)
  indextype is ctxsys.context
  parameters ('DATASTORE CTXSYS.DEFAULT_DATASTORE');
```

Specifying MULTI_COLUMN_DATASTORE The following example creates a multi-column datastore preference called my_multi on the three text columns to be concatenated and indexed:

```
begin
ctx_ddl.create_preference('my_multi', 'MULTI_COLUMN_DATASTORE');
ctx_ddl.set_attribute('my_multi', 'columns', 'column1, column2, column3');
end;
```

Specifying URL Data Storage This example creates a URL_DATASTORE preference called my_url to which the http_proxy, no_proxy, and timeout attributes are set. The timeout attribute is set to 300 seconds. The defaults are used for the attributes that are not set.

```
begin
```

```
 ctx_ddl.create_preference('my_url','URL_DATASTORE');
 ctx_ddl.set_attribute('my_url','HTTP_PROXY','www-proxy.us.oracle.com');
 ctx_ddl.set_attribute('my_url','NO_PROXY','us.oracle.com');
 ctx_ddl.set_attribute('my_url','Timeout','300');
end;
```

Specifying File Data Storage The following example creates a data storage preference using the FILE_DATASTORE. This tells the system that the files to be indexed are stored in the operating system. The example uses CTX_DDL.SET_ATTRIBUTE to set the PATH attribute of to the directory /docs.

```
begin
ctx_ddl.create_preference('mypref', 'FILE_DATASTORE');
ctx_ddl.set_attribute('mypref', 'PATH', '/docs');
end;
```

NULL_FILTER Example: Indexing HTML Documents

If your document set is entirely HTML, Oracle recommends that you use the NULL_FILTER in your filter preference, which does no filtering.

For example, to index an HTML document set, you can specify the system-defined preferences for NULL_FILTER and HTML_SECTION_GROUP as follows:

```
create index myindex on docs(htmlfile) indextype is ctxsys.context
  parameters('filter ctxsys.null_filter
  section group ctxsys.html_section_group');
```

PROCEDURE_FILTER Example

Consider a filter procedure CTXSYS.NORMALIZE that you define with the following signature:

```
PROCEDURE NORMALIZE(id IN ROWID, charset IN VARCHAR2, input IN CLOB,
output IN OUT NOCOPY VARCHAR2);
```

To use this procedure as your filter, you set up your filter preference as follows:

```
begin
ctx_ddl.create_preference('myfilt', 'procedure_filter');
ctx_ddl.set_attribute('myfilt', 'procedure', 'normalize');
ctx_ddl.set_attribute('myfilt', 'input_type', 'clob');
ctx_ddl.set_attribute('myfilt', 'output_type', 'varchar2');
ctx_ddl.set_attribute('myfilt', 'rowid_parameter', 'TRUE');
ctx_ddl.set_attribute('myfilt', 'charset_parameter', 'TRUE');
end;
```

BASIC_LEXER Example: Setting Printjoins Characters

Printjoin characters are non-alphanumeric characters that are to be included in index tokens, so that words such as *web-site* are indexed as *web-site*.

The following example sets printjoin characters to be the hyphen and underscore with the BASIC_LEXER:

```
begin
ctx_ddl.create_preference('mylex', 'BASIC_LEXER');
ctx_ddl.set_attribute('mylex', 'printjoins', '_-');
end;
```

To create the index with printjoins characters set as previously shown, issue the following statement:

```
create index myindex on mytable ( docs )
  indextype is ctxsys.context
  parameters ( 'LEXER mylex' );
```

MULTI_LEXER Example: Indexing a Multi-Language Table

You use the MULTI_LEXER preference type to index a column containing documents in different languages. For example, you can use this preference type when your text column stores documents in English, German, and French.

The first step is to create the multi-language table with a primary key, a text column, and a language column as follows:

```
create table globaldoc (
   doc_id number primary key,
   lang varchar2(3),
   text clob
);
```

Assume that the table holds mostly English documents, with some German and Japanese documents. To handle the three languages, you must create three sub-lexers, one for English, one for German, and one for Japanese:

```
ctx_ddl.create_preference('english_lexer','basic_lexer');
ctx_ddl.set_attribute('english_lexer','index_themes','yes');
ctx_ddl.set_attribute('english_lexer','theme_language','english');

ctx_ddl.create_preference('german_lexer','basic_lexer');
ctx_ddl.set_attribute('german_lexer','composite','german');
ctx_ddl.set_attribute('german_lexer','mixed_case','yes');
ctx_ddl.set_attribute('german_lexer','alternate_spelling','german');
```

```
ctx_ddl.create_preference('japanese_lexer','japanese_vgram_lexer');
```

Create the multi-lexer preference:

```
ctx_ddl.create_preference('global_lexer', 'multi_lexer');
```

Since the stored documents are mostly English, make the English lexer the default using CTX_DDL.ADD_SUB_LEXER:

```
ctx_ddl.add_sub_lexer('global_lexer','default','english_lexer');
```

Now add the German and Japanese lexers in their respective languages with CTX_DDL.ADD_SUB_LEXER procedure. Also assume that the language column is expressed in the standard ISO 639-2 language codes, so add those as alternate values.

```
ctx_ddl.add_sub_lexer('global_lexer','german','german_lexer','ger');
ctx_ddl.add_sub_lexer('global_lexer','japanese','japanese_lexer','jpn');
```

Now create the index globalx, specifying the multi-lexer preference and the language column in the parameter clause as follows:

```
create index globalx on globaldoc(text) indextype is ctxsys.context
parameters ('lexer global_lexer language column lang');
```

BASIC_WORDLIST Example: Enabling Substring and Prefix Indexing

The following example sets the wordlist preference for prefix and substring indexing. Having a prefix and sub-string component to your index improves performance for wildcard queries.

For prefix indexing, the example specifies that Oracle Text create token prefixes between three and four characters long:

```
begin
ctx_ddl.create_preference('mywordlist', 'BASIC_WORDLIST');
ctx_ddl.set_attribute('mywordlist','PREFIX_INDEX','TRUE');
ctx_ddl.set_attribute('mywordlist','PREFIX_MIN_LENGTH', '3');
ctx_ddl.set_attribute('mywordlist','PREFIX_MAX_LENGTH', '4');
ctx_ddl.set_attribute('mywordlist','SUBSTRING_INDEX', 'YES');
end;
```

Creating Section Groups for Section Searching

When documents have internal structure such as in HTML and XML, you can define document sections using embedded tags before you index. This enables you to query within the sections using the WITHIN operator. You define sections as part of a section group.

Example: Creating HTML Sections

The following code defines a section group called htmgroup of type HTML_SECTION_GROUP. It then creates a zone section in htmgroup called heading identified by the <H1> tag:

```
begin
ctx_ddl.create_section_group('htmgroup', 'HTML_SECTION_GROUP');
ctx_ddl.add_zone_section('htmgroup', 'heading', 'H1');
end;
```

See Also: Chapter 8, "Document Section Searching"

Using Stopwords and Stoplists

A stopword is a word that is not to be indexed. A stopword is usually a low information word such as *this* or *that* in English.

The system supplies a list of stopwords called a stoplist for every language. By default during indexing, the system uses the Oracle Text default stoplist for your language.

You can edit the default stoplist CTXSYS.DEFAULT_STOPLIST or create your own with the following PL/SQL procedures:

- CTX_DDL.CREATE_STOPLIST
- CTX_DDL.ADD_STOPWORD
- CTX_DDL.REMOVE_STOPWORD

You specify your custom stoplists in the parameter clause of CREATE INDEX.

You can also dynamically add stopwords after indexing with the ALTER INDEX statement.

Multi-Language Stoplists

You can create multi-language stoplists to hold language-specific stopwords. A multi-language stoplist is useful when you use the MULTI_LEXER to index a table that contains documents in different languages, such as English, German, and Japanese.

To create a multi-language stoplist, use the CTX_DLL.CREATE_STOPLIST procedure and specify a stoplist type of MULTI_STOPLIST. You add language specific stopwords with CTX_DDL.ADD_STOPWORD.

Stopthemes and Stopclasses

In addition to defining your own stopwords, you can define stopthemes, which are themes that are not to be indexed. This feature is available for English and French only.

You can also specify that numbers are not to be indexed. A class of alphanumeric characters such a numbers that is not to be indexed is a *stopclass*.

You record your own stopwords, stopthemes, stopclasses by creating a single stoplist, to which you add the stopwords, stopthemes, and stopclasses. You specify the stoplist in the paramstring for CREATE INDEX.

PL/SQL Procedures for Managing Stoplists

You use the following procedures to manage stoplists, stopwords, stopthemes, and stopclasses:

- CTX_DDL.CREATE_STOPLIST
- CTX_DDL.ADD_STOPWORD
- CTX_DDL.ADD_STOPTHEME
- CTX_DDL.ADD_STOPCLASS
- CTX_DDL.REMOVE_STOPWORD
- CTX_DDL.REMOVE_STOPTHEME
- CTX_DDL.REMOVE_STOPCLASS
- CTX_DDL.DROP_STOPLIST

> **See Also:** *Oracle Text Reference* to learn more about using these commands.

Creating an Index

You create an Oracle Text index as an extensible index using the CREATE INDEX SQL command.

You can create four types of indexes:

- CONTEXT
- CTXCAT
- CTXRULE
- CTXXPATH

Creating a CONTEXT Index

The context index type is well-suited for indexing large coherent documents such as MS Word, HTML or plain text. With a context index, you can also customize your index in a variety of ways.

The documents must be loaded in a text table.

CONTEXT Index and DML

A CONTEXT index is not transactional. When you perform inserts, updates, or deletes on the base table, you must explicitly synchronize the index with CTX_DDL.SYNC_INDEX.

See Also: "Synchronizing the Index" in this chapter.

Default CONTEXT Index Example

The following command creates a default context index called myindex on the text column in the docs table:

```
CREATE INDEX myindex ON docs(text) INDEXTYPE IS CTXSYS.CONTEXT;
```

When you use CREATE INDEX without explicitly specifying parameters, the system does the following for all languages by default:

- Assumes that the text to be indexed is stored directly in a text column. The text column can be of type CLOB, BLOB, BFILE, VARCHAR2, or CHAR.
- Detects the column type and uses filtering for the binary column types of BLOB and BFILE. Most document formats are supported for filtering. If your column is plain text, the system does not use filtering.

> **Note:** For document filtering to work correctly in your system, you must ensure that your environment is set up correctly to support the Inso filter.
>
> To learn more about configuring your environment to use the Inso filter, see the *Oracle Text Reference*.

- Assumes the language of text to index is the language you specify in your database setup.
- Uses the default stoplist for the language you specify in your database setup. Stoplists identify the words that the system ignores during indexing.
- Enables fuzzy and stemming queries for your language, if this feature is available for your language.

You can always change the default indexing behavior by creating your own preferences and specifying these custom preferences in the parameter string of CREATE INDEX.

Custom CONTEXT Index Example: Indexing HTML Documents

To index an HTML document set located by URLs, you can specify the system-defined preference for the NULL_FILTER in the CREATE INDEX statement.

You can also specify your section group htmgroup that uses HTML_SECTION_GROUP and datastore my_url that uses URL_DATASTORE as follows:

```
begin
 ctx_ddl.create_preference('my_url','URL_DATASTORE');
 ctx_ddl.set_attribute('my_url','HTTP_PROXY','www-proxy.us.oracle.com');
 ctx_ddl.set_attribute('my_url','NO_PROXY','us.oracle.com');
 ctx_ddl.set_attribute('my_url','Timeout','300');
end;

begin
ctx_ddl.create_section_group('htmgroup', 'HTML_SECTION_GROUP');
ctx_ddl.add_zone_section('htmgroup', 'heading', 'H1');
end;
```

You can then index your documents as follows:

```
create index myindex on docs(htmlfile) indextype is ctxsys.context
parameters('datastore my_url filter ctxsys.null_filter section group
htmgroup');
```

See Also: "Creating Preferences" in this chapter for more examples on creating a custom `context` index.

Creating a CTXCAT Index

The CTXCAT indextype is well-suited for indexing small text fragments and related information. If created correctly, this type of index can give better structured query performance over a CONTEXT index.

CTXCAT Index and DML

A CTXCAT index is transactional. When you perform DML (inserts, updates, and deletes) on the base table, Oracle Text automatically synchronizes the index. Unlike a CONTEXT index, no CTX_DDL.SYNC_INDEX is necessary.

Note: Applications that insert without invoking triggers such as SQL*Loader will not result in automatic index synchronization as described previously.

About CTXCAT Sub-Indexes and Their Costs

A CTXCAT index is comprised of sub-indexes that you define as part of your index set. You create a sub-index on one or more columns to improve mixed query performance.

However, adding sub-indexes to the index set has its costs. The time Oracle Text takes to create a CTXCAT index depends on its total size, and the total size of a CTXCAT index is directly related to

- total text to be indexed
- number of sub-indexes in the index set
- number of columns in the base table that make up the sub-indexes

Having many component indexes in your index set also degrades DML performance since more indexes must be updated.

Because of the added index time and disk space costs for creating a CTXCAT index, carefully consider the query performance benefit each component index gives your application before adding it to your index set.

Creating CTXCAT Sub-indexes

An online auction site that must store item descriptions, prices and bid-close dates for ordered look-up provides a good example for creating a CTXCAT index.

Figure 3–3 Auction Table Schema and CTXCAT Index

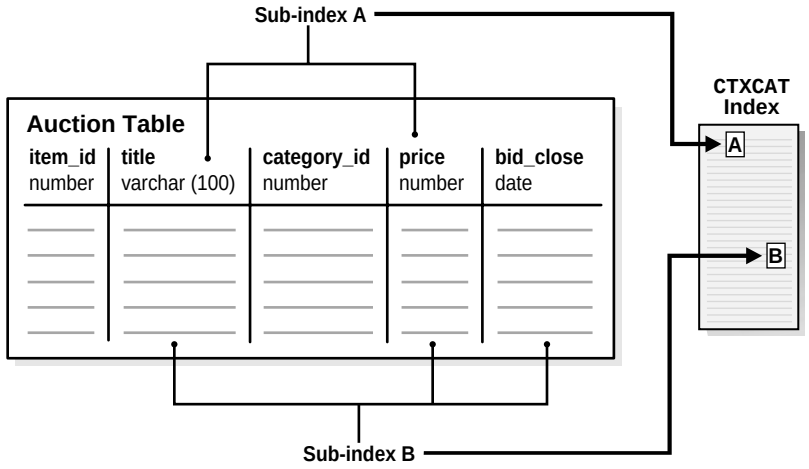


Figure 3–3 shows a table called AUCTION with the following schema:

```
create table auction(
item_id number,
title varchar2(100),
category_id number,
price number,
bid_close date);
```

To create your sub-indexes, create an index set to contain them:

```
begin
ctx_ddl.create_index_set('auction_iset');
end;
```

Next, determine the structured queries your application is likely to issue. The CATSEARCH query operator takes a mandatory text clause and optional structured clause.

In our example, this means all queries include a clause for the `title` column which is the text column.

Assume that the structured clauses fall into the following categories:

Structured Clauses	Sub-index Definition to Serve Query	Category
'price < 200'	'price'	A
'price = 150'		
'order by price'		
'price = 100 order by bid_close'	'price, bid_close'	B
'order by price, bid_close'		

Structured Query Clause Category A The structured query clause contains an expression for only the price column as follows:

```
SELECT FROM auction WHERE CATSEARCH(title, 'camera', 'price < 200')> 0;
SELECT FROM auction WHERE CATSEARCH(title, 'camera', 'price = 150')> 0;
SELECT FROM auction WHERE CATSEARCH(title, 'camera', 'order by price')> 0;
```

These queries can be served using sub-index B, but for efficiency you can also create a sub-index only on `price`, which we call sub-index A:

```
begin
ctx_ddl.add_index('auction_iset','price'); /* sub-index A */
end;
```

Structured Query Clause Category B The structured query clause includes an equivalence expression for `price` ordered by `bid_close`, and an expression for ordering by price and `bid_close` in that order:

```
SELECT FROM auction WHERE CATSEARCH(title, 'camera','price = 100 order by bid_
close')> 0;
SELECT FROM auction WHERE CATSEARCH(title, 'camera','order by price, bid_
close')> 0;
```

These queries can be served with a sub-index defined as follows:

```
begin
ctx_ddl.add_index('auction_iset','price, bid_close'); /* sub-index B */
end;
```

Like a combined b-tree index, the column order you specify with CTX_DDL.ADD_INDEX affects the efficiency and viability of the index scan Oracle Text uses to serve specific queries. For example, if two structured columns p and q have a b-tree index specified as 'p,q', Oracle Text cannot scan this index to sort 'order by q,p'.

Creating CTXCAT Index

The following example combines the previous examples and creates the index set preference with the two sub-indexes:

```
begin
ctx_ddl.create_index_set('auction_iset');
ctx_ddl.add_index('auction_iset','price'); /* sub-index A */
ctx_ddl.add_index('auction_iset','price, bid_close'); /* sub-index B */
end;
```

Figure 3–3 on page 3-33 shows how the sub-indexes A and B are created from the auction table. Each sub-index is a b-tree index on the text column and the named structured columns. For example, sub-index A is an index on the title column and the bid_close column.

You create the combined catalog index with CREATE INDEX as follows:

```
CREATE INDEX auction_titlex ON AUCTION(title) INDEXTYPE IS CTXSYS.CTXCAT
PARAMETERS ('index set auction_iset');
```

> **See Also:** *Oracle Text Reference* to learn more about creating a CTXCAT index with CREATE INDEX.

Creating a CTXRULE Index

You use the CTXRULE index to build a document classification application. In such an application, a stream of incoming documents is classified based on their content.

> **See Also:** Chapter 6, "Document Classification" for more information on document classification and the CTXRULE index.

Document routing is achieved by creating a CTXRULE index on a table or queries. The queries define your categories. You can use the MATCHES operator to classify single documents.

Create a Table of Queries

The first step is to create a table of queries that define your classifications. We create a table myqueries to hold the category name and query text:

```
CREATE TABLE myqueries (
queryid NUMBER PRIMARY KEY,
category VARCHAR2(30),
query VARCHAR2(2000)
);
```

Populate the table with the classifications and the queries that define each. For example, consider a classification for the subjects *US Politics*, *Music*, and *Soccer*.:

```
INSERT INTO myqueries VALUES(1, 'US Politics', 'democrat or republican');
INSERT INTO myqueries VALUES(2, 'Music', 'ABOUT(music)');
INSERT INTO myqueries VALUES(3, 'Soccer', 'ABOUT(soccer)');
```

Using CTX_CLS.TRAIN You can also generate a table of rules (queries) with the CTX_CLS.TRAIN procedure, which takes as input a document training set.

> **See Also:** *Oracle Text Reference* for more information on CTX_CLS.TRAIN.

Create the CTXRULE Index

Use CREATE INDEX to create the CTXRULE index. You can specify lexer, storage, section group, and wordlist parameters if needed:

```
CREATE INDEX ON myqueries(query) INDEXTYPE IS CTXRULE PARAMETERS('lexer lexer_
pref storage storage_pref section group section_pref wordlist wordlist_pref');
```

Classifying a Document

With a CTXRULE index created on query set, you can use the MATCHES operator to classify a document.

Assume that incoming documents are stored in the table news:

```
CREATE TABLE news (
newsid NUMBER,
author VARCHAR2(30),
source VARCHAR2(30),
article CLOB);
```

You can create a before insert trigger with MATCHES to route each document to another table news_route based on its classification:

```
BEGIN
  -- find matching queries
  FOR c1 IN (select category
```

```
              from myqueries
             where MATCHES(query, :new.article)>0)
  LOOP
    INSERT INTO news_route(newsid, category)
      VALUES (:new.newsid, c1.category);
  END LOOP;
END;
```

Index Maintenance

This section describes maintaining your index in the event of an error or indexing failure.

Viewing Index Errors

Sometimes an indexing operation might fail or not complete successfully. When the system encounters an error indexing a row, it logs the error in an Oracle Text view.

You can view errors on your indexes with CTX_USER_INDEX_ERRORS. View errors on all indexes as CTXSYS with CTX_INDEX_ERRORS.

For example to view the most recent errors on your indexes, you can issue:

```
SELECT err_timestamp, err_text FROM ctx_user_index_errors ORDER BY err_
timestamp DESC;
```

To clear the view of errors, you can issue:

```
DELETE FROM ctx_user_index_errors;
```

This view is cleared automatically when you create a new index.

See Also: *Oracle Text Reference* to learn more about these views.

Dropping an Index

You must drop an existing index before you can re-create it with CREATE INDEX.

You drop an index using the DROP INDEX command in SQL.

If you try to create an index with an invalid PARAMETERS string, you still need to drop it before you can re-create it.

For example, to drop an index called newsindex, issue the following SQL command:

```
DROP INDEX newsindex;
```

If Oracle Text cannot determine the state of the index, for example as a result of an indexing malfunction, you cannot drop the index as described previously. Instead use:

```
DROP INDEX newsindex FORCE;
```

See Also: *Oracle Text Reference* to learn more about this command.

Resuming Failed Index

You can sometimes resume a failed index creation operation using the ALTER INDEX command. You typically resume a failed index after you have investigated and corrected the index failure. Not all index failures can be resumed.

Index optimization commits at regular intervals. Therefore if an optimization operation fails, all optimization work up to the commit point has already been saved.

See Also: *Oracle Text Reference* to learn more about the ALTER INDEX command syntax.

Example: Resuming a Failed Index

The following command resumes the indexing operation on newsindex with 10 megabytes of memory:

```
ALTER INDEX newsindex REBUILD PARAMETERS('resume memory 10M');
```

Rebuilding an Index

You can rebuild a valid index using ALTER INDEX. You might rebuild an index when you want to index with a new preference.

Generally, there is no advantage in rebuilding an index over dropping it and re-creating it with CREATE INDEX.

See Also: *Oracle Text Reference* to learn more about the ALTER INDEX command syntax.

Example: Rebuilding and Index

The following command rebuilds the index, replacing the lexer preference with my_lexer.

```
ALTER INDEX newsindex REBUILD PARAMETERS('replace lexer my_lexer');
```

Dropping a Preference

You might drop a custom index preference when you no longer need it for indexing.

You drop index preferences with the procedure CTX_DDL.DROP_PREFERENCE.

Dropping a preference does not affect the index created from the preference.

> **See Also:** *Oracle Text Reference* to learn more about the syntax for the CTX_DDL.DROP_PREFERENCE procedure.

Example

The following code drops the preference my_lexer.

```
begin
ctx_ddl.drop_preference('my_lexer');
end;
```

Managing DML Operations for a CONTEXT Index

DML operations to the base table refer to when documents are inserted, updated or deleted from the base table. This section describes how you can monitor, synchronize, and optimize the Oracle Text CONTEXT index when DML operations occur.

> **Note:** CTXCAT indexes are transactional and thus updated immediately when there is an update to the base table. Manual synchronization as described in this section is not necessary for a CTXCAT index.

Viewing Pending DML

When documents in the base table are inserted, updated, or deleted, their ROWIDs are held in a DML queue until you synchronize the index. You can view this queue with the CTX_USER_PENDING view.

For example, to view pending DML on all your indexes, issue the following statement:

```
SELECT pnd_index_name, pnd_rowid, to_char(pnd_timestamp, 'dd-mon-yyyy
hh24:mi:ss') timestamp FROM ctx_user_pending;
```

This statement gives output in the form:

```
PND_INDEX_NAME                 PND_ROWID          TIMESTAMP
------------------------------ ------------------ --------------------
MYINDEX                        AAADXnAABAAAS3SAAC 06-oct-1999 15:56:50
```

See Also: *Oracle Text Reference* to learn more about this view.

Synchronizing the Index

Synchronizing the index involves processing all pending updates, inserts, and deletes to the base table. You can do this in PL/SQL with the `CTX_DDL.SYNC_INDEX` procedure.

The following example synchronizes the index with 2 megabytes of memory:

```
begin
ctx_ddl.sync_index('myindex', '2M');
end;
```

Setting Background DML

You can set `CTX_DDL.SYNC_INDEX` to run automatically at regular intervals using the `DBMS_JOB.SUBMIT` procedure. Oracle Text includes a SQL script you can use to do this. The location of this script is:

```
$ORACLE_HOME/ctx/sample/script/drjobdml.sql
```

To use this script, you must be the index owner and you must have execute privileges on the `CTX_DDL` package. You must also set the `job_queue_processes` parameter in your Oracle Database initialization file.

For example, to set the index synchronization to run every 360 minutes on myindex, you can issue the following in SQL*Plus:

```
SQL> @drjobdml myindex 360
```

See Also: *Oracle Text Reference* to learn more about the `CTX_DDL.SYNC_INDEX` command syntax.

Index Optimization

Frequent index synchronization can fragment your CONTEXT index. Index fragmentation can adversely affect query response time. You can optimize your CONTEXT index to reduce fragmentation and index size and so improve query performance.

To understand index optimization, you must understand the structure of the index and what happens when it is synchronized.

CONTEXT Index Structure

The CONTEXT index is an inverted index where each word contains the list of documents that contain that word. For example, after a single initial indexing operation, the word DOG might have an entry as follows:

```
DOG DOC1 DOC3 DOC5
```

Index Fragmentation

When new documents are added to the base table, the index is synchronized by adding new rows. Thus if you add a new document (DOC 7) with the word *dog* to the base table and synchronize the index, you now have:

```
DOG DOC1 DOC3 DOC5
DOG DOC7
```

Subsequent DML will also create new rows:

```
DOG DOC1 DOC3 DOC5
DOG DOC7
DOG DOC9
DOG DOC11
```

Adding new documents and synchronizing the index causes index fragmentation. In particular, background DML which synchronizes the index frequently generally produces more fragmentation than synchronizing in batch.

Less frequent batch processing results in longer document lists, reducing the number of rows in the index and hence reducing fragmentation.

You can reduce index fragmentation by optimizing the index in either FULL or FAST mode with CTX_DDL.OPTIMIZE_INDEX.

Document Invalidation and Garbage Collection

When documents are removed from the base table, Oracle Text marks the document as removed but does not immediately alter the index.

Because the old information takes up space and can cause extra overhead at query time, you must remove the old information from the index by optimizing it in FULL mode. This is called garbage collection. Optimizing in FULL mode for garbage collection is necessary when you have frequent updates or deletes to the base table.

Single Token Optimization

In addition to optimizing the entire index, you can optimize single tokens. You can use token mode to optimize index tokens that are frequently searched, without spending time on optimizing tokens that are rarely referenced.

For example, you can specify that only the token *DOG* be optimized in the index, if you know that this token is updated and queried frequently.

An optimized token can improve query response time for the token.

To optimize an index in token mode, you can use CTX_DDL.OPTIMIZE_INDEX.

Viewing Index Fragmentation and Garbage Data

With the CTX_REPORT.INDEX_STATS procedure, you can create a statistical report on your index. The report includes information on optimal row fragmentation, list of most fragmented tokens, and the amount of garbage data in your index. Although this report might take long to run for large indexes, it can help you decide whether to optimize your index.

> **See Also:** *Oracle Text Reference* to learn more about using this procedure.

Examples: Optimizing the Index

To optimize an index, Oracle recommends that you use CTX_DDL.OPTIMIZE_INDEX.

> **See Also:** *Oracle Text Reference* for the CTX_DDL.OPTIMIZE_INDEX command syntax and examples.

4

Querying

This chapter describes Oracle Text querying and associated features. The following topics are covered:

Overview of Queries

The basic Oracle Text query takes a query expression, usually a word with or without operators, as input. Oracle Text returns all documents (previously indexed) that satisfy the expression along with a relevance score for each document. Scores can be used to order the documents in the result set.

To issue an Oracle Text query, use the SQL `SELECT` statement. Depending on the type of index you create, you use either the `CONTAINS` or `CATSEARCH` operator in the `WHERE` clause. You can use these operators programatically wherever you can use the `SELECT` statement, such as in PL/SQL cursors.

Use the `MATCHES` operator to classify documents with a `CTXRULE` index.

Querying with CONTAINS

When you create an index of type `CONTEXT`, you must use the `CONTAINS` operator to issue your query. An index of type `CONTEXT` is suited for indexing collections of large coherent documents.

With the `CONTAINS` operator, you can use a number of operators to define your search criteria. These operators enable you to issue logical, proximity, fuzzy, stemming, thesaurus and wildcard searches. With a correctly configured index, you

can also issue section searches on documents that have internal structure such as HTML and XML.

With CONTAINS, you can also use the ABOUT operator to search on document themes.

CONTAINS SQL Example

In the SELECT statement, specify the query in the WHERE clause with the CONTAINS operator. Also specify the SCORE operator to return the score of each hit in the hitlist. The following example shows how to issue a query:

```
SELECT SCORE(1), title from news WHERE CONTAINS(text, 'oracle', 1) > 0;
```

You can order the results from the highest scoring documents to the lowest scoring documents using the ORDER BY clause as follows:

```
SELECT SCORE(1), title from news
          WHERE CONTAINS(text, 'oracle', 1) > 0
          ORDER BY SCORE(1) DESC;
```

The CONTAINS operator must always be followed by the > 0 syntax, which specifies that the score value returned by the CONTAINS operator must be greater than zero for the row to be returned.

When the SCORE operator is called in the SELECT statement, the CONTAINS operator must reference the score label value in the third parameter as in the previous example.

CONTAINS PL/SQL Example

In a PL/SQL application, you can use a cursor to fetch the results of the query.

The following example issues a CONTAINS query against the NEWS table to find all articles that contain the word *oracle*. The titles and scores of the first ten hits are output.

```
declare
  rowno number := 0;
begin
  for c1 in (SELECT SCORE(1) score, title FROM news
              WHERE CONTAINS(text, 'oracle', 1) > 0
              ORDER BY SCORE(1) DESC)
  loop
    rowno := rowno + 1;
    dbms_output.put_line(c1.title||': '||c1.score);
```

```
    exit when rowno = 10;
  end loop;
end;
```

This example uses a cursor FOR loop to retrieve the first ten hits. An alias *score* is declared for the return value of the SCORE operator. The score and title are output to standard out using cursor dot notation.

Structured Query with CONTAINS

A structured query, also called a mixed query, is a query that has a CONTAINS predicate to query a text column and has another predicate to query a structured data column.

To issue a structured query, you specify the structured clause in the WHERE condition of the SELECT statement.

For example, the following SELECT statement returns all articles that contain the word *oracle* that were written on or after October 1, 1997:

```
SELECT SCORE(1), title, issue_date from news
          WHERE CONTAINS(text, 'oracle', 1) > 0
          AND issue_date >= ('01-OCT-97')
          ORDER BY SCORE(1) DESC;
```

Note: Even though you can issue structured queries with CONTAINS, consider creating a ctxcat index and issuing the query with CATSEARCH, which offers better structured query performance.

Querying with CATSEARCH

When you create an index of type CTXCAT, you must use the CATSEARCH operator to issue your query. An index of type CTXCAT is best suited when your application stores short text fragments in the text column and other associated information in related columns.

For example, an application serving an online auction site might have a table that stores item description in a text column and associated information such as date and price in other columns. With a CTXCAT index, you can create b-tree indexes on one or more of these columns. The result is that when you use the CATSEARCH

operator to search a CTXCAT index, query performance is generally faster for mixed queries.

The operators available for CATSEARCH queries are limited to logical operations such as AND or OR. The operators you can use to define your structured criteria are greater than, less than, equality, BETWEEN, and IN.

CATSEARCH SQL Query

A typical query with CATSEARCH might include a structured clause as follows to find all rows that contain the word *camera* ordered by the bid_close date:

```
SELECT FROM auction WHERE CATSEARCH(title, 'camera', 'order by bid_close desc')>
0;
```

The type of structured query you can issue depends on how you create your sub-indexes.

See Also: "Creating a CTXCAT Index" in Chapter 3, "Indexing".

As shown in the previous example, you specify the structured part of a CATSEARCH query with the third structured_query parameter. The columns you name in the structured expression must have a corresponding sub-index.

For example, assuming that category_id and bid_close have a sub-index in the ctxcat index for the AUCTION table, you can issue the following structured query:

```
SELECT FROM auction WHERE CATSEARCH(title, 'camera', 'category_id=99 order by
bid_close desc')> 0;
```

CATSEARCH Example

The following example shows a field section search against a CTXCAT index using CONTEXT grammar by means of a query template in a CATSEARCH query.

```
-- Create and populate table
create table BOOKS (ID number, INFO varchar2(200), PUBDATE DATE);

insert into BOOKS values(1, '<author>NOAM CHOMSKY</author><subject>CIVIL
   RIGHTS</subject><language>ENGLISH</language><publisher>MIT
   PRESS</publisher>', '01-NOV-2003');

insert into BOOKS values(2, '<author>NICANOR PARRA</author><subject>POEMS
  AND ANTIPOEMS</subject><language>SPANISH</language>
```

```
  <publisher>VASQUEZ</publisher>', '01-JAN-2001');

insert into BOOKS values(1, '<author>LUC SANTE</author><subject>XML
  DATABASE</subject><language>FRENCH</language><publisher>FREE
  PRESS</publisher>', '15-MAY-2002');

commit;

-- Create index set and section group
exec ctx_ddl.create_index_set('BOOK_INDEX_SET');
exec ctx_ddl.add_index('BOOKSET','PUBDATE');

exec ctx_ddl.create_section_group('BOOK_SECTION_GROUP',
      'BASIC_SECTION_GROUP');
exec ctx_ddl.add_field_section('BOOK_SECTION_GROUP','AUTHOR','AUTHOR');
exec ctx_ddl.add_field_section('BOOK_SECTION_GROUP','SUBJECT','SUBJECT');
exec ctx_ddl.add_field_section('BOOK_SECTION_GROUP','LANGUAGE','LANGUAGE');
exec ctx_ddl.add_field_section('BOOK_SECTION_GROUP','PUBLISHER','PUBLISHER');



-- Create index
create index books_index on books(info) indextype is ctxsys.ctxcat
  parameters('index set book_index_set section group book_section_group');

-- Use the index
-- Note that: even though CTXCAT index can be created with field sections, it
-- cannot be accessed using CTXCAT grammar (default for CATSEARCH).
-- We need to use query template with CONTEXT grammar to access field
-- sections with CATSEARCH

select  id, info from books
where catsearch(info,
'<query>
      <textquery grammar="context">
              NOAM within author and english within language
      </textquery>
 </query>',
'order by pubdate')>0;
```

Querying with MATCHES

When you create an index of type CTXRULE, you must use the MATCHES operator to classify your documents. The CTXRULE index is essentially an index on the set of queries that define your classifications.

For example, if you have an incoming stream of documents that need to be routed according to content, you can create a set of queries that define your categories. You create the queries as rows in a text column. It is possible to create this type of table with the CTX_CLS.TRAIN procedure.

You then index the table to create a CTXRULE index. When documents arrive, you use the MATCHES operator to classify each document

See Also: Chapter 6, "Document Classification"

MATCHES SQL Query

A MATCHES query finds all rows in a query table that match a given document. Assuming that a table `querytable` has a CTXRULE index associated with it, you can issue the following query:

```
SELECT classification FROM querytable WHERE MATCHES(query_string,:doc_text) > 0;
```

Note the bind variable `:doc_text` which contains the document CLOB to be classified.

Putting it all together for a simple example:

```
create table queries (
   query_id      number,
   query_string  varchar2(80)
 );

 insert into queries values (1, 'oracle');
 insert into queries values (2, 'larry or ellison');
 insert into queries values (3, 'oracle and text');
 insert into queries values (4, 'market share');

 create index queryx on queries(query_string)
   indextype is ctxsys.ctxrule;

 select query_id from queries
  where matches(query_string,
                'Oracle announced that its market share in databases
```

```
                    increased over the last year.')>0
```

This query will return queries 1 (the word *oracle* appears in the document) and 4 (the phrase *market share* appears in the document) but not 2 (neither the word *larry* nor the word *ellison* appears, and not 3 (there is no text in the document, so it does not match the query).

Note that in this example, the document was passed in as a string for simplicity. Typically, your document would be passed in a bind variable.

The document text used in a matches query can be VARCHAR2 or CLOB. It does not accept BLOB input, so you cannot match filtered documents directly. Instead, you must filter the binary content to CLOB using the INSO filter. For the following example, we make two assumptions: one, that the document data is in bind variable :doc_blob; and, two, that we have already defined a policy, my_policy, that CTX_DOC.POLICY_FILTER can use:

```
declare
  doc_text clob;
begin
  -- create a temporary CLOB to hold the document text
  doc_text := dbms_lob.createtemporary(doc_text, TRUE, DBMS_LOB.SESSION);

  -- create a simple policy for this example
  ctx_ddl.create_preference(preference_name => 'fast_filter',
                      object_name       => 'INSO_FILTER');
  ctx_ddl.set_attribute(preference_name => 'fast_filter',
                      attribute_name    => 'OUTPUT_FORMATTING',
                      attribute_value   => 'FALSE');
  ctx_ddl.create_policy(policy_name     => 'my_policy',
                      filter            => 'fast_filter);

  -- call ctx_doc.policy_filter to filter the BLOB to CLOB data
  ctx_doc.policy_filter('my_policy', :doc_blob, doc_text, FALSE);

  -- now do the matches query using the CLOB version
  for c1 in (select * from queries where matches(query_string, doc_text)>0)
  loop
    -- do what you need to do here
  end loop;

  dbms_lob.freetemporary(doc_text);
end;
```

The procedure CTX_DOC.POLICY_FILTER filters the BLOB into the CLOB data, since you need to get the text into a CLOB to issue a MATCHES query. It takes as one argument the name of a policy you have already created with CTX_DDL.CREATE_POLICY. (See the *Oracle Text Reference* for information on CTX_DOC.POLICY_FILTER.)

If your file is text in the database character set, you can create a BFILE and load it to a CLOB using the function DBMS_LOB.LOADFROMFILE, or you can use UTL_FILE to read the file into a temp CLOB locator.

If your file needs INSO filtering, you can load the file into a BLOB instead, and call CTX_DOC.POLICY_FILTER as previously shown.

> **See Also:** Chapter 6, "Document Classification" for more extended classification examples.

MATCHES PL/SQL Example

The following example assumes that the table of queries profiles has a CTXRULE index associated with it. It also assumes that the table newsfeed contains a set of news articles to be categorized.

This example loops through the newsfeed table, categorizing each article using the MATCHES operator. The results are stored in the results table.

```
PROMPT  Populate the category table based on newsfeed articles
PROMPT
set serveroutput on;
declare
  mypk   number;
  mytitle varchar2(1000);
  myarticles clob;
  mycategory varchar2(100);
  cursor doccur is select pk,title,articles from newsfeed;
  cursor mycur is  select category from profiles where matches(rule,
myarticles)>0;
  cursor rescur is select category, pk, title from results order by category,pk;

begin
  dbms_output.enable(1000000);
  open doccur;
  loop
    fetch doccur into mypk, mytitle, myarticles;
    exit when doccur%notfound;
    open mycur;
    loop
```

```
      fetch mycur into mycategory;
      exit when mycur%notfound;
      insert into results values(mycategory, mypk, mytitle);
    end loop;
    close mycur;
    commit;
  end loop;
  close doccur;
  commit;

end;
/
```

The following example displays the categorized articles by category.

```
PROMPT  display the list of articles for every category
PROMPT
set serveroutput on;

declare
  mypk   number;
  mytitle varchar2(1000);
  mycategory varchar2(100);
  cursor catcur is select category from profiles order by category;
  cursor rescur is select pk, title from results where category=mycategory order
by pk;

begin
  dbms_output.enable(1000000);
  open catcur;
  loop
    fetch catcur into mycategory;
    exit when catcur%notfound;
    dbms_output.put_line('********** CATEGORY: '||mycategory||' *************');
open rescur;
    loop
      fetch rescur into mypk, mytitle;
      exit when rescur%notfound;
dbms_output.put_line('**  ('||mypk||'). '||mytitle);
    end loop;
    close rescur;
    dbms_output.put_line('**');
dbms_output.put_line('*****************************************************');
  end loop;
  close catcur;
```

```
end;
/
```

See Also: Chapter 6, "Document Classification" for more extended classification examples.

Word and Phrase Queries

A word query is a query on a word or phrase. For example, to find all the rows in your text table that contain the word *dog*, you issue a query specifying *dog* as your query term.

You can issue word queries with both `CONTAINS` and `CATSEARCH` SQL operators. However, phrase queries are interpreted differently.

CONTAINS Phrase Queries

If multiple words are contained in a query expression, separated only by blank spaces (no operators), the string of words is considered a phrase and Oracle Text searches for the entire string during a query.

For example, to find all documents that contain the phrase *international law*, you issue your query with the phrase *international law*.

CATSEARCH Phrase Queries

With the CATSEARCH operator, the AND operator is inserted between words in phrases. For example, a query such as international law is interpreted as *international AND law*.

Querying Stopwords

Stopwords are words for which Oracle Text does not create an index entry. They are usually common words in your language that are unlikely to be searched on by themselves.

Oracle Text includes a default list of stopwords for your language. This list is called a stoplist. For example, in English, the words *this* and *that* are defined as stopwords in the default stoplist. You can modify the default stoplist or create new stoplists with the `CTX_DDL` package. You can also add stopwords after indexing with the `ALTER INDEX` statement.

You cannot query on a stopword by itself or on a phrase composed of only stopwords. For example, a query on the word *this* returns no hits when *this* is defined as a stopword.

You can query on phrases that contain stopwords as well as non-stopwords such as *this boy talks to that girl*. This is possible because the Oracle Text index records the position of stopwords even though it does not create an index entry for them.

When you include a stopword within your query phrase, the stopword matches any word. For example, the query:

```
'Jack was big'
```

matches phrases such as *Jack is big* and *Jack grew big* assuming *was* is a stopword. Note that this query matches grew, even though it is not a stopword.

ABOUT Queries and Themes

An ABOUT query is a query on a document theme. A document theme is a concept that is sufficiently developed in the text. For example, an ABOUT query on *US politics* might return documents containing information about US presidential elections and US foreign policy. Documents need not contain the exact phrase *US politics* to be returned.

During indexing, document themes are derived from the knowledge base, which is a hierarchical list of categories and concepts that represents a view of the world. Some examples of themes in the knowledge catalog are concrete concepts such as *jazz music*, *football*, or *Nelson Mandela*. Themes can also be abstract concepts such as *happiness* or *honesty*.

During indexing, the system can also identify and index document themes that are sufficiently developed in the document, but do not exist in the knowledge base.

You can augment the knowledge base to define concepts and terms specific to your industry or query application. When you do so, ABOUT queries are more precise for the added concepts.

ABOUT queries perform best when you create a theme component in your index. Theme components are created by default for English and French.

See Also: *Oracle Text Reference*

Querying Stopthemes

Oracle Text enables you to query on themes with the ABOUT operator. A stoptheme is a theme that is not to be indexed. You can add and remove stopthemes with the CTX_DLL package. You can add stopthemes after indexing with the ALTER INDEX statement.

Query Expressions

A query expression is everything in between the single quotes in the text_query argument of the CONTAINS or CATSEARCH operator. What you can include in a query expression in a CONTAINS query is different from what you can include in a CATSEARCH operator.

CONTAINS Operators

A CONTAINS query expression can contain query operators that enable logical, proximity, thesaural, fuzzy, and wildcard searching. Querying with stored expressions is also possible. Within the query expression, you can use grouping characters to alter operator precedence. This book refers to these operators as the CONTEXT grammar.

With CONTAINS, you can also use the ABOUT query to query document themes.

See Also: "The CONTEXT Grammar" in this chapter.

CATSEARCH Operator

With the CATSEARCH operator, you specify your query expression with the text_query argument and your optional structured criteria with the structured_query argument. The text_query argument enables you to query words and phrases. You can use logical operations, such as logical and, or, and not. This book refers to these operators as the CTXCAT grammar.

If you want to use the much richer set of operators supported by the CONTEXT grammar, you can use the query template feature with CATSEARCH.

With structured_query argument, you specify your structured criteria. You can use the following SQL operations:

- =
- <=
- >=
- >
- <
- IN
- BETWEEN

You can also use ORDER BY clause to order your output.

See Also: "The CTXCAT Grammar" in this chapter.

MATCHES Operator

Unlike CONTAINS and CATSEARCH, MATCHES does not take a query expression as input.

Instead, the MATCHES operator takes a document as input and finds all rows in a query (rule) table that match it. As such, you can use MATCHES to classify documents according to the rules they match.

See Also: "Querying with MATCHES" in this chapter.

Case-Sensitive Searching

Oracle Text supports case-sensitivity for word and ABOUT queries.

Word Queries

Word queries are case-insensitive by default. This means that a query on the term *dog* returns the rows in your text table that contain the word *dog*, *Dog*, or *DOG*.

You can enable case-sensitive searching by enabling the mixed_case attribute in your BASIC_LEXER index preference. With a case-sensitive index, your queries must be issued in exact case. This means that a query on *Dog* matches only documents with *Dog*. Documents with dog or *DOG* are not returned as hits.

Stopwords and Case-Sensitivity If you have case-sensitivity enabled for word queries and you issue a query on a phrase containing stopwords and non-stopwords, you must specify the correct case for the stopwords. For example, a query on *the dog* does not return text that contains *The Dog*, assuming that *the* is a stopword.

ABOUT Queries

ABOUT queries give the best results when your query is formulated with proper case. This is because the normalization of your query is based on the knowledge catalog which is case-sensitive. Attention to case is required especially for words that have different meanings depending on case, such as *turkey* the bird and *Turkey* the country.

However, you need not enter your query in exact case to obtain relevant results from an ABOUT query. The system does its best to interpret your query. For example, if you enter a query of ORACLE and the system does not find this concept in the knowledge catalog, the system might use *Oracle* as a related concept for look-up.

Query Feedback

Feedback information provides broader term, narrower term, and related term information for a specified query with a context index. You obtain this information programatically with the CTX_QUERY.HFEEDBACK procedure.

Broader term, narrower term, and related term information is useful for suggesting other query terms to the user in your query application.

The feedback information returned is obtained from the knowledge base and contains only those terms that are also in the index. This increases the chances that terms returned from HFEEDBACK produce hits over the currently indexed document set.

> **See Also:** *Oracle Text Reference* for more information about using CTX_QUERY.HFEEDBACK

Query Explain Plan

Explain plan information provides a graphical representation of the parse tree for a CONTAINS query expression. You can obtain this information programatically with the CTX_QUERY.EXPLAIN procedure.

Explain plan information tells you how a query is expanded and parsed without having the system execute the query. Obtaining explain information is useful for knowing the expansion for a particular stem, wildcard, thesaurus, fuzzy, soundex, or ABOUT query. Parse trees also show the following information:

- order of execution
- ABOUT query normalization
- query expression optimization
- stop-word transformations
- breakdown of composite-word tokens for supported languages

> **See Also:** *Oracle Text Reference* for more information about using CTX_QUERY.EXPLAIN

Using a Thesaurus in Queries

Oracle Text enables you to define a thesaurus for your query application.

Defining a custom thesaurus enables you to process queries more intelligently. Since users of your application might not know which words represent a topic, you can

define synonyms or narrower terms for likely query terms. You can use the thesaurus operators to expand your query to include thesaurus terms.

See Also: Chapter 9, "Working With a Thesaurus"

Document Section Searching

Section searching enables you to narrow text queries down to sections within documents.

Section searching can be implemented when your documents have internal structure, such as HTML and XML documents. For example, you can define a section for the <H1> tag that enables you to query within this section using the `WITHIN` operator.

You can set the system to automatically create sections from XML documents.

You can also define attribute sections to search attribute text in XML documents.

Note: Section searching is supported for only word queries with a `CONTEXT` index.

See Also: Chapter 8, "Document Section Searching"

Using Query Templating

Query templates are an alternative to the existing query languages. Rather than passing a query string to CONATINS or CATSEARCH, you pass a structured document which contains the query string in a tagged element. Within this document, you can enable additional query features:

- Query Rewrite
- Query Relaxation
- Query Language
- Alternative Scoring
- Alternative Grammar

Query Rewrite

Query applications sometimes parse end user queries, interpreting a query string in one or more ways using different operator combinations. For example, if a user enters a query of *kukui nut*, your application might issue the queries *{kukui nut}* and *{kukui or nut}* in order to increase recall.

The query rewrite feature enables you to submit a single query that expands the original query into the rewritten versions. The results are returned with no duplication.

You specify your rewrite sequences with the query template feature. The rewritten versions of the query are executed efficiently with a single call to CONTAINS or CATSEARCH.

The following template defines a query rewrite sequence. The query of *{kukui nut}* is rewritten as follows:

{kukui} {nut}

{kukui} ; {nut}

{kukui} AND {nut}

{kukui} ACCUM {nut}

The query rewrite template for these transformations is as follows:

```
select id from docs where CONTAINS (text,
 '<query>
   <textquery lang="ENGLISH" grammar="CONTEXT"> kukui nut
     <progression>
       <seq><rewrite>transform((TOKENS, "{", "}", " "))</rewrite></seq>
       <seq><rewrite>transform((TOKENS, "{", "}", " ; "))</rewrite>/seq>
       <seq><rewrite>transform((TOKENS, "{", "}", "AND"))</rewrite></seq>
       <seq><rewrite>transform((TOKENS, "{", "}", "ACCUM"))</rewrite></seq>
     </progression>
   </textquery>
  <score datatype="INTEGER" algorithm="COUNT"/>
</query>')>0;
```

Query Relaxation

Query relaxation enables your application to execute the most restrictive version of a query first, progressively relaxing the query until the required number of hits are obtained.

For example, your application might search first on *black pen* and then the query is relaxed to *black NEAR pen* to obtain more hits.

The following query template defines a query relaxation sequence. The query of *black pen* is issued in sequence as

{black} {pen}

{black} NEAR {pen}

{black} AND {pen}

{black} ACCUM {pen}

The query rewrite template for these transformations is as follows:

```
select id from docs where CONTAINS (text,
 '<query>
   <textquery lang="ENGLISH" grammar="CONTEXT">
     <progression>
       <seq>{black} {pen}</seq>
       <seq>{black} NEAR {pen}</seq>
       <seq>{black} AND {pen}</seq>
       <seq>{black} ACCUM {pen}</seq>
     </progression>
   </textquery>
   <score datatype="INTEGER" algorithm="COUNT"/>
</query>')>0;
```

Query hits are returned in this sequence with no duplication as long as the application needs results.

Query relaxation is most effective when your application needs the top n hits to a query, which you can obtain with the FIRST_ROWS hint or in a PL/SQL cursor.

Using query templating to relax a query as such is more efficient than re-executing a query.

Query Language

When you use the multi-lexer to index a column containing documents in different languages, you can specify which language lexer to use during querying. You do so using the `lang` parameter in the query template.

With the MULTI_LEXER in previous releases, you could only change the query language by altering the session language before executing the query.

```
select id from docs where CONTAINS (text,
```

```
'<query><textquery lang="french">bon soir</textquery></query>')>0;
```

Alternative Scoring

You can use query templating to specify alternative scoring algorithms to use, other than the default.

```
select id from docs where CONTAINS (text,
'<query>
 <textquery grammar="CONTEXT" lang="english"> mustang  </textquery>
 <score datatype="float" algorithm="DEFAULT"/>
</query>')>0
```

Alternative Grammar

Query templating enables you to use the CONTEXT grammar with CATSEARCH queries and vice-versa.

```
select id from docs where CONTAINS (text,
'<query>
  <textquery grammar="CTXCAT">San Diego</textquery>
  <score datatype="integer"/>
</query>')>0;
```

Query Analysis

Oracle Textenables you to create a log of queries and to analyze the queries it contains. For example, suppose you have an application that searches a database of large animals, and your analysis of its queries shows that users are continually searching for the word *mouse*; this analysis might induce you to rewrite your application so that a search for *mouse* redirects the user to a database of small animals instead of simply returning an unsuccessful search.

With query analysis, you can find out

- which queries were made
- which queries were successful
- which queries were unsuccessful
- how many times each query was made

You can combine these factors in various ways, such as determining the 50 most frequent unsuccessful queries made by your application.

You start query logging with CTX_OUTPUT.START_QUERY_LOG. The query log will contain all queries made to all context indexes that the program is using until a CTX_OUTPUT.END_QUERY_LOG procedure is issued. Use CTX_REPORT.QUERY_LOG_SUMMARY to get a report of queries made.

> **See Also:** *Oracle Text Reference* for syntax and examples for these procedures.

Other Query Features

In your query application, you can use other query features such as proximity searching. Table 4–1 lists some of these features.

Table 4–1 Other Oracle Text Query Features

Feature	Description	Implement With
Case Sensitive Searching	Enables you to search on words or phrases exactly as entered in the query. For example, a search on *Roman* returns documents that contain *Roman* and not *roman*.	BASIC_LEXER when you create the index
Base Letter Conversion	Queries words with or without diacritical marks such as tildes, accents, and umlauts. For example, with a Spanish base-letter index, a query of *energía* matches documents containing both *energía* and *energia*.	BASIC_LEXER when you create the index
Word Decompounding (German and Dutch)	Enables searching on words that contain specified term as sub-composite.	BASIC_LEXER when you create the index
Alternate Spelling (German, Dutch, and Swedish)	Searches on alternate spellings of words	BASIC_LEXER when you create the index
Proximity Searching	Searches for words near one another	NEAR operator when you issue the query
Stemming	Searches for words with same root as specified term	$ operator at when you issue the query

Table 4–1 (Cont.) Other Oracle Text Query Features

Feature	Description	Implement With
Fuzzy Searching	Searches for words that have similar spelling to specified term	FUZZY operator when you issue the query
Query Explain Plan	Generates query parse information	CTX_QUERY.EXPLAIN PL/SQL procedure after you index
Hierarchical Query Feedback	Generates broader term, narrower term and related term information for a query	CTX_QUERY.HFEEDBACK PL/SQL procedure after you index.
Browse index	Browses the words around a seed word in the index	CTX_QUERY.BROWSE_WORDS PL/SQL after you index.
Count hits	Counts the number of hits in a query	CTX_QUERY.COUNT_HITS PL/SQL procedure after you index.
Stored Query Expression	Stores the text of a query expression for later reuse in another query.	CTX_QUERY.STORE_SQE PL/SQL procedure after you index.
Thesaural Queries	Uses a thesaurus to expand queries.	Thesaurus operators such as SYN and BT as well as the ABOUT operator. Use CTX_THES package to maintain thesaurus.

The CONTEXT Grammar

The CONTEXT grammar is the default grammar for CONTAINS. With this grammar, you can add complexity to your searches with operators. You use the query operators in your query expression. For example, the logical operator AND enables you to search for all documents that contain two different words. The ABOUT operator enables you to search on concepts.

You can also use the WITHIN operator for section searching, the NEAR operator for proximity searches, the stem, fuzzy, and thesaural operators for expanding a query expression.

With CONTAINS, you can also use the CTXCAT grammar with the query template feature.

The following sections describe some of the Oracle Text operators.

> **See Also:** *Oracle Text Reference* for complete information about using query operators.

ABOUT Query

Use the ABOUT operator in English or French to query on a concept. The query string is usually a concept or theme that represents the idea to be searched on. Oracle Text returns the documents that contain the theme.

Word information and theme information are combined into a single index. To issue a theme query, your index must have a theme component which is created by default in English and French.

You issue a theme query using the ABOUT operator inside the query expression. For example, to retrieve all documents that are about *politics*, write your query as follows:

```
SELECT SCORE(1), title FROM news
          WHERE CONTAINS(text, 'about(politics)', 1) > 0
          ORDER BY SCORE(1) DESC;
```

> **See Also:** *Oracle Text Reference* for more information about using the ABOUT operator.

Logical Operators

Logical operators such as AND or OR allow you to limit your search criteria in a number of ways. Table 4–2 describes some of these operators.

Table 4–2 Logical Operators

Operator	Symbol	Description	Example Expression
AND	&	Use the AND operator to search for documents that contain at least one occurrence of *each* of the query terms. Score returned is the minimum of the operands.	`'cats AND dogs'` `'cats & dogs'`

Table 4–2 (Cont.) Logical Operators

Operator	Symbol	Description	Example Expression
OR	|	Use the OR operator to search for documents that contain at least one occurrence of *any* of the query terms. Score returned is the maximum of the operands.	`'cats | dogs'` `'cats OR dogs'`
NOT	~	Use the NOT operator to search for documents that contain one query term and not another.	To obtain the documents that contain the term *animals* but not *dogs*, use the following expression: `'animals ~ dogs'`
ACCUM	,	Use the ACCUM operator to search for documents that contain at least one occurrence of any of the query terms. The accumulate operator ranks documents according to the total term weight of a document.	The following query returns all documents that contain the terms *dogs*, *cats* and *puppies* giving the highest scores to the documents that contain all three terms: `'dogs, cats, puppies'`
EQUIV	=	Use the EQUIV operator to specify an acceptable substitution for a word in a query.	The following example returns all documents that contain either the phrase *alsatians are big dogs* or *German shepherds are big dogs*: `'German shepherds=alsatians are big dogs'`

Section Searching

Section searching is useful for when your document set is HTML or XML. For HTML, you can define sections using embedded tags and then use the WITHIN operator to search these sections.

For XML, you can have the system automatically create sections for you. You can query with the WITHIN operator or with the INPATH operator for path searching.

See Also: Chapter 8, "Document Section Searching"

Proximity Queries with NEAR and NEAR_ACCUM Operators

You can search for terms that are near to one another in a document with the NEAR operator.

For example, to find all documents where *dog* is within 6 words of *cat*, issue the following query:

```
'near((dog, cat), 6)'
```

The NEAR_ACCUM operator combines the functionality of the NEAR operator with that of the ACCUM operator. Like NEAR, it returns terms that are within a given proximity of each other; however, if one term is not found, it ranks documents according to the frequency of the occurrence of the term that is found.

> **See Also:** *Oracle Text Reference* for more information about using the NEAR and NEAR_ACCUM operators.

Fuzzy, Stem, Soundex, Wildcard and Thesaurus Expansion Operators

You can expand your queries into longer word lists with operators such as wildcard, fuzzy, stem, soundex, and thesaurus.

> **See Also:** *Oracle Text Reference* for more information about using these operators.
>
> "Is it OK to have many expansions in a query?" in Chapter 7, "Performance Tuning"

Using CTXCAT Grammar

You can use the CTXCAT grammar in CONTAINS queries. To do so, use a query template specification in the `text_query` parameter of CONTAINS.

You might take advantage of the CTXCAT grammar when you need an alternative and simpler query grammar.

> **See Also:** *Oracle Text Reference* for more information about using these operators.

Stored Query Expressions

You can use the procedure `CTX_QUERY.STORE_SQE` to store the definition of a query without storing any results. Referencing the query with the `CONTAINS` SQE operator references the definition of the query. In this way, stored query expressions make it easy for defining long or frequently used query expressions.

Stored query expressions are not attached to an index. When you call CTX_QUERY.STORE_SQE, you specify only the name of the stored query expression and the query expression.

The query definitions are stored in the Text data dictionary. Any user can reference a stored query expression.

> **See Also:** *Oracle Text Reference* to learn more about the syntax of CTX_QUERY.STORE_SQE.

Defining a Stored Query Expression

You define and use a stored query expression as follows:

1. Call CTX_QUERY.STORE_SQE to store the queries for the text column. With STORE_SQE, you specify a name for the stored query expression and a query expression.
2. Call the stored query expression in a query expression using the SQE operator. Oracle Text returns the results of the stored query expression in the same way it returns the results of a regular query. The query is evaluated at the time the stored query expression is called.

   You can delete a stored query expression using REMOVE_SQE.

SQE Example

The following example creates a stored query expression called *disaster* that searches for documents containing the words *tornado*, *hurricane*, or *earthquake*:

```
begin
ctx_query.store_sqe('disaster', 'tornado | hurricane | earthquake');
end;
```

To execute this query in an expression, write your query as follows:

```
SELECT SCORE(1), title from news
   WHERE CONTAINS(text, 'SQE(disaster)', 1) > 0
   ORDER BY SCORE(1);
```

> **See Also:** *Oracle Text Reference* to learn more about the syntax of CTX_QUERY.STORE_SQE.

Calling PL/SQL Functions in CONTAINS

You can call user-defined functions directly in the CONTAINS clause as long as the function satisfies the requirements for being named in a SQL statement. The caller must also have EXECUTE privilege on the function.

For example, assuming the function *french* returns the French equivalent of an English word, you can search on the French word for *cat* by writing:

```
SELECT SCORE(1), title from news
   WHERE CONTAINS(text, french('cat'), 1) > 0
   ORDER BY SCORE(1);
```

> **See Also:** *Oracle Database SQL Reference* for more information about creating user functions and calling user functions from SQL,

Optimizing for Response Time

A CONTAINS query optimized for response time provides a fast solution for when you need the highest scoring documents from a hitlist.

The following example returns the first twenty hits to standard out. This example uses the FIRST_ROWS(n) hint and a cursor.

```
declare
cursor c is
  select /*+ FIRST_ROWS(20) */ title, score(1) score
    from news where contains(txt_col, 'dog', 1) > 0 order by score(1) desc;
begin
  for c1 in c
  loop
    dbms_output.put_line(c1.score||':'||substr(c1.title,1,50));
    exit when c%rowcount = 21;
  end loop;
end;
/
```

> **See Also:** "Optimizing Queries for Response Time" in Chapter 7, "Performance Tuning"

Other Factors that Influence Query Response Time

Besides using query hints, there are other factors that can influence query response time such as:

- collection of table statistics
- memory allocation
- sorting
- presence of LOB columns in your base table
- partitioning
- parallelism
- the number term expansions in your query

> **See Also:** "Frequently Asked Questions a About Query Performance" in Chapter 7, "Performance Tuning"

Counting Hits

To count the number of hits returned from a query with only a CONTAINS predicate, you can use CTX_QUERY.COUNT_HITS in PL/SQL or COUNT(*) in a SQL SELECT statement.

If you want a rough hit count, you can use CTX_QUERY.COUNT_HITS in estimate mode (EXACT parameter set to FALSE). With respect to response time, this is the fastest count you can get.

To count the number of hits returned from a query that contains a structured predicate, use the COUNT(*) function in a SELECT statement.

SQL Count Hits Example

To find the number of documents that contain the word *oracle*, issue the query with the SQL COUNT function as follows:

```
SELECT count(*) FROM news WHERE CONTAINS(text, 'oracle', 1) > 0;
```

Counting Hits with a Structured Predicate

To find the number of documents returned by a query with a structured predicate, use COUNT(*) as follows:

```
SELECT COUNT(*) FROM news WHERE CONTAINS(text, 'oracle', 1) > 0 and author =
'jones';
```

PL/SQL Count Hits Example

To find the number of documents that contain the word *oracle*, use COUNT_HITS as follows:

```
declare count number;
begin
 count := ctx_query.count_hits(index_name => my_index, text_query => 'oracle',
                               exact => TRUE);
 dbms_output.put_line('Number of docs with oracle:');
 dbms_output.put_line(count);
end;
```

> **See Also:** *Oracle Text Reference* to learn more about the syntax of CTX_QUERY.COUNT_HITS.

The CTXCAT Grammar

The CTXCAT grammar is the default grammar for CATSEARCH. This grammar supports logical operations such as AND and OR as well as phrase queries.

The CATSEARCH query operators have the following syntax:

Operation	Syntax	Description of Operation
Logical AND	a b c	Returns rows that contain a, b and c.
Logical OR	a | b | c	Returns rows that contain a, b, or c.
Logical NOT	a - b	Returns rows that contain a and not b.
hyphen with no space	a-b	Hyphen treated as a regular character. For example, if the hyphen is defined as skipjoin, words such as *web-site* treated as the single query term *website*. Likewise, if the hyphen is defined as a printjoin, words such as *web-site* treated as *web site* with the space in the CTXCAT query language.
" "	"a b c"	Returns rows that contain the phrase "a b c". For example, entering "Sony CD Player" means return all rows that contain this sequence of words.

Operation	Syntax	Description of Operation
( )	(A B) | C	Parentheses group operations. This query is equivalent to the `CONTAINS` query (A &B) | C.

Using CONTEXT Grammar with CATSEARCH

In addition, you can use the CONTEXT grammar in CATSEARCH queries. To do so, use a query template specification in the text_query parameter.

You might use the CONTAINS grammar as such when you need to issue proximity, thesaurus, or ABOUT queries with a CTXCAT index.

> **See Also:** *Oracle Text Reference* for more information about using these operators.

5

Document Presentation

This chapter describes document presentation. The following topics are covered:

- Highlighting Query Terms
- Obtaining Lists of Themes, Gists, and Theme Summaries
- Document Presentation and Highlighting

Highlighting Query Terms

In Oracle Text query applications, you can present selected documents with query terms highlighted for text queries or with themes highlighted for ABOUT queries.

You can generate three types of output associated with highlighting: a marked-up version of the document, a plain text version of the document (filtered output), and highlight offset information for the document.

The three types of output are generated by three different procedures in the CTX_DOC (document services) PL/SQL package. In addition, you can obtain plain text and HTML versions for each type of output.

Text highlighting

For text highlighting, you supply the query, and Oracle Text highlights words in document that satisfy the query. You can obtain plain-text or HTML highlighting.

Theme Highlighting

For ABOUT queries, the CTX_DOC procedures highlight and mark up words or phrases that best represent the ABOUT query.

CTX_DOC Highlighting Procedures

There are three highlighting procedures in CTX_DOC:

- CTX_DOC.HIGHLIGHT
- CTX_DOC.MARKUP
- CTX_DOC.FILTER
- CTX_DOC.POLICY_FILTER

Highlight Procedure

Highlight offset information is useful for when you write your own custom routines for displaying documents.

To obtain highlight offset information, use the CTX_DOC.HIGHLIGHT procedure. This procedure takes a query and a document, and returns highlight offset information for either plaintext or HTML formats.

With offset information, you can display a highlighted version of document as desired. For example, you can display the document with different font types or colors rather than using the standard plain text markup obtained from CTX_DOC.MARKUP.

> **See Also:** *Oracle Text Reference* for more information about using CTX_DOC.HIGHLIGHT.

Markup Procedure

The CTX_DOC.MARKUP procedure takes a document reference and a query, and returns a marked-up version of the document. The output can be either marked-up plaintext or marked-up HTML.

You can customize the markup sequence for HTML navigation.

CTX_DOC.MARKUP Example The following example is taken from the Web application described in Appendix A, "CONTEXT Query Application". The procedure showDoc takes an HTML document and a query, creates the highlight markup, and outputs the result to an in-memory buffer. It then uses htp.print to display it in the browser.

```
procedure showDoc (p_id in varchar2, p_query in varchar2) is

 v_clob_selected   CLOB;
 v_read_amount     integer;
```

```
 v_read_offset     integer;
 v_buffer          varchar2(32767);
 v_query           varchar(2000);
 v_cursor          integer;

 begin
   htp.p('<html><title>HTML version with highlighted terms</title>');
   htp.p('<body bgcolor="#ffffff">');
   htp.p('<b>HTML version with highlighted terms</b>');

   begin
     ctx_doc.markup (index_name => 'idx_search_table',
                     textkey    => p_id,
                     text_query => p_query,
                     restab     => v_clob_selected,
                     query_id   => 0,
                     starttag   => '<i><font color=red>',
                     endtag     => '</font></i>');

     v_read_amount := 32767;
     v_read_offset := 1;
     begin
      loop
        dbms_lob.read(v_clob_selected,v_read_amount,v_read_offset,v_buffer);
        htp.print(v_buffer);
        v_read_offset := v_read_offset + v_read_amount;
        v_read_amount := 32767;
      end loop;
     exception
      when no_data_found then
         null;
     end;

     exception
      when others then
        null; --showHTMLdoc(p_id);
   end;
end showDoc;
end;
/
show errors
set define on
```

> **See Also:** *Oracle Text Reference* for more information about CTX_DOC.MARKUP.

Filter Procedure

When documents are stored in their native formats such as Microsoft Word, you can use the filter procedure CTX_DOC.FILTER to obtain either a plain text or HTML version of the document.

> **See Also:** *Oracle Text Reference* for more information about CTX_DOC.FILTER.

CTX_DOC.POLICY_FILTER Procedure

This procedure takes a binary document as BLOB and uses the Inso filter to output text to a CLOB. This procedure is useful with MATCHES query, which can use CLOB data as input. The procedure can also be called from a user datastore procedure as a binary to text filter.

Obtaining Lists of Themes, Gists, and Theme Summaries

The following table describes lists of themes, gists, and theme summaries.

Table 5–1 Lists of Themes, Gists, and Theme Summaries

Output Type	Description
List of Themes	A list of the main concepts of a document. You can generate list of themes where each theme is a single word or phrase or where each theme is a hierarchical list of parent themes.
Gist	Text in a document that best represents what the document is about as a whole.
Theme Summary	Text in a document that best represents a given theme in the document.

To obtain this output, you use procedures in the CTX_DOC supplied package. With this package, you can do the following:

- Identify documents by ROWID in addition to primary key
- Store results in-memory for improved performance

Lists of Themes

A list of themes is a list of the main concepts in a document. Use the CTX_DOC.THEMES procedure to generate lists of themes.

> **See Also:** *Oracle Text Reference* to learn more about the command syntax for CTX_DOC.THEMES.

In-Memory Themes

The following example generates the top 10 themes for document 1 and stores them in an in-memory table called the_themes. The example then loops through the table to display the document themes.

```
declare
 the_themes ctx_doc.theme_tab;

begin
 ctx_doc.themes('myindex','1',the_themes, numthemes=>10);
 for i in 1..the_themes.count loop
  dbms_output.put_line(the_themes(i).theme||':'||the_themes(i).weight);
  end loop;
end;
```

Result Table Themes

To create a theme table:

```
create table ctx_themes (query_id number,
                         theme varchar2(2000),
                         weight number);
```

Single Themes To obtain a list of themes where each element in the list is a single theme, issue:

```
begin
ctx_doc.themes('newsindex','34','CTX_THEMES',1,full_themes => FALSE);
end;
```

Full Themes To obtain a list of themes where each element in the list is a hierarchical list of parent themes, issue:

```
begin
ctx_doc.themes('newsindex','34','CTX_THEMES',1,full_themes => TRUE);
end;
```

Gist and Theme Summary

A gist is the text of a document that best represents what the document is about as a whole. A theme summary is the text of a document that best represents a single theme in the document.

Use the procedure CTX_DOC.GIST to generate gists and theme summaries. You can specify the size of the gist or theme summary when you call the procedure.

> **See Also:** *Oracle Text Reference* to learn about the command syntax for CTX_DOC.GIST.

In-Memory Gist

The following example generates a non-default size generic gist of at most 10 paragraphs. The result is stored in memory in a CLOB locator. The code then de-allocates the returned CLOB locator after using it.

```
declare
  gklob clob;
  amt number := 40;
  line varchar2(80);

begin
 ctx_doc.gist('newsindex','34','gklob',1,glevel => 'P',pov => 'GENERIC',
numParagraphs => 10);
  -- gklob is NULL when passed-in, so ctx-doc.gist will allocate a temporary
  -- CLOB for us and place the results there.

  dbms_lob.read(gklob, amt, 1, line);
  dbms_output.put_line('FIRST 40 CHARS ARE:'||line);
  -- have to de-allocate the temp lob
  dbms_lob.freetemporary(gklob);
 end;
```

Result Table Gists

To create a gist table:

```
create table ctx_gist (query_id  number,
                       pov       varchar2(80),
                       gist      CLOB);
```

The following example returns a default sized paragraph level gist for document 34:

```
begin
ctx_doc.gist('newsindex','34','CTX_GIST',1,'PARAGRAPH', pov =>'GENERIC');
```

```
end;
```

The following example generates a non-default size gist of ten paragraphs:

```
begin
ctx_doc.gist('newsindex','34','CTX_GIST',1,'PARAGRAPH', pov =>'GENERIC',
numParagraphs => 10);
end;
```

The following example generates a gist whose number of paragraphs is ten percent of the total paragraphs in document:

```
begin
ctx_doc.gist('newsindex','34','CTX_GIST',1, 'PARAGRAPH', pov =>'GENERIC',
maxPercent => 10);
end;
```

Theme Summary

The following example returns a theme summary on the theme of *insects* for document with textkey 34. The default Gist size is returned.

```
begin
ctx_doc.gist('newsindex','34','CTX_GIST',1, 'PARAGRAPH', pov => 'insects');
end;
```

Document Presentation and Highlighting

Typically, a query application enables the user to view the documents returned by a query. The user selects a document from the hit list and then the application presents the document in some form.

With Oracle Text, you can display a document in different ways. For example, you can present documents with query terms highlighted. Highlighted query terms can be either the words of a word query or the themes of an ABOUT query in English.

You can also obtain gist (document summary) and theme information from documents with the CTX_DOC PL/SQL package.

Table 5–2 describes the different output you can obtain and which procedure to use to obtain each type.

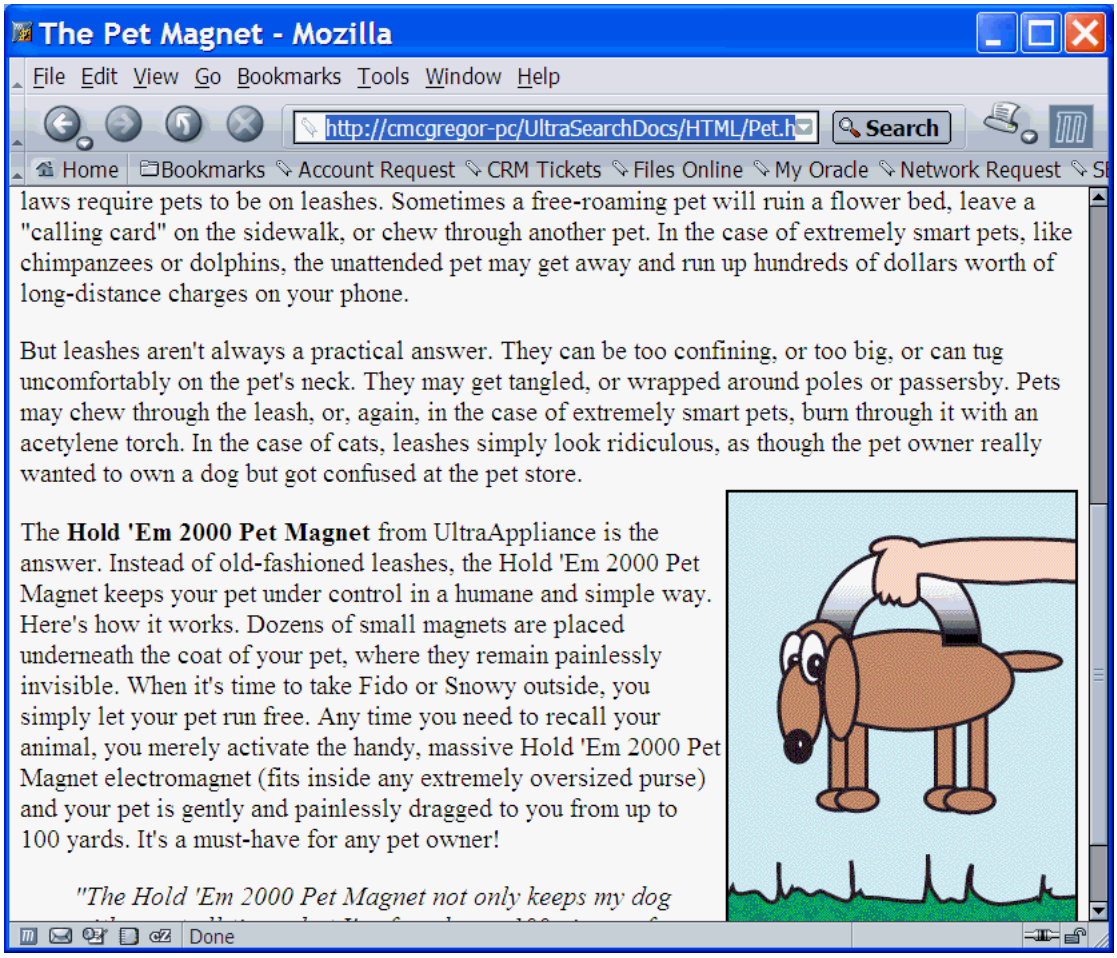
Table 5–2 CTX_DOC Output

Output	Procedure
Plain text version, no highlights	CTX_DOC.FILTER
HTML version of document, no highlights	CTX_DOC.FILTER
Highlighted document, plain text version	CTX_DOC.MARKUP
Highlighted document, HTML version	CTX_DOC.MARKUP
Highlight offset information for plain text version	CTX_DOC.HIGHLIGHT
Highlight offset information for HTML version	CTX_DOC.HIGHLIGHT
Theme summaries and gist of document.	CTX_DOC.GIST
List of themes in document.	CTX_DOC.THEMES

See Also: The *Oracle Text Reference*

Figure 5–1 shows an original document to which we can apply highlighting, gisting, and theme extraction in the following sections.

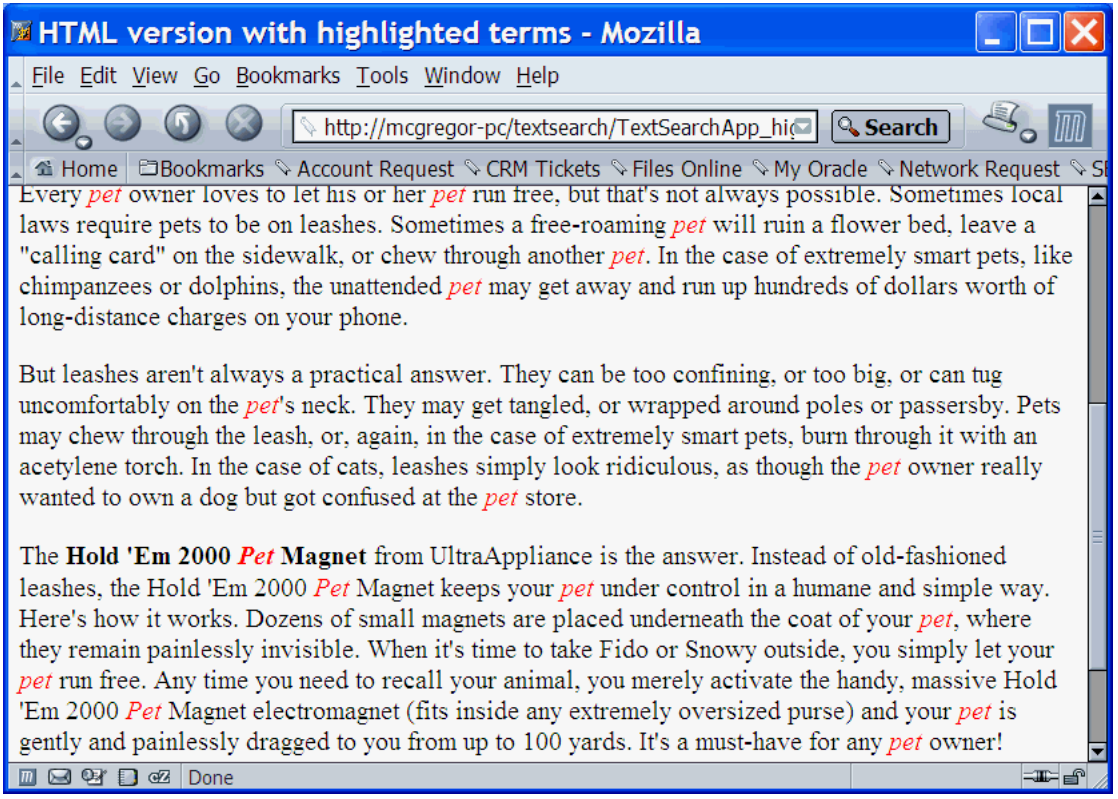
Figure 5–1 Sample Document for Highlighting, Gisting, and Theme Extraction



Highlighting Example

Figure 5–2 is a screen shot of a query application presenting the document shown in Figure 5–1 with the query term *pet* highlighted. This output was created using the text query application produced by a wizard described in Appendix A, "CONTEXT Query Application".

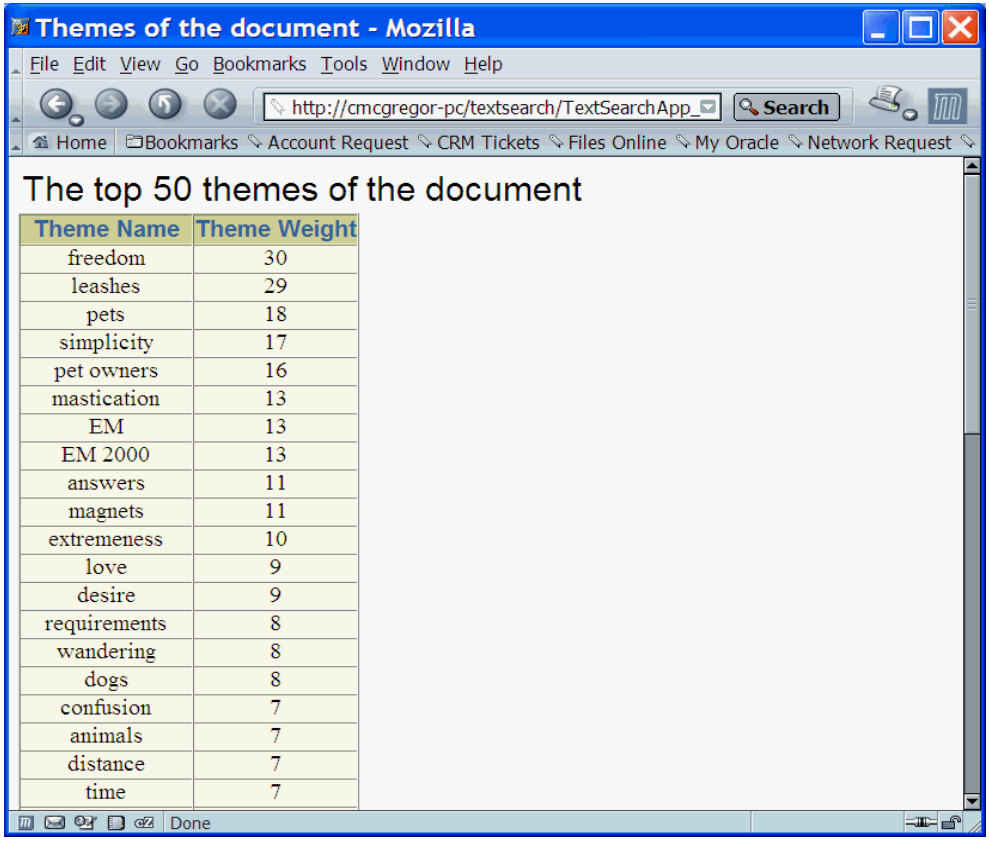
Figure 5–2 Query Application Presenting Highlighted Document



Document List of Themes Example

Figure 5–3 is a screen shot of a query application presenting a list of themes for the document shown in Figure 5–1. This output was created using the text query application produced by a wizard described in Appendix A, "CONTEXT Query Application".

Figure 5–3 Query Application Displaying Document Themes

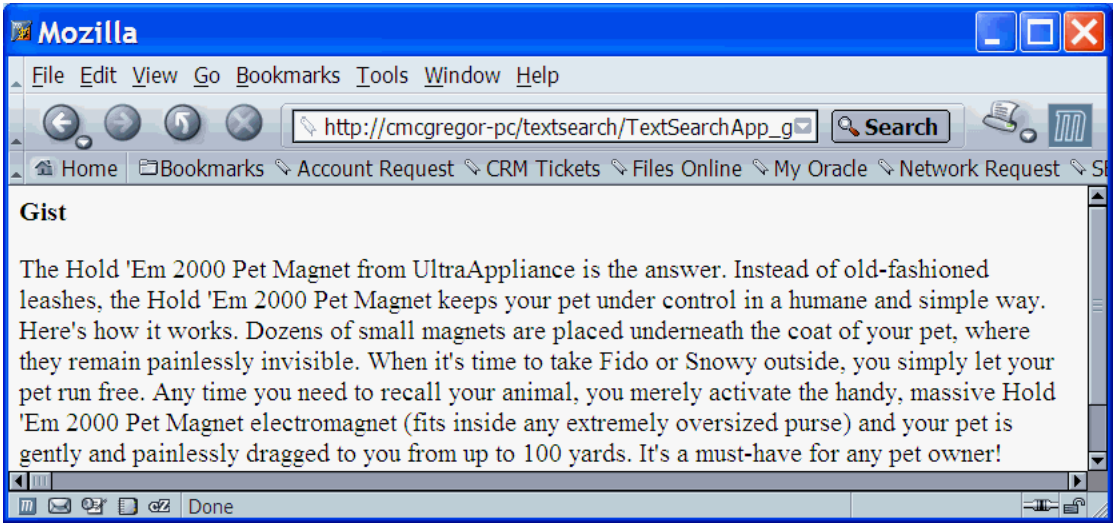


Theme Name	Theme Weight
freedom	30
leashes	29
pets	18
simplicity	17
pet owners	16
mastication	13
EM	13
EM 2000	13
answers	11
magnets	11
extremeness	10
love	9
desire	9
requirements	8
wandering	8
dogs	8
confusion	7
animals	7
distance	7
time	7

Gist Example

Figure 5–4 is a screen shot of a query application presenting a gist of the document shown in Figure 5–1. This output was created using the text query application produced by a wizard described in Appendix A, "CONTEXT Query Application".

Figure 5–4 Query Application Presenting Document Gist



6

Document Classification

This chapter includes the following topics:

Overview

A major problem facing businesses and institutions today is that of information overload. Sorting out useful documents from documents that are not of interest challenges the ingenuity and resources of both individuals and organizations.

One way to sift through numerous documents is to use keyword search engines. However, keyword searches have limitations. One major drawback is that keyword searches don't discriminate by context. In many languages, a word or phrase may have multiple meanings, so a search may result in many matches that are not on the desired topic. For example, a query on the phrase *river bank* might return documents about the Hudson River Bank & Trust Company, because the word *bank* has two meanings.

An alternative strategy is to have human beings sort through documents and classify them by content, but this is not feasible for very large volumes of documents.

Oracle Text offers various approaches to document classification. Under *rule-based classification*, you write the classification rules yourself. With *supervised classification*, Oracle Text creates classification rules based on a set of sample documents that you

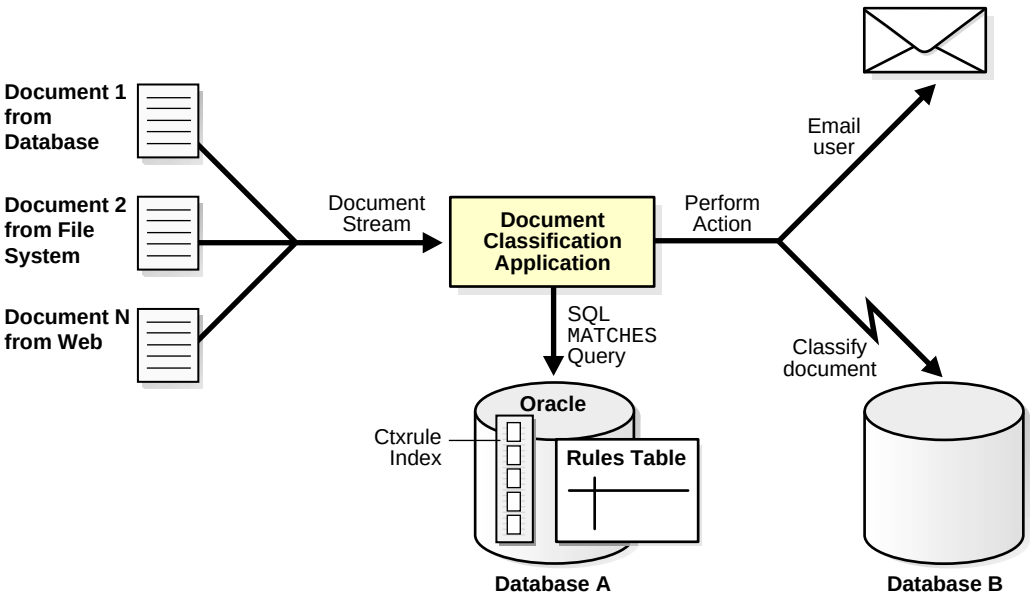
pre-classify. Finally, with *unsupervised classification* (also known as *clustering*), Oracle Text performs all the steps, from writing the classification rules to classifying the documents, for you.

Classification Applications

Oracle Text enables you to build document classification applications. A document classification application performs some action based on document content. Actions include assigning category ids to a document for future lookup or sending a document to a user. The result is a set or stream of categorized documents. Figure 6–1 illustrates how the classification process works.

Oracle Text enables you to create document classification applications in different ways. This chapter defines a typical classification scenario and shows how you can use Oracle Text to build a solution.

Figure 6–1 Overview of a Document Classification Application



Classification Solutions

Oracle Text enables you to classify documents in the following ways:

- Rule-Based Classification. In rule-based classification, you group your documents together, decide on categories, and formulate the rules that define those categories; these rules are actually query phrases. You then index the rules and use the MATCHES operator to classify documents.

  Advantage: Rule-based classification is very accurate for small document sets. Results are always based on what you define, since you write the rules.

  Disadvantages: Defining rules can be tedious for large document sets with many categories. As your document set grows, you may need to write correspondingly more rules.

- Supervised Classification. This method is similar to rule-based classification, but the rule writing step is automated with CTX_CLS.TRAIN. CTX_CLS.TRAIN formulates a set of classification rules from a sample set of pre-classified documents that you provide. As with rule-based classification, you use MATCHES operator to classify documents.

  Oracle Text offers two versions of supervised classification, one using the RULE_CLASSIFIER preference and one using the SVM_CLASSIFIER preference. These are discussed in "Supervised Classification" on page 6-8.

  Advantage: Rules are written for you automatically. This is useful for large document sets.

  Disadvantages:

  - You must assign documents to categories before generating the rules.
  - Rules may not be as specific or accurate as those you write yourself.

- Unsupervised Classification (Clustering). All steps from grouping your documents to writing the category rules are automated with CTX_CLS.CLUSTERING. Oracle Text statistically analyzes your document set and clusters documents according to content.

  Advantages:

  - You don't need to provide either the classification rules or the sample documents as a training set.
  - Helps to discover patterns and content similarities in your document set that you might overlook.

In fact, you can use unsupervised classification when you do not have a clear idea of rules or classifications. One possible scenario is to use unsupervised classification to provide an initial set of categories, and to subsequently build on these through supervised classification.

Disadvantages:

- Clustering might result in unexpected groupings, since the clustering operation is not user-defined, but based on an internal algorithm.
- You do not see the rules that create the clusters.
- The clustering operation is CPU intensive and can take at least the same time as indexing.

Rule-Based Classification

Rule-based classification (sometimes called "simple classification") is the basic way of creating an Oracle Text classification application.

The basic steps for rule-based classification are as follows. Specific steps are explored in greater detail in the example.

1. Create a table for the documents to be classified, and populate it.
2. Create a rule table (also known as a category table). The rule table consists of categories that you name, such as "medicine" or "finance," and the rules that sort documents into those categories.

   These rules are actually queries. For example, you might define the "medicine" category as consisting of documents that include the words "hospital," "doctor," or "disease," so you would set up a rule of the form "hospital OR doctor OR disease." See "CTXRULE Parameters and Limitations" for information on which operators are allowed for queries.
3. Create a CTXRULE index on the rule table.
4. Classify the documents.

Rule-based Classification Example

In this example, we gather news articles on different subjects and then classify them.

Once our rules are created, we can index them and then use the MATCHES statement to classify documents. The steps are as follows:

Step 1 Create schema

We create the tables to store the data. The news_table stores the documents to be classified. The news_categories table stores the categories and rules that define our categories. The news_id_cat table stores the document ids and their associated categories after classification.

```
create table news_table (
       tk number primary key not null,
       title varchar2(1000),
       text clob);

create table news_categories (
        queryid  number primary key not null,
        category varchar2(100),
        query    varchar2(2000));

create table news_id_cat (
        tk number,
        category_id number);
```

Step 2 Load Documents with SQLLDR

In this step, we load the HTML news articles into the news_table using the SQLLDR program. The filenames and titles are read from loader.dat.

```
LOAD DATA
     INFILE 'loader.dat'
     INTO TABLE news_table
     REPLACE
     FIELDS TERMINATED BY ';'
     (tk         INTEGER EXTERNAL,
      title      CHAR,
      text_file  FILLER CHAR,
      text       LOBFILE(text_file) TERMINATED BY EOF)
```

Step 3 Create Categories

In this step, we define our categories and write the rules that define each of our categories. Our categories are the following:

United States	Europe	Middle East
Asia	Africa	Conflicts
Finance	Technology	Consumer Electronics
Latin America	World Politics	U.S. Politics
Astronomy	Paleontology	Health
Natural Disasters	Law	Music News

A rule is a query that selects documents for the category. For example, the category 'Asia' has a rule of 'China or Pakistan or India or Japan'. We insert our rules in the `news_categories` table:

```
insert into news_categories values(1,'United States','Washington or George Bush or Colin
Powell');
insert into news_categories values(2,'Europe','England or Britain or Germany');
insert into news_categories values(3,'Middle East','Israel or Iran or Palestine');

insert into news_categories values(4,'Asia','China or Pakistan or India or Japan');
insert into news_categories values(5,'Africa','Egypt or Kenya or Nigeria');
insert into news_categories values(6,'Conflicts','war or soliders or military or troops');

insert into news_categories values(7,'Finance','profit or loss or wall street');
insert into news_categories values(8,'Technology','software or computer or Oracle or Intel or
IBM or Microsoft');

insert into news_categories values(9,'Consumer electronics','HDTV or electronics');
insert into news_categories values(10,'Latin America','Venezuela or Colombia or Argentina or
Brazil or Chile');

insert into news_categories values(11,'World Politics','Hugo Chavez or George Bush or Tony Blair
or Saddam Hussein or United Nations');

insert into news_categories values(12,'US Politics','George Bush or Democrats or Republicans or
civil rights or Senate or White House');

insert into news_categories values(13,'Astronomy','Jupiter or Earth or star or planet or Orion
or Venus or Mercury or Mars or Milky Way or Telescope or astronomer or NASA or astronaut');

insert into news_categories values(14,'Paleontology','fossils or scientist or paleontologist or
dinosaur or Nature');

insert into news_categories values(15,'Health','stem cells or embryo or health or medical
```

```
or medicine or World Health Organization or AIDS or HIV or virus or centers for disease control
or vaccination');

insert into news_categories values(16,'Natural Disasters','earthquake or hurricane or tornado');
insert into news_categories values(17,'Law','abortion or Supreme Court or illegal or legal or
legislation');
insert into news_categories values(18,'Music News','piracy or anti-piracy or Recording Industry
Association of America or copyright or copy-protection or CDs or music or artist or song');

commit;
```

Step 4 Create the CTXRULE index

In this step, we create a CTXRULE index on our news_categories query column.

```
create index news_cat_idx on news_categories(query)
indextype is ctxsys.ctxrule;
```

Step 5 Classify Documents

To classify the documents, we use the `CLASSIFIER.THIS` PL/SQL procedure (a simple procedure designed for this example), which scrolls through the `news_table`, matches each document to a category, and writes the categorized results into the `news_id_cat` table.

```
create or replace package classifier as
procedure this;
end;
/

show errors

create or replace package body classifier as

 procedure this
 is
  v_document    clob;
  v_item        number;
  v_doc         number;
 begin

  for doc in (select tk, text from news_table)
     loop
        v_document := doc.text;
        v_item := 0;
```

```
        v_doc  := doc.tk;
        for c in (select queryid, category from news_categories
             where matches(query, v_document) > 0 )
          loop
            v_item := v_item + 1;
            insert into news_id_cat values (doc.tk,c.queryid);
          end loop;
   end loop;

 end this;

end;
/
show errors
exec classifier.this
```

CTXRULE Parameters and Limitations

The following considerations apply to indexing a CTXRULE index.

- The BASIC_LEXER, CHINESE_LEXER, JAPANESE_LEXER, and KOREAN_LEXER types are supported for indexing your query set.
- Filter, memory, datastore, and [no]populate parameters are not applicable to index type CTXRULE.
- The CREATE INDEX storage clause is supported for creating the index on the queries.
- Wordlists are supported for stemming operations on your query set.
- Queries for CTXRULE are similar to those of CONTAINS queries. Basic phrasing ("dog house") is supported, as are the following CONTAINS operators: ABOUT, AND, NEAR, NOT, OR, STEM, WITHIN, and THESAURUS. Additionally, wildcards are supported. Section groups are supported for using the MATCHES operator to classify documents. Field sections are also supported; however, CTXRULE does not directly support field queries, so you must use a query rewrite on a CONTEXT query.

See Also: "Creating a CTXRULE Index" in Chapter 3, "Indexing"

Supervised Classification

With supervised classification, you employ the CTX_CLS.TRAIN procedure to automate the rule writing step. CTX_CLS.TRAIN uses a training set of sample

documents to deduce classification rules. This is the major advantage over rule-based classification, in which you must write the classification rules.

However, before you can run the CTX_CLS.TRAIN procedure, you must manually create categories and assign each document in the sample training set to a category. See the *Oracle Text Reference* for more information on CTX_CLS.TRAIN.

When the rules are generated, you index them to create a CTXRULE index. You can then use the MATCHES operator to classify an incoming stream of new documents.

You may choose between two different classification algorithms for supervised classification:

- **Decision Tree classification.** The advantage of Decision Tree classification is that the generated rules are easily observed (and modified).
- **SVM-based classification.** This method uses the Support Vector Machine (SVM) algorithm for creating rules. The advantage of SVM-based classification is that it is often more accurate than Decision Tree classification. The disadvantage is that it generates binary rules, so the rules themselves are opaque.

Decision Tree Supervised Classification

To use Decision Tree classification, you set the preference argument to CTX_CLS.TRAIN to RULE_CLASSIFIER.

This form of classification uses a *decision tree* algorithm for creating rules. Generally speaking, a decision tree is a method of deciding between two (or more, but usually two) choices. In document classification, the choices are "the document matches the training set" or "the document does not match the training set."

A decision tree has a set of attributes that can be tested. In this case, these include:

- words from the document
- stems of words from the document (as an example, the stem of *running* is *run*)
- themes from the document (if themes are supported for the language in use)

The learning algorithm in Oracle Text builds one or more decision trees for each category provided in the training set. These decision trees are then coded into queries suitable for use by a CTXRULE index. As a trivial example, if one category is provided with a training document that consists of "Japanese beetle" and another category with a document reading "Japanese currency," the algorithm may create decision trees based on the words "Japanese," "beetle," and "currency," and classify documents accordingly.

The decision trees include the concept of *confidence*. Each rule that is generated is allocated a percentage value that represents the accuracy of the rule, given the current training set. In trivial examples, this accuracy is almost always 100%, but this merely represents the limitations of the training set. Similarly, the rules generated from a trivial training set may seem to be less than what you might expect, but these are sufficient to distinguish the different categories given the current training set.

The advantage of the Decision Tree method is that it can generate rules that are easily inspected and modified by a human. Using Decision Tree classification makes sense when you want to the computer to generate the bulk of the rules, but you want to fine tune them afterward by editing the rule sets.

Decision Tree Supervised Classification Example

The following SQL example steps through creating your document and classification tables, classifying the documents, and generating the rules. It then goes on to generate rules with `CTX_CLS.TRAIN`.

Rules are then indexed to create `CTXRULE` index and new documents are classified with `MATCHES`.

The general steps for supervised classification can be broken down as follows:

- Create the Category Rules
- Index Rules to Categorize New Documents

Create the Category Rules The `CTX_CLS.TRAIN` procedure requires an input training document set. A training set is a set of documents that have already been assigned a category.

Step 1 Create and populate a training document table

Create and load a table of training documents. This example uses a simple set; three concern fast food and three concern computers.

```
create table docs (
  doc_id number primary key,
  doc_text   clob);

insert into docs values
(1, 'MacTavishes is a fast-food chain specializing in burgers, fries and -
shakes. Burgers are clearly their most important line.');
insert into docs values
(2, 'Burger Prince are an up-market chain of burger shops, who sell burgers -
```

```
and fries in competition with the likes of MacTavishes.');
insert into docs values
(3, 'Shakes 2 Go are a new venture in the low-cost restaurant arena,
specializing in semi-liquid frozen fruit-flavored vegetable oil products.');
insert into docs values
(4, 'TCP/IP network engineers generally need to know about routers, firewalls,
hosts, patch cables networking etc');
insert into docs values
(5, 'Firewalls are used to protect a network from attack by remote hosts,
generally across TCP/IP');
```

Step 2 Create category tables, category descriptions and ids

```
-------------------------------------------------------------------------
-- Create category tables
-- Note that "category_descriptions" isn't really needed for this demo -
-- it just provides a descriptive name for the category numbers in
-- doc_categories
-------------------------------------------------------------------------

create table category_descriptions (
  cd_category    number,
  cd_description varchar2(80));

create table doc_categories (
  dc_category    number,
  dc_doc_id      number,
  primary key (dc_category, dc_doc_id))
  organization index;

-- descriptons for categories

insert into category_descriptions values (1, 'fast food');
insert into category_descriptions values (2, 'computer networking');
```

Step 3 Assign each document to a category

In this case, the fast food documents all go into category 1, and the computer documents into category 2.

```
insert into doc_categories values (1, 1);
insert into doc_categories values (1, 2);
insert into doc_categories values (1, 3);
insert into doc_categories values (2, 4);
insert into doc_categories values (2, 5);
```

Step 4 Create a CONTEXT index to be used by CTX_CLS.TRAIN

Create an Oracle Text preference for the index. This allows us to experiment with the effects of turning themes on and off:

```
exec ctx_ddl.create_preference('my_lex', 'basic_lexer');
exec ctx_ddl.set_attribute    ('my_lex', 'index_themes', 'no');
exec ctx_ddl.set_attribute    ('my_lex', 'index_text',   'yes');

create index docsindex on docs(doc_text) indextype is ctxsys.context
parameters ('lexer my_lex');
```

Step 5 Create the rules table

Create the table that will be populated by the generated rules.

```
create table rules(
  rule_cat_id     number,
  rule_text       varchar2(4000),
  rule_confidence number
);
```

Step 6 Call CTX_CLS.TRAIN procedure to generate category rules

Now call the CTX_CLS.TRAIN procedure to generate some rules. Note all the arguments are the names of tables, columns or indexes previously created in this example. The `rules` table now contains the rules, which you can view.

```
begin
  ctx_cls.train(
    index_name => 'docsindex',
    docid      => 'doc_id',
    cattab     => 'doc_categories',
    catdocid   => 'dc_doc_id',
    catid      => 'dc_category',
    restab     => 'rules',
    rescatid   => 'rule_cat_id',
    resquery   => 'rule_text',
    resconfid  => 'rule_confidence'
  );
end;
/
```

Step 7 Fetch the generated rules, viewed by category

Fetch the generated rules. For convenience's sake, the `rules` table is joined with `category_descriptions` so we can see to which category each rule applies:

```
select cd_description, rule_confidence, rule_text from rules, category_
descriptions where cd_category = rule_cat_id;
```

Index Rules to Categorize New Documents Once the rules are generated, you can test them by first indexing them and then using MATCHES to classify new documents. The process is as follows:

Step 1 Index the rules to create the CTXRULE index

Use CREATE INDEX to create the CTXRULE index on the previously generated rules:

```
create index rules_idx on rules (rule_text) indextype is ctxsys.ctxrule;
```

Step 2 Test an incoming document using MATCHES

```
set serveroutput on;

declare
   incoming_doc clob;
begin
   incoming_doc
       := 'I have spent my entire life managing restaurants selling burgers';
   for c in
     ( select distinct cd_description from rules, category_descriptions
       where cd_category = rule_cat_id
       and matches (rule_text, incoming_doc) > 0) loop
     dbms_output.put_line('CATEGORY: '||c.cd_description);
   end loop;
end;
/
```

SVM-Based Supervised Classification

The second method we can use for training purposes is known as Support Vector Machine (SVM) classification. SVM is a type of machine learning algorithm derived from statistical learning theory. A property of SVM classification is the ability to learn from a very small sample set.

Using the SVM classifier is much the same as using the Decision Tree classifier, with the following differences.

- The preference used in the call to CTX_CLS.TRAIN should be of type SVM_CLASSIFIER instead of RULE_CLASSIFIER. (If you don't want to modify any attributes, you can use the predefined preference CTXSYS.SVM_CLASSIFIER.)

- The CONTEXT index on the table does not have to be populated; that is, you can use the NOPOPULATE keyword. The classifier uses it only to find the source of the text, by means of datastore and filter preferences, and to determine how to process the text, through lexer and sectioner preferences.
- The table for the generated rules must have (as a minimum) these columns:

```
cat_id      number,
type        number,
rule        blob );
```

As you can see, the generated rule is written into a BLOB column. It is therefore opaque to the user, and unlike Decision Tree classification rules, it cannot be edited or modified. The trade-off here is that you often get considerably better accuracy with SVM than with Decision Tree classification.

With SVM classification, allocated memory has to be large enough to load the SVM model; otherwise, the application built on SVM will incur an out-of-memory error. Here is how to calculate the memory allocation:

Minimum memory request (in bytes) = number of unique categories x *number of features*
example: (*value of MAX_FEATURES attributes*) x 8

If necessary to meet the minimum memory requirements, either:

- increase SGA memory (if in shared server mode)
- increase PGA memory (if in dedicated server mode)

SVM-Based Supervised Classification Example

The following example uses SVM-based classification. It uses essentially the same steps as the Decision Tree example. Some differences between the examples:

- In this example, we set the SVM_CLASSIFIER preference with CTX_DDL.CREATE_PREFERENCE rather than setting it in CTX_CLS.TRAIN. (You can do it either way.)
- In this example, our category table includes category descriptions, unlike the category table in the Decision Tree example. (You can do it either way.)
- CTX_CLS.TRAIN takes fewer arguments than in the Decision Tree example, as rules are opaque to the user.

Step 1 Create and populate the training document table:

```
create table doc (id number primary key, text varchar2(2000));
insert into doc values(1,'1 2 3 4 5 6');
```

```
insert into doc values(2,'3 4 7 8 9 0');
insert into doc values(3,'a b c d e f');
insert into doc values(4,'g h i j k l m n o p q r');
insert into doc values(5,'g h i j k s t u v w x y z');
```

Step 2 Create and populate the category table:

```
create table testcategory (
        doc_id number,
        cat_id number,
        cat_name varchar2(100)
         );
insert into testcategory values (1,1,'number');
insert into testcategory values (2,1,'number');
insert into testcategory values (3,2,'letter');
insert into testcategory values (4,2,'letter');
insert into testcategory values (5,2,'letter');
```

Step 3 Create the context index on the document table:

In this case, we create the index without population.

```
create index docx on doc(text) indextype is ctxsys.context
       parameters('nopopulate');
```

Step 4 Set SVM_CLASSIFIER:

This can also be done in `CTX.CLS_TRAIN`.

```
exec ctx_ddl.create_preference('my_classifier','SVM_CLASSIFIER');
exec ctx_ddl.set_attribute('my_classifier','MAX_FEATURES','100');
```

Step 5 Create the result (rule) table:

```
create table restab (
       cat_id  number,
       rule    blob);
```

Step 6 Perform the training:

```
exec ctx_cls.train('docx', 'id','testcategory','doc_id','cat_id',
     'restab','my_classifier');
```

Step 7 Create a CTXRULE index on the rules table:

```
exec ctx_ddl.create_preference('my_filter','NULL_FILTER');
create index restabx on restab (rule)
       indextype is ctxsys.ctxrule
```

```
       parameters ('filter my_filter classifier my_classifier');
```

Now we can classify two unknown documents:

```
select cat_id, match_score(1) from restab
       where matches(rule, '4 5 6',1)>50;

select cat_id, match_score(1) from restab
       where matches(rule, 'f h j',1)>50;

drop table doc;
drop table testcategory;
drop table restab;
exec ctx_ddl.drop_preference('my_classifier');
exec ctx_ddl.drop_preference('my_filter');
```

Unsupervised Classification (Clustering)

With Rule-Based Classification, you write the rules for classifying documents yourself. With Supervised Classification, Oracle Text writes the rules for you, but you must provide a set of training documents that you pre-classify. With *unsupervised classification* (also known as *clustering*), you don't even have to provide a training set of documents.

Clustering is done with the `CTX_CLS.CLUSTERING` procedure. `CTX_CLS.CLUSTERING` assigns documents to different groups, known as *clusters*, according to the similarity with documents already in a cluster. The result is that documents in a cluster are more similar to one another than documents in different clusters. "More similar" basically means "sharing more attributes." As noted in "Decision Tree Supervised Classification" on page 6-9, attributes may consist of simple words (or tokens), word stems, and themes (where supported).

`CTX_CLS.CLUSTERING` assigns output to two tables (which may be in-memory tables):

- A document assignment table containing information about which cluster the procedure assigned a document, and how similar the document is to the assigned cluster. This information takes the form of document identification, cluster identification, and a similarity score between the cluster and an assigned document.
- A cluster description table containing information about what a generated cluster is about. This table contains cluster identification, cluster description

text, a suggested cluster label, number of documents assigned, and a quality score of the cluster.

CTX_CLS.CLUSTERING employs a K-MEAN algorithm to perform clustering. Use the KMEAN_CLUSTER preference to determine how CTX_CLS.CLUSTERING works. For example, you can have CTX_CLS.CLUSTERING produce either flat or hierarchical clustering. Hierarchical clustering produces something akin to a knowledge tree, in which clusters of greater specificity (for instance, whose documents concern dogs or cats) are placed on a layer below more general clusters (such as one for animals).

> **See Also:** For more on cluster types and hierarchical clustering, see the *Oracle Text Reference* guide.

Clustering Example

The following SQL example creates a small collection of documents in the collection table and creates a CONTEXT index. It then creates a document assignment and cluster description table, which are populated with a call to the CLUSTERING procedure. The output would then be viewed with a select statement:

```
set serverout on

/* collect document into a table */
create table collection (id number primary key, text varchar2(4000));
insert into collection values (1, 'Oracle Text can index any document or textual content.');
insert into collection values (2, 'Ultra Search uses a crawler to access documents.');
insert into collection values (3, 'XML is a tag-based markup language.');
insert into collection values (4, 'Oracle10g XML DB treats XML as a native datatype in the
database.');
insert into collection values (5, 'There are three Text index types to cover all text search
needs.');
insert into collection values (6, 'Ultra Search also provides API for content management
solutions.');

create index collectionx on collection(text) indextype is ctxsys.context
parameters('nopopulate');

/* prepare result tables, if you omit this step, procedure will create table automatically */
create table restab (
       docid NUMBER,
       clusterid NUMBER,
       score NUMBER);
```

```
create table clusters (
       clusterid NUMBER,
       descript varchar2(4000),
       label varchar2(200),
       sze   number,
       quality_score number,
       parent number);

/* set the preference */
exec ctx_ddl.drop_preference('my_cluster');
exec ctx_ddl.create_preference('my_cluster','KMEAN_CLUSTERING');
exec ctx_ddl.set_attribute('my_cluster','CLUSTER_NUM','3');

/* do the clustering */
exec ctx_output.start_log('my_log');
exec ctx_cls.clustering('collectionx','id','restab','clusters','my_cluster');
exec ctx_output.end_log;
```

See Also: *Oracle Text Reference* for CTX_CLS.CLUSTERING syntax and examples.

7

Performance Tuning

This chapter discusses how to improve your query and indexing performance. The following topics are covered:

- Optimizing Queries with Statistics
- Optimizing Queries for Response Time
- Optimizing Queries for Throughput
- Tracing
- Parallel Queries
- Tuning Queries with Blocking Operations
- Frequently Asked Questions a About Query Performance
- Frequently Asked Questions About Indexing Performance
- Frequently Asked Questions About Updating the Index

Optimizing Queries with Statistics

Query optimization with statistics uses the collected statistics on the tables and indexes in a query to select an execution plan that can process the query in the most efficient manner. As a general rule, Oracle recommends that you collect statistics on your base table if you are interested in improving your query performance.

The optimizer attempts to choose the best execution plan based on the following parameters:

- the selectivity on the `CONTAINS` predicate
- the selectivity of other predicates in the query

- the CPU and I/O costs of processing the CONTAINS predicates

> **Note:** Importing and exporting of statistics on domain indexes, including Oracle Text indexes, is not supported with the DBMS_STATS package. For more information on importing and exporting statistics, see the *PL/SQL Packages and Types Reference* guide.

The following sections describe how to use statistics with the extensible query optimizer. Optimizing with statistics enables a more accurate estimation of the selectivity and costs of the CONTAINS predicate and thus a better execution plan.

Collecting Statistics

By default, Oracle Text uses the cost-based optimizer to determine the best execution plan for a query. To allow the optimizer to better estimate costs, you can calculate the statistics on the table you query. To do so, issue the following statement:

```
ANALYZE TABLE <table_name> COMPUTE STATISTICS;
```

Alternatively, you can estimate the statistics on a sample of the table as follows:

```
ANALYZE TABLE <table_name> ESTIMATE STATISTICS 1000 ROWS;
```

or

```
ANALYZE TABLE <table_name> ESTIMATE STATISTICS 50 PERCENT;
```

You can also collect statistics in parallel with the DBMS_STATS.GATHER_TABLE_STATS procedure.

```
begin
DBMS_STATS.GATHER_TABLE_STATS('owner', 'table_name',
                                       estimate_percent=>50,
                                       block_sample=>TRUE,
                                       degree=>4) ;
end  ;
```

These statements collect statistics on all the objects associated with table_name including the table columns and any indexes (b-tree, bitmap, or Text domain) associated with the table.

To re-collect the statistics on a table, you can issue the ANALYZE command as many times as necessary or use the DBMS_STATS package

By collecting statistics on the Text domain index, the Oracle Database cost-based optimizer is able to do the following:

- estimate the selectivity of the CONTAINS predicate
- estimate the I/O and CPU costs of using the Text index, that is, the cost of processing the CONTAINS predicate using the domain index
- estimate the I/O and CPU costs of each invocation of CONTAINS

Knowing the selectivity of a CONTAINS predicate is useful for queries that contain more than one predicate, such as in structured queries. This way the cost-based optimizer can better decide whether to use the domain index to evaluate CONTAINS or to apply the CONTAINS predicate as a post filter.

See Also:

Oracle Database SQL Reference and *Oracle Database Performance Tuning Guide* for more information about the ANALYZE command.

PL/SQL Packages and Types Reference for information about DBMS_STATS package.

Example

Consider the following structured query:

```
select score(1) from tab where contains(txt, 'freedom', 1) > 0 and author =
'King' and year > 1960;
```

Assume the author column is of type VARCHAR2 and the year column is of type NUMBER. Assume that there is a b-tree index on the author column.

Also assume that the structured author predicate is highly selective with respect to the CONTAINS predicate and the year predicate. That is, the structured predicate (author = 'King') returns a much smaller number of rows with respect to the year and CONTAINS predicates individually, say 5 rows returned versus 1000 and 1500 rows respectively.

In this situation, Oracle Text can execute this query more efficiently by first doing a b-tree index range scan on the structured predicate (author = 'King'), followed by a table access by rowid, and then applying the other two predicates to the rows returned from the b-tree table access.

Note: When statistics are not collected for a Text index, the cost-based optimizer assumes low selectivity and index costs for the CONTAINS predicate.

Re-Collecting Statistics

After synchronizing your index, you can re-collect statistics on a single index to update the cost estimates.

If your base table has been re-analyzed before the synchronization, it is sufficient to analyze the index after the synchronization without re-analyzing the entire table.

To do so, you can issue any of the following statements:

```
ANALYZE INDEX <index_name> COMPUTE STATISTICS;
or

ANALYZE INDEX <index_name> ESTIMATE STATISTICS SAMPLE 1000 ROWS;

or

ANALYZE INDEX <index_name> ESTIMATE STATISTICS SAMPLE 50 PERCENT;
```

Deleting Statistics

You can delete the statistics associated with a table by issuing:

```
ANALYZE TABLE <table_name> DELETE STATISTICS;
```

You can delete statistics on one index by issuing the following statement:

```
ANALYZE INDEX <index_name> DELETE STATISTICS;
```

Optimizing Queries for Response Time

By default, Oracle Text optimizes queries for throughput. This results in queries returning all rows in shortest time possible.

However, in many cases, especially in a Web application scenario, queries must be optimized for response time, when you are only interested in obtaining the first few hits of a potentially large hitlist in the shortest time possible.

The following sections describe some ways to optimize CONTAINS queries for response time:

- Improved Response Time with FIRST_ROWS(n) for ORDER BY Queries
- Improved Response Time using Local Partitioned CONTEXT Index
- Improved Response Time with Local Partitioned Index for Order by Score

Other Factors that Influence Query Response Time

There are other factors that can influence query response time such as:

- collection of table statistics
- memory allocation
- sorting
- presence of LOB columns in your base table
- partitioning
- parallelism
- the number term expansions in your query

> **See Also:** "Frequently Asked Questions a About Query Performance" in this chapter.

Improved Response Time with FIRST_ROWS(n) for ORDER BY Queries

When you need the first rows of an ORDER BY query, Oracle recommends that you use this new fully cost-based hint in place of FIRST_ROWS.

Note: As this hint is cost-based, Oracle recommends that you collect statistics on your tables before you use this hint. See "Collecting Statistics" in this chapter.

You use the FIRST_ROWS(n) in cases where you want the first n number of rows in the shortest possible time. For example, consider the following PL/SQL block that uses a cursor to retrieve the first 10 hits of a query and uses the `FIRST_ROWS(n)` hint to optimize the response time:

```
declare
cursor c is
```

```
select /* FIRST_ROWS(10) */ article_id from articles_tab
   where contains(article, 'Oracle')>0 order by pub_date desc;

begin
for i in c
loop
insert into t_s values(i.pk, i.col);
exit when c%rowcount > 11;
end loop;
end;
/
```

The cursor c is a SELECT statement that returns the rowids that contain the word *test* in sorted order. The code loops through the cursor to extract the first 10 rows. These rows are stored in the temporary table t_s.

With the FIRST_ROWS hint, Oracle Text instructs the Text index to return rowids in score-sorted order, if possible.

Without the hint, Oracle Text sorts the rowids after the Text index has returned *all* the rows in unsorted order that satisfy the CONTAINS predicate. Retrieving the entire result set as such takes time.

Since only the first 10 hits are needed in this query, using the hint results in better performance.

> **Note:** Use the FIRST_ROWS(n) hint when you need only the first few hits of a query. When you need the entire result set, do not use this hint as it might result in poor performance.

About the FIRST_ROWS Hint

You can also optimize for response time using the related FIRST_ROWS hint. Like FIRST_ROWS(n), when queries are optimized for response time, Oracle Text returns the first rows in the shortest time possible.

For example, you can use this hint as follows

```
select /*+ FIRST_ROWS */ pk, score(1), col from ctx_tab
           where contains(txt_col, 'test', 1) > 0 order by score(1) desc;
```

However, this hint is only rule-based. This means that Oracle Text always chooses the index which satisfies the ORDER BY clause. This might result in sub-optimal performance for queries in which the CONTAINS clause is very selective. In these

cases, Oracle recommends that you use the `FIRST_ROWS(n)` hint, which is fully cost-based.

Improved Response Time using Local Partitioned CONTEXT Index

Partitioning your data and creating local partitioned indexes can improve your query performance. On a partitioned table, each partition has its own set of index tables. Effectively, there are multiple indexes, but the results from each are combined as necessary to produce the final result set.

You create the CONTEXT index using the LOCAL keyword:

```
CREATE INDEX index_name ON table_name (column_name)
INDEXTYPE IS ctxsys.context
PARAMETERS ('...')
LOCAL
```

With partitioned tables and indexes, you can improve performance of the following types of queries:

- Range Search on Partition Key Column
- ORDER BY Partition Key Column

Range Search on Partition Key Column

This is a query that restricts the search to a particular range of values on a column that is also the partition key. For example, consider a query on a date range:

```
SELECT storyid FROM storytab WHERE CONTAINS(story, 'oliver')>0 and pub_date
BETWEEN '1-OCT-93' AND '1-NOV-93';
```

If the date range is quite restrictive, it is very likely that the query can be satisfied by only looking in a single partition.

ORDER BY Partition Key Column

This is a query that requires only the first N hits and the ORDER BY clause names the partition key. Consider an ORDER BY query on a price column to fetch the first 20 hits such as:

```
SELECT * FROM (
SELECT itemid FROM item_tab WHERE CONTAINS(item_desc, 'cd player')>0 ORDER BY
price)
WHERE ROWNUM < 20;
```

In this example, with the table partitioned by price, the query might only need to get hits from the first partition to satisfy the query.

Improved Response Time with Local Partitioned Index for Order by Score

Using the FIRST_ROWS hint on a local partitioned index might result in poor performance, especially when you order by score. This is because all hits to the query across all partitions must be obtained before the results can be sorted.

You can work around this by using an inline view when you use the FIRST_ROWS hint. Specifically, you can use the FIRST_ROWS hint to improve query performance on a local partitioned index under the following conditions:

- The text query itself including the order by SCORE() clause is expressed as an in-line view.
- The text query inside the in-line view contains the FIRST_ROWS or DOMAIN_INDEX_SORT hint.
- The query on the in-line view has ROWNUM predicate limiting number of rows to fetch from the view.

For example, if you have the following text query and local text index created on a partitioned table doc_tab:

```
select doc_id, score(1) from doc_tab
   where contains(doc, 'oracle', 1)>0
   order by score(1) desc;
```

and you are only interested in fetching top 20 rows, you can rewrite the query to

```
select * from
     (select /*+ FIRST_ROWS */ doc_id, score(1) from doc_tab
         where contains(doc, 'oracle', 1)>0 order by score(1) desc)
where rownum < 21;
```

See Also: *Oracle Database Performance Tuning Guide* for more information about the query optimizer and using hints such as FIRST_ROWS.

For more information about the EXPLAIN PLAN command, *Oracle Database Performance Tuning Guide* and *Oracle Database SQL Reference*.

Optimizing Queries for Throughput

Optimizing a query for throughput returns all hits in the shortest time possible. This is the default behavior.

The following sections describe how you can explicitly optimize for throughput.

CHOOSE and ALL ROWS Modes

By default, queries are optimized for throughput under the CHOOSE and ALL_ROWS modes. When queries are optimized for throughput, Oracle Text returns *all* rows in the shortest time possible.

FIRST_ROWS Mode

In FIRST_ROWS mode, the Oracle Database optimizer optimizes for fast response time by having the Text domain index return score-sorted rows, if possible. This is the default behavior when you use the FIRST_ROWS hint.

If you want to optimize for better throughput under FIRST_ROWS, you can use the DOMAIN_INDEX_NO_SORT hint. Better throughput means you are interested in getting all the rows to a query in the shortest time.

The following example achieves better throughput by not using the Text domain index to return score-sorted rows. Instead, Oracle Text sorts the rows after all the rows that satisfy the CONTAINS predicate are retrieved from the index:

```
select /*+ FIRST_ROWS DOMAIN_INDEX_NO_SORT */ pk, score(1), col from ctx_tab
           where contains(txt_col, 'test', 1) > 0 order by score(1) desc;
```

> **See Also:** *Oracle Database Performance Tuning Guide* for more information about the query optimizer and using hints such as FIRST_ROWS and CHOOSE.

Tracing

Oracle Text includes a *tracing* facility that enables you to identify bottlenecks in indexing and querying.

Oracle Text provides a set of predefined *traces*. Each trace is identified by a unique number. There is also a symbol in CTX_OUTPUT for this number.

Each trace measures a specific numeric quantity—for instance, the number of $I rows selected during text queries.

Traces are cumulative counters, so usage is as follows:

1. The user enables a trace.
2. The user performs one or more operations. Oracle Text measures activities and accumulates the results in the trace.
3. The user retrieves the trace value, which is the total value across all operations done in step 2.
4. The user resets the trace to 0.
5. The user starts over at Step 2.

So, for instance, if in step 2 the user runs two queries, and query 1 selects 15 rows from `$I`, and query 2 selects 17 rows from `$I`, then in step 3 the value of the trace would be 32 (15 + 17).

Traces are associated with a session—they can measure operations that take place within a single session, and, conversely, cannot make measurements across sessions.

During parallel sync or optimize, the trace profile will be copied to the slave sessions if and only if tracing is currently enabled. Each slave will accumulate its own traces and implicitly write all trace values to the slave logfile before termination.

For more information on tracing, see the *Oracle Text Reference*.

Parallel Queries

Oracle Text supports parallel query on a local CONTEXT index. That is, based on the parallel degree of the index and various system attributes, Oracle Text determines number of parallel query slaves to be spawned to process the index. Each parallel query slave will process one or more index partitions. This is the default query behavior for local indexes created in parallel.

In general, parallel queries are good for DSS or analytical systems with large data collection, multiple CPUs, and low number of concurrent users.

However, for heavily loaded systems with high number of concurrent users, parallel query can result in degrading your overall query throughput. In addition, typical top N text queries with order by partition key column, such as

```
select * from (
        select story_id from stories_tab where contains(...)>0 order by
publication_date desc)
    where rownum <= 10;
```

will generally perform *worse* with a parallel query.

You can disable parallel querying after a parallel index operation with ALTER INDEX command as follows

```
Alter index <text index name> NOPARALLEL;
Alter index <text index name> PARALLEL 1;
```

You can also enable or increase the parallel degree by doing

```
Alter index <text index name> paralllel < parallel degree >;
```

Tuning Queries with Blocking Operations

Issuing a query with more than one predicate can cause a blocking operation in the execution plan. For example, consider the following mixed query:

```
select docid from mytab where contains(text, 'oracle', 1) > 0
  AND colA > 5
  AND colB > 1
  AND colC > 3;
```

Assume that all predicates are unselective and colA, colB, and colC have bitmap indexes. The Oracle Database cost-based optimizer chooses the following execution plan:

```
TABLE ACCESS BY ROWIDS
  BITMAP CONVERSION TO ROWIDS
    BITMAP AND
      BITMAP INDEX COLA_BMX
      BITMAP INDEX COLB_BMX
      BITMAP INDEX COLC_BMX
      BITMAP CONVERSION FROM ROWIDS
        SORT ORDER BY
          DOMAIN INDEX MYINDEX
```

Since the BITMAP AND is a blocking operation, Oracle Text must temporarily save the rowid and score pairs returned from the Oracle Text domain index before executing the BITMAP AND operation.

Oracle Text attempts to save these rowid and score pairs in memory. However, when the size of the result set containing these rowid and score pairs exceeds the

SORT_AREA_SIZE initialization parameter, Oracle Text spills these results to temporary segments on disk.

Since saving results to disk causes extra overhead, you can improve performance by increasing the SORT_AREA_SIZE parameter using ALTER SESSION as follows:

```
alter session set SORT_AREA_SIZE = <new memory size in bytes>;
```

For example, to set the buffer to approximately 8 megabytes, you can issue:

```
alter session set SORT_AREA_SIZE = 8300000;
```

> **See Also:** *Oracle Database Performance Tuning Guide* and *Oracle Database Reference* for more information on SORT_AREA_SIZE.

Frequently Asked Questions a About Query Performance

This section answers some of the frequently asked questions about query performance.

What is *Query Performance*?

Answer: There are generally two measures of query performance:

- response time, the time to get an answer to an individual query, and
- throughput, the number of queries that can be run in any time period; for example, queries per second).

These two are related, but are not the same. In a heavily loaded system, you normally want maximum throughput, whereas in a relatively lightly loaded system, you probably want minimum response time. Also, some applications require a query to deliver all its hits to the user, whereas others might only require the first 20 hits from an ordered set. It is important to distinguish between these two scenarios.

What is the fastest type of text query?

Answer: The fastest type of query will meet the following conditions:

- Single CONTAINS clause
- No other conditions in the WHERE clause
- No ORDER BY clause at all
- Only the first page of results is returned (for example, the first 10 or 20 hits).

Should I collect statistics on my tables?

Answer: Yes. Collecting statistics on your tables enables Oracle Text to do cost-based analysis. This helps Oracle Text choose the most efficient execution plan for your queries.

See Also: "Optimizing Queries with Statistics" in this chapter.

How does the size of my data affect queries?

Answer: The speed at which the text index can deliver ROWIDs is not affected by the actual size of the data. Text query speed will be related to the number of rows that must be fetched from the index table, number of hits requested, number of hits produced by the query, and the presence or absence of sorting.

How does the format of my data affect queries?

Answer: The format of the documents (plain ascii text, HTML or Microsoft Word) should make no difference to query speed. The documents are filtered to plain text at indexing time, not query time.

The cleanliness of the data will make a difference. Spell-checked and sub-edited text for publication tends to have a much smaller total vocabulary (and therefore size of the index table) than informal text such as emails, which will contain many spelling errors and abbreviations. For a given index memory setting, the extra text takes up more memory, which can lead to more fragmented rows than in the cleaner text, which can adversely affect query response time.

What is a *functional* versus an *indexed* lookup?

Answer: There are two ways the kernel can query the text index. In the first and most common case, the kernel asks the text index for all the rowids that satisfy a particular text search. These rowids are returned in batches. In the second, the kernel passes individual rowids to the text index, and asks whether that particular rowid satisfies a certain text criterion.

The second is known as a functional lookup, and is most commonly done where there is a very selective structured clause, so that only a few rowids must be checked against the text index. An example of a search where a functional lookup may be used:

```
SELECT ID, SCORE(1), TEXT FROM MYTABLE
WHERE START_DATE = '21 Oct 1992'         <- highly selective
AND CONTAINS (TEXT, 'commonword') > 0    <- unselective
```

Functional invocation is also used for text query ordered by structured column (for example date, price) and text query is unselective.

What tables are involved in queries?

Answer: All queries look at the index token table. Its name has the form DR$indexname$I. This contains the list of tokens (column TOKEN_TEXT) and the information about the row and word positions where the token occurs (column TOKEN_INFO).

The row information is stored as internal DOCID values. These must be translated into external ROWID values. The table used for this depends on the type of lookup: For functional lookups, the $K table, DR$indexname$K, is used. This is a simple Index Organized Table (IOT) which contains a row for each DOCID / ROWID pair.

For indexed lookups, the $R table, DR$indexname$R, is used. This holds the complete list of ROWIDs in a BLOB column.

Hence we can easily find out whether a functional or indexed lookup is being used by examining a SQL trace, and looking for the $K or $R tables.

Note: These internal index tables are subject to change from release to release. Oracle recommends that you do not directly access these tables in your application.

Does sorting the results slow a text-only query?

Answer: Yes, it certainly does.

If there is no sorting, then Oracle Text can return results as it finds them, which is quicker in the common case where the application needs to display only a page of results at a time.

How do I make a ORDER BY score query faster?

Answer: Sorting by relevance (SCORE(n)) can be extremely quick if the FIRST_ROWS(n) hint is used. In this case, Oracle Text performs a high speed internal sort when fetching from the text index tables.

An example of such a query:

```
SELECT /*+ FIRST_ROWS(10) */ ID, SCORE(1), TEXT FROM MYTABLE
```

```
WHERE CONTAINS (TEXT, 'searchterm', 1) > 0
ORDER BY SCORE(1) DESC;
```

Note that for this to work efficiently, there must be no other criteria in the WHERE clause other than a single CONTAINS.

Which Memory Settings Affect Querying?

Answer: For querying, you want to strive for a large system global area (SGA). You can set these parameters related to SGA in your Oracle Database initialization file. You can also set these parameters dynamically.

The SORT_AREA_SIZE parameter controls the memory available for sorting for ORDER BY queries. You should increase the size of this parameter if you frequently order by structured columns.

See Also:

Oracle Database Administrator's Guide for more information on setting SGA related parameters.

Oracle Database Performance Tuning Guide for more information on memory allocation and setting the SORT_AREA_SIZE parameter.

Does out of line LOB storage of wide base table columns improve performance?

Answer: Yes. Typically, a SELECT statement selects more than one column from your base table. Since Oracle Text fetches columns to memory, it is more efficient to store wide base table columns such as LOBs out of line, especially when these columns are rarely updated but frequently selected.

When LOBs are stored out of line, only the LOB locators need to be fetched to memory during querying. Out of line storage reduces the effective size of the base table making it easier for Oracle Text to cache the entire table to memory. This reduces the cost of selecting columns from the base table, and hence speeds up text queries.

In addition, having smaller base tables cached in memory enables more index table data to be cached during querying, which improves performance.

How can I make a CONTAINS query on more than one column faster?

Answer: The fastest type of query is one where there is only a single CONTAINS clause, and no other conditions in the WHERE clause.

Consider the following multiple CONTAINS query:

```
SELECT title, isbn FROM booklist
  WHERE CONTAINS (title, 'horse') > 0
    AND CONTAINS (abstract, 'racing') > 0
```

We can obtain the same result with section searching and the WITHIN operator as follows:

```
SELECT title, isbn FROM booklist
  WHERE CONTAINS (alltext,
    'horse WITHIN title AND racing WITHIN abstract')>0
```

This will be a much faster query. In order to use a query like this, we must copy all the data into a single text column for indexing, with section tags around each column's data. This can be done with PL / SQL procedures before indexing, or by making use of the USER_DATASTORE datastore during indexing to synthesize structured columns with the text column into one document.

Is it OK to have many expansions in a query?

Answer: Each distinct word used in a query will require at least one row to be fetched from the index table. It is therefore best to keep the number of expansions down as much as possible.

You should not use expansions such as wild cards, thesaurus, stemming and fuzzy matching unless they are necessary to the task. In general, a few expansions (say up to 20) is OK, but you should try to avoid more than 100 or so expansions in a query. The query feedback mechanism can be used to determine the number of expansions for any particular query expression.

In addition for wildcard and stem queries, you can remove the cost of term expansion from query time to index time by creating prefix, substring or stem indexes. Query performance increases at the cost of longer indexing time and added disk space.

Prefix and substring indexes can improve wildcard performance. You enable prefix and substring indexing with the BASIC_WORDLIST preference. The following example sets the wordlist preference for prefix and substring indexing. For prefix indexing, it specifies that Oracle Text create token prefixes between 3 and 4 characters long:

```
begin
ctx_ddl.create_preference('mywordlist', 'BASIC_WORDLIST');
ctx_ddl.set_attribute('mywordlist','PREFIX_INDEX','TRUE');
```

```
ctx_ddl.set_attribute('mywordlist','PREFIX_MIN_LENGTH', '3');
ctx_ddl.set_attribute('mywordlist','PREFIX_MAX_LENGTH', '4');
ctx_ddl.set_attribute('mywordlist','SUBSTRING_INDEX', 'YES');
end
```

You enable stem indexing with the BASIC_LEXER preference:

```
begin
ctx_ddl.create_preference('mylex', 'BASIC_LEXER');
ctx_ddl.set_attribute ( 'mylex', 'index_stems', 'ENGLISH');
end;
```

How can local partition indexes help?

Answer: You can create local partitioned CONTEXT indexes on partitioned tables. This means that on a partitioned table, each partition has its own set of index tables. Effectively, there are multiple indexes, but the results from each are combined as necessary to produce the final result set.

The index is created using the LOCAL keyword:

```
CREATE INDEX index_name ON table_name (column_name)
INDEXTYPE IS ctxsys.context
PARAMETERS ('...')
LOCAL
```

With partitioned tables and local indexes, you can improve performance of the following types of CONTAINS queries:

- Range Search on Partition Key Column

  This is a query that restricts the search to a particular range of values on a column that is also the partition key.

- ORDER BY Partition Key Column

  This is a query that requires only the first N hits and the ORDER BY clause names the partition key

> **See Also:** "Improved Response Time using Local Partitioned CONTEXT Index" in this chapter.

Should I query in parallel?

Answer: Depends. Even though parallel querying is the default behavior for indexes created in parallel, it usually results in degrading overall query throughput on heavily loaded systems.

In general, parallel queries are good for DSS or analytical systems with large data collections, multiple CPUs, and low number of concurrent users.

> **See Also:** "Parallel Queries" in this chapter.

Should I index themes?

Answer: Indexing theme information with a CONTEXT index takes longer and also increases the size of your index. However, theme indexes enable ABOUT queries to be more precise by using the knowledge base, if available. If your application uses ABOUT queries heavily, it might be worthwhile to create a theme component to the index, despite the extra indexing time and extra storage space required.

> **See Also:** "ABOUT Queries and Themes" in Chapter 4, "Querying".

When should I use a CTXCAT index?

Answer: CTXCAT indexes work best when text is in small chunks, maybe a few lines maximum, and searches need to restrict or sort the result set according to certain structured criteria, usually numbers or dates.

For example, consider an on-line auction site. Each item for sale has a short description, a current bid price, and dates for the start and end of the auction. A user might want to see all the records with *antique cabinet* in the description, with a current bid price less than $500. Since he's particularly interested in newly posted items, he wants the results sorted by auction start time.

Such a search is not always efficient with a CONTAINS structured query on a CONTEXT index, where the response time can vary significantly depending on the structured and CONTAINS clauses. This is because the intersection of structured and CONTAINS clauses or the ordering of text query is computed during query time.

By including structured information such as price and date within the CTXCAT index, query response time is always in an optimal range regardless of search criteria. This is because the interaction between text and structured query is pre-computed during indexing. Consequently query response time is optimum.

When is a CTXCAT index NOT suitable?

Answer: There are differences in the time and space needed to create the index. CTXCAT indexes take a bit longer to create and use considerably more disk space than CONTEXT indexes. If you are tight on disk space, you should consider carefully whether CTXCAT indexes are appropriate for you.

With respect to query operators, you can now use the richer CONTEXT grammar in CATSEARCH queries with query templates. The older restriction of a single CATSEARCH query grammar no longer holds.

What optimizer hints are available, and what do they do?

Answer: The optimizer hint `INDEX(table column)` can be used in the usual way to drive the query with a text or b-tree index.

You can also use the `NO_INDEX(table column)` hint to disable a specific index.

Additionally, the `FIRST_ROWS(n)` hint has a special meaning for text queries and should be used when you need the first n hits to a query. Use of the `FIRST_ROWS` hint in conjunction with `ORDER BY SCORE(n) DESC` tells Oracle Text to accept a sorted set from the text index, and not to do a further sort.

> **See Also:** "Optimizing Queries for Response Time" in this chapter.

Frequently Asked Questions About Indexing Performance

This section answers some of the frequently asked questions about indexing performance.

How long should indexing take?

Answer: Indexing text is a resource-intensive process. Obviously, the speed of indexing will depend on the power of the hardware involved.

As a benchmark, with an average document size of 5K, Oracle Text can index approximately 200 documents per second with the following hardware and parallel configuration:

- 4x400Mhz Sun Sparc CPUs
- 4 gig of RAM
- EMC symmetrix (24 disks striped)

- Parallel degree of 5 with 5 partitions
- Index memory of 600MB per index process
- XML news documents that averaged 5K in size
- USER_DATASTORE

Other factors such as your document format, location of your data, and the calls to user-defined datastores, filters, and lexers can have an impact on your indexing speed.

Which index memory settings should I use?

Answer: You can set your index memory with the system parameters DEFAULT_INDEX_MEMORY and MAX_INDEX_MEMORY. You can also set your index memory at run time with the CREATE INDEX `memory` parameter in the parameter string.

You should aim to set the DEFAULT_INDEX_MEMORY value as high as possible, without causing paging.

You can also improve Indexing performance by increasing the SORT_AREA_SIZE system parameter.

Experience has shown that using a large index memory setting, even into hundreds of megabytes, will improve the speed of indexing and reduce the fragmentation of the final indexes. However, if set too high, then the memory paging that occurs will cripple indexing speed.

With parallel indexing, each stream requires its own index memory. When dealing with very large tables, you can tune your database system global area (SGA) differently for indexing and retrieval. For querying, you are hoping to get as much information cached in the system global area's (SGA) block buffer cache as possible. So you should be allocating a large amount of memory to the block buffer cache. But this will not make any difference to indexing, so you would be better off reducing the size of the SGA to make more room for a large index memory settings during indexing.

You set the size of SGA in your Oracle Database initialization file.

See Also:

Oracle Text Reference to learn more about Oracle Text system parameters.

Oracle Database Administrator's Guide for more information on setting SGA related parameters.

Oracle Database Performance Tuning Guide for more information on memory allocation and setting the SORT_AREA_SIZE parameter.

How much disk overhead will indexing require?

Answer: The overhead, the amount of space needed for the index tables, varies between about 50% of the original text volume and 200%. Generally, the larger the total amount of text, the smaller the overhead, but many small records will use more overhead than fewer large records. Also, clean data (such as published text) will require less overhead than dirty data such as emails or discussion notes, since the dirty data is likely to include many unique words from mis-spellings and abbreviations.

A text-only index is smaller than a combined text and theme index. A prefix and substring index makes the index significantly larger.

How does the format of my data affect indexing?

Answer: You can expect much lower storage overhead for formatted documents such as Microsoft Word files since such documents tend to be very large compared to the actual text held in them. So 1GB of Word documents might only require 50MB of index space, whereas 1GB of plain text might require 500MB, since there is ten times as much plain text in the latter set.

Indexing time is less clear-cut. Although the reduction in the amount of text to be indexed will have an obvious effect, you must balance this out against the cost of filtering the documents with the INSO filter or other user-defined filters.

Can parallel indexing improve performance?

Answer: Parallel indexing can improve index performance when you have a large amount of data, and have multiple CPUs.

You use the PARALLEL keyword when creating the index:

```
CREATE INDEX index_name ON table_name (column_name)
INDEXTYPE IS ctxsys.context PARAMETERS ('...') PARALLEL 3;
```

This will create the index with up to three separate indexing processes depending on your resources.

Parallel indexing can also be used to create local partitioned indexes on partitioned tables. However, indexing performance only improves when you have multiple CPUs.

Note: Using PARALLEL to create a local partitioned index enables parallel queries. (Creating a non-partitioned index in parallel does not turn on parallel query processing.)

Parallel querying degrades query throughput especially on heavily loaded systems. Because of this, Oracle recommends that you disable parallel querying after parallel indexing. To do so, use ALTER INDEX NOPARALLEL.

How can I improve index performance for creating local partitioned index?

Answer: When you have multiple CPUs, you can improve indexing performance by creating a local index in parallel. There are two ways to index in parallel:

You can create a local partitioned index in parallel in two ways:

- Use the `PARALLEL` clause with the `LOCAL` clause in `CREATE INDEX`. In this case, the maximum parallel degree is limited to the number of partitions you have.
- Create an unusable index first, then run the `DBMS_PCLXUTIL.BUILD_PART_INDEX` utility. This method can result in a higher degree of parallelism, especially if you have more CPUs than partitions.

The following is an example for the second method. In this example, the base table has three partitions. We create a local partitioned unusable index first, the run the `DBMS_PCLUTIL.BUILD_PART_INDEX`, which builds the 3 partitions in parallel (inter-partition parallelism). Also inside each partition, index creation is done in parallel (intra-partition parallelism) with a parallel degree of 2.

```
create index tdrbip02bx on tdrbip02b(text)
indextype is ctxsys.context local (partition tdrbip02bx1,
                                   partition tdrbip02bx2,
                                   partition tdrbip02bx3)
unusable;

exec dbms_pclxutil.build_part_index(3,2,'TDRBIP02B','TDRBIP02BX',TRUE);
```

How can I tell how much indexing has completed?

Answer: You can use the `CTX_OUTPUT.START_LOG` procedure to log output from the indexing process. Filename will normally be written to `$ORACLE_HOME/ctx/log`, but you can change the directory using the `LOG_DIRECTORY` parameter in `CTX_ADM.SET_PARAMETER`.

> **See Also:** *Oracle Text Reference* to learn more about using this procedure.

Frequently Asked Questions About Updating the Index

This section answers some of the frequently asked questions about updating your index and related performance issues.

How often should I index new or updated records?

Answer: The less often you run reindexing with `CTX_DLL.SYNC_INDEX`, the less fragmented your indexes will be, and the less you will need to optimize them.

However, this means that your data will become progressively more out of date, which may be unacceptable for your users.

Many systems are OK with overnight indexing. This means data that is less than a day old is not searchable. Other systems use hourly, ten minute, or five minute updates.

> **See Also:** *Oracle Text Reference* to learn more about using `CTX_DDL.SYNC_INDEX`.
>
> "Managing DML Operations for a CONTEXT Index" in Chapter 3, "Indexing"

How can I tell when my indexes are getting fragmented?

Answer: The best way is to time some queries, run index optimization, then time the same queries (restarting the database to clear the SGA each time, of course). If the queries speed up significantly, then optimization was worthwhile. If they don't, you can wait longer next time.

You can also use CTX_REPORT.INDEX_STATS to analyze index fragmentation.

See Also: *Oracle Text Reference* to learn more about using the CTX_REPORT package.

"Index Optimization" in Chapter 3, "Indexing".

Does memory allocation affect index synchronization?

Answer: Yes, the same way as for normal indexing. But of course, there are often far fewer records to be indexed during a synchronize operation, so it is not usually necessary to provide hundreds of megabytes of indexing memory.

8

Document Section Searching

This chapter describes how to use document sections in an Oracle Text query application.

The following topics are discussed in this chapter:

About Document Section Searching

Section searching enables you to narrow text queries down to blocks of text within documents. Section searching is useful when your documents have internal structure, such as HTML and XML documents.

You can also search for text at the sentence and paragraph level.

Enabling Section Searching

The steps for enabling section searching for your document collection are:

1. Create a section group
2. Define your sections
3. Index your documents
4. Section search with `WITHIN`, `INPATH`, or `HASPATH` operators

Create a Section Group

Section searching is enabled by defining section groups. You use one of the system-defined section groups to create an instance of a section group. Choose a section group appropriate for your document collection.

You use section groups to specify the type of document set you have and implicitly indicate the tag structure. For instance, to index HTML tagged documents, you use the HTML_SECTION_GROUP. Likewise, to index XML tagged documents, you can use the XML_SECTION_GROUP.

The following table list the different types of section groups you can use:

Section Group Preference	Description
NULL_SECTION_GROUP	This is the default. Use this group type when you define no sections or when you define *only* SENTENCE or PARAGRAPH sections.
BASIC_SECTION_GROUP	Use this group type for defining sections where the start and end tags are of the form <A> and </A>. Note: This group type dopes not support input such as unbalanced parentheses, comments tags, and attributes. Use HTML_SECTION_GROUP for this type of input.
HTML_SECTION_GROUP	Use this group type for indexing HTML documents and for defining sections in HTML documents.
XML_SECTION_GROUP	Use this group type for indexing XML documents and for defining sections in XML documents.

Section Group Preference	Description
AUTO_SECTION_GROUP	Use this group type to automatically create a zone section for each start-tag/end-tag pair in an XML document. The section names derived from XML tags are case-sensitive as in XML. Attribute sections are created automatically for XML tags that have attributes. Attribute sections are named in the form *tag@attribute*. Stop sections, empty tags, processing instructions, and comments are not indexed. The following limitations apply to automatic section groups: ■ You cannot add zone, field or special sections to an automatic section group. ■ Automatic sectioning does not index XML document types (root elements.) However, you can define stop-sections with document type. ■ The length of the indexed tags including prefix and namespace cannot exceed 64 characters. Tags longer than this are not indexed.
PATH_SECTION_GROUP	Use this group type to index XML documents. Behaves like the AUTO_SECTION_GROUP. The difference is that with this section group you can do path searching with the INPATH and HASPATH operators. Queries are also case-sensitive for tag and attribute names.
NEWS_SECTION_GROUP	Use this group for defining sections in newsgroup formatted documents according to RFC 1036.

You use the CTX_DDL package to create section groups and define sections as part of section groups. For example, to index HTML documents, create a section group with HTML_SECTION_GROUP:

```
begin
ctx_ddl.create_section_group('htmgroup', 'HTML_SECTION_GROUP');
end;
```

Define Your Sections

You define sections as part of the section group. The following example defines an zone section called heading for all text within the HTML < H1> tag:

```
begin
ctx_ddl.create_section_group('htmgroup', 'HTML_SECTION_GROUP');
ctx_ddl.add_zone_section('htmgroup', 'heading', 'H1');
end;
```

Note: If you are using the AUTO_SECTION_GROUP or PATH_SECTION_GROUP to index an XML document collection, you need not explicitly define sections since the system does this for you during indexing.

See Also: "Section Types" in this chapter for more information about sections.

"XML Section Searching" in this chapter for more information about section searching with XML.

Index your Documents

When you index your documents, you specify your section group in the parameter clause of CREATE INDEX.

```
create index myindex on docs(htmlfile) indextype is ctxsys.context
parameters('filter ctxsys.null_filter section group htmgroup');
```

Section Searching with WITHIN Operator

When your documents are indexed, you can query within sections using the WITHIN operator. For example, to find all the documents that contain the word *Oracle* within their headings, issue the following query:

```
'Oracle WITHIN heading'
```

See Also: *Oracle Text Reference* to learn more about using the WITHIN operator.

Path Searching with INPATH and HASPATH Operators

When you use the PATH_SECTION_GROUP, the system automatically creates XML sections for you. In addition to using the WITHIN operator to issue queries, you can issue path queries with the INPATH and HASPATH operators.

See Also: "XML Section Searching" to learn more about using these operators.

Oracle Text Reference to learn more about using the `INPATH` operator.

Section Types

All sections types are blocks of text in a document. However, sections can differ in the way they are delimited and the way they are recorded in the index. Sections can be one of the following:

- Zone Section
- Field Section
- Stop Section
- MDATA Section
- Attribute Section (for XML documents)
- Special Sections (sentence or paragraphs)

Table 8–1 shows which section types may be used with each kind of section group.

Table 8–1 Section Types and Section Groups

Section Group	ZONE	FIELD	SPECIAL	STOP	ATTRIBUTE	MDATA
NULL	NO	NO	YES	NO	NO	NO
BASIC	YES	YES	YES	NO	NO	YES
HTML	YES	YES	YES	NO	NO	YES
XML	YES	YES	YES	NO	YES	YES
NEWS	YES	YES	YES	NO	NO	YES
AUTO	NO	NO	NO	YES	NO	NO
PATH	NO	NO	NO	NO	NO	NO

Zone Section

A zone section is a body of text delimited by start and end tags in a document. The positions of the start and end tags are recorded in the index so that any words in

between the tags are considered to be within the section. Any instance of a zone section must have a start and an end tag.

For example, the text between the <TITLE> and </TITLE> tags can be defined as a zone section as follows:

```
<TITLE>Tale of Two Cities</TITLE>
It was the best of times...
```

Zone sections can nest, overlap, and repeat within a document.

When querying zone sections, you use the WITHIN operator to search for a term across all sections. Oracle Text returns those documents that contain the term within the defined section.

Zone sections are well suited for defining sections in HTML and XML documents. To define a zone section, use CTX_DDL.ADD_ZONE_SECTION.

For example, assume you define the section booktitle as follows:

```
begin
ctx_ddl.create_section_group('htmgroup', 'HTML_SECTION_GROUP');
ctx_ddl.add_zone_section('htmgroup', 'booktitle', 'TITLE');
end;
```

After you index, you can search for all the documents that contain the term *Cities* within the section *booktitle* as follows:

```
'Cities WITHIN booktitle'
```

With multiple query terms such as *(dog and cat) WITHIN booktitle*, Oracle Text returns those documents that contain *cat* and *dog* within the same instance of a booktitle section.

Repeated Zone Sections Zone sections can repeat. Each occurrence is treated as a separate section. For example, if <H1> denotes a heading section, they can repeat in the same documents as follows:

```
<H1> The Brown Fox </H1>
<H1> The Gray Wolf </H1>
```

Assuming that these zone sections are named Heading, the query *Brown WITHIN Heading* returns this document. However, a query of *(Brown and Gray) WITHIN Heading* does not.

Overlapping Zone Sections Zone sections can overlap each other. For example, if <B> and <I> denote two different zone sections, they can overlap in a document as follows:

```
plain <B> bold <I> bold and italic </B> only italic </I>  plain
```

Nested Zone Sections Zone sections can nest, including themselves as follows:

```
<TD> <TABLE><TD>nested cell</TD></TABLE></TD>
```

Using the WITHIN operator, you can write queries to search for text in sections within sections. For example, assume the BOOK1, BOOK2, and AUTHOR zone sections occur as follows in documents doc1 and doc2:

doc1:

```
<book1> <author>Scott Tiger</author> This is a cool book to read.</book1>
```

doc2:

```
<book2> <author>Scott Tiger</author> This is a great book to read.</book2>
```

Consider the nested query:

```
'(Scott within author) within book1'
```

This query returns only doc1.

Field Section

A field section is similar to a zone section in that it is a region of text delimited by start and end tags. A field section is different from a zone section in that the region is indexed separate from the rest of the document.

Since field sections are indexed differently, you can also get better query performance over zone sections for when you have a large number of documents indexed.

Field sections are more suited to when you have a single occurrence of a section in a a document such as a field in a news header. Field sections can also be made visible to the rest of the document.

Unlike zone sections, field sections have the following restrictions:

- field sections cannot overlap
- field sections cannot repeat

- field sections cannot nest

Visible and Invisible Field Sections By default, field sections are indexed as a sub-document separate from the rest of the document. As such, field sections are invisible to the surrounding text and can only be queried by explicitly naming the section in the WITHIN clause.

You can make field sections visible if you want the text within the field section to be indexed as part of the enclosing document. Text within a visible field section can be queried with or without the WITHIN operator.

The following example shows the difference between using invisible and visible field sections.

The following code defines a section group basicgroup of the BASIC_SECTION_GROUP type. It then creates a field section in basicgroup called Author for the <A> tag. It also sets the visible flag to FALSE to create an invisible section:

```
begin
ctx_ddl.create_section_group('basicgroup', 'BASIC_SECTION_GROUP');
ctx_ddl.add_field_section('basicgroup', 'Author', 'A', FALSE);
end;
```

Because the Author field section is not visible, to find text within the Author section, you must use the WITHIN operator as follows:

```
'(Martin Luther King) WITHIN Author'
```

A query of *Martin Luther King* without the WITHIN operator does not return instances of this term in field sections. If you want to query text within field sections without specifying WITHIN, you must set the visible flag to TRUE when you create the section as follows:

```
begin
ctx_ddl.add_field_section('basicgroup', 'Author', 'A', TRUE);
end;
```

Nested Field Sections Field sections cannot be nested. For example, if you define a field section to start with <TITLE> and define another field section to start with <FOO>, the two sections *cannot* be nested as follows:

```
<TITLE> dog <FOO> cat </FOO> </TITLE>
```

To work with nested sections, define them as zone sections.

Repeated Field Sections Repeated field sections are allowed, but `WITHIN` queries treat them as a single section. The following is an example of repeated field section in a document:

```
<TITLE> cat </TITLE>
<TITLE> dog </TITLE>
```

The query *dog and cat within title* returns the document, even though these words occur in different sections.

To have `WITHIN` queries distinguish repeated sections, define them as zone sections.

Stop Section

A stop section may be added to an automatic section group. Adding a stop section causes the automatic section indexing operation to ignore the specified section in XML documents.

> **Note:** Adding a stop section causes no section information to be created in the index. However, the text within a stop section is always searchable.

Adding a stop section is useful when your documents contain many low information tags. Adding stop sections also improves indexing performance with the automatic section group.

The number of stop sections you can add is unlimited.

Stop sections do not have section names and hence are not recorded in the section views.

MDATA Section

An `MDATA` section is used to reference user-defined metadata for a document. Using `MDATA` sections can speed up mixed queries.

Consider the case in which you want to query both according to text content and document type (magazine or newspaper or novel). You could create an index with a column for text and a column for the document type, and then perform a mixed query of this form—in this case, searching for all novels with the phrase *Adam Thorpe* (author of the novel *Ulverton*):

```
SELECT id FROM documents
   WHERE doctype = 'novel'
      AND CONTAINS(text, 'Adam Thorpe')>0;
```

However, it is usually faster to incorporate the attribute (in this case, the document type) into a field section, rather than use a separate column, and then use a single CONTAINS query:

```
SELECT id FROM documents
  WHERE CONTAINS(text, 'Adam Thorpe AND novel WITHIN doctype')>0;
```

There are two drawbacks to this approach:

- Each time the attribute is updated, the entire text document must be re-indexed, resulting in increased index fragmentation and slower rates of processing DML.
- Field sections tokenize the section value. This has several effects. Special characters in metadata, such as decimal points or currency characters, are not easily searchable; value searching (searching for *Thurston Howell* but not *Thurston Howell, Jr.*) is difficult; multi-word values are queried by phrase, which is slower than single-token searching; and multi-word values do not show up in browse-words, making author browsing or subject browsing impossible.

For these reasons, using MDATA sections instead of field sections may be worthwhile. MDATA sections are indexed like field sections, but metadata values can be added to and removed from documents without the need to re-index the document text. Unlike field sections, MDATA values are not tokenized. Additionally, MDATA section indexing generally takes up less disk space than field section indexing.

Use CTX_DDL.ADD_MDATA_SECTION to add an MDATA section to a section group. This example adds an MDATA section called AUTHOR and gives it the value *Soseki Natsume* (author of the novel *Kokoro*).

```
ctx_ddl.create.section.group('htmgroup', 'HTML_SECTION_GROUP');
ctx_ddl.add_mdata_section('htmgroup', 'author', 'Soseki Natsume');
```

MDATA values can be changed with CTX_DDL.ADD_MDATA and removed with CTX_DDL.REMOVE_MDATA. MDATA sections can have multiple values. Only the owner of the index is allowed to call CTX_DDL.ADD_MDATA and CTX_DDL.REMOVE_MDATA.

Neither CTX_DDL.ADD_MDATA nor CTX_DDL.REMOVE_MDATA are supported for CTXCAT, CTXXPTH and CTXRULE indexes.

MDATA values are not passed through a lexer. Instead, all values undergo a simplified normalization:

- Leading and trailing whitespace on the value is removed.
- The value is truncated to 64 bytes.
- The value is converted to upper case.
- The value is indexed as a single value; if the value consists of multiple words, it is not broken up.
- Case is preserved. If the document is dynamically generated, you can implement case-insensitivity by uppercasing MDATA values and making sure to search only in uppercase.

Once a document has had MDATA metadata added to it, you can query for that metadata using the MDATA CONTAINS query operator:

```
SELECT id FROM documents
   WHERE CONTAINS(text, 'Tokyo and MDATA(author, Soseki Natsume)')>0;
```

This query will only be successful if an AUTHOR tag has the exact value *Soseki Natsume* (after simplified tokenization). *Soseki* or *Natsume Soseki* will not work.

Other things to note about MDATA:

- MDATA values are not highlightable, will not appear in the output of CTX_DOC.TOKENS, and will not show up when FILTER PLAINTEXT is enabled.
- MDATA sections must be unique within section groups. You cannot have an MDATA section named FOO and a zone or field section of the same name in the same section group.
- Like field sections, MDATA sections cannot overlap or nest. An MDATA section is implicitly closed by the first tag encountered. For instance, in this example:

  ```
  <AUTHOR>Dickens <B>Shelley</B> Keats</AUTHOR>
  ```

  The <B> tag closes the AUTHOR MDATA section; as a result, this document has an AUTHOR of 'Dickens', but not of 'Shelley' or 'Keats'.
- To prevent race conditions, each call to ADD_MDATA and REMOVE_MDATA locks out other calls on that rowid for that index for all values and sections. However, since ADD_MDATA and REMOVE_MDATA do not commit, it is possible for an application to deadlock when calling them both. It is the application's responsibility to prevent deadlocking.

See Also: The CONTAINS query operators chapter of the *Oracle Text Reference* for information on the MDATA operator, and the CTX_DDL package chapter of the *Oracle Text Reference* for information on adding and removing MDATA sections

Attribute Section

You can define attribute sections to query on XML attribute text. You can also have the system automatically define and index XML attributes for you.

See Also: "XML Section Searching" in this chapter.

Special Sections

Special sections are not recognized by tags. Currently the only special sections supported are sentence and paragraph. This enables you to search for combination of words within sentences or paragraphs.

The sentence and paragraph boundaries are determined by the lexer. For example, the BASIC_LEXER recognizes sentence and paragraph section boundaries as follows:

Table 8–2 Sentence and Paragraph Section Boundaries for BASIC_LEXER

Special Section	Boundary
SENTENCE	WORD / PUNCT / WHITESPACE
	WORD / PUNCT / NEWLINE
PARAGRAPH	WORD / PUNCT / NEWLINE / WHITESPACE
	WORD / PUNCT / NEWLINE / NEWLINE

If the lexer cannot recognize the boundaries, no sentence or paragraph sections are indexed.

To add a special section, use the CTX_DDL.ADD_SPECIAL_SECTION procedure. For example, the following code enables searching within sentences within HTML documents:

```
begin
ctx_ddl.create_section_group('htmgroup', 'HTML_SECTION_GROUP');
ctx_ddl.add_special_section('htmgroup', 'SENTENCE');
end;
```

You can also add zone sections to the group to enable zone searching in addition to sentence searching. The following example adds the zone section `Headline` to the section group `htmgroup`:

```
begin
ctx_ddl.create_section_group('htmgroup', 'HTML_SECTION_GROUP');
ctx_ddl.add_special_section('htmgroup', 'SENTENCE');
ctx_ddl.add_zone_section('htmgroup', 'Headline', 'H1');
end;
```

HTML Section Searching

HTML has internal structure in the form of tagged text which you can use for section searching. For example, you can define a section called headings for the `<H1>` tag. This enables you to search for terms only within these tags across your document set.

To query, you use the `WITHIN` operator. Oracle Text returns all documents that contain your query term within the headings section. Thus, if you wanted to find all documents that contain the word oracle within headings, you issue the following query:

```
'oracle within headings'
```

Creating HTML Sections

The following code defines a section group called `htmgroup` of type `HTML_SECTION_GROUP`. It then creates a zone section in `htmgroup` called `heading` identified by the <H1> tag:

```
begin
ctx_ddl.create_section_group('htmgroup', 'HTML_SECTION_GROUP');
ctx_ddl.add_zone_section('htmgroup', 'heading', 'H1');
end;
```

You can then index your documents as follows:

```
create index myindex on docs(htmlfile) indextype is ctxsys.context
parameters('filter ctxsys.null_filter section group htmgroup');
```

After indexing with section group `htmgroup`, you can query within the heading section by issuing a query as follows:

```
'Oracle WITHIN heading'
```

Searching HTML Meta Tags

With HTML documents you can also create sections for NAME/CONTENT pairs in <META> tags. When you do so you can limit your searches to text within CONTENT.

Example: Creating Sections for <META>Tags

Consider an HTML document that has a META tag as follows:

```
<META NAME="author" CONTENT="ken">
```

To create a zone section that indexes all CONTENT attributes for the META tag whose NAME value is author:

```
begin
ctx_ddl.create_section_group('htmgroup', 'HTML_SECTION_GROUP');
ctx_ddl.add_zone_section('htmgroup', 'author', 'meta@author');
end
```

After indexing with section group htmgroup, you can query the document as follows:

```
'ken WITHIN author'
```

XML Section Searching

Like HTML documents, XML documents have tagged text which you can use to define blocks of text for section searching. The contents of a section can be searched on with the WITHIN or INPATH operators.

For XML searching, you can do the following:

- automatic sectioning
- attribute searching
- document type sensitive sections
- path section searching

Automatic Sectioning

You can set up your indexing operation to automatically create sections from XML documents using the section group AUTO_SECTION_GROUP. The system creates zone sections for XML tags. Attribute sections are created for the tags that have attributes and these sections named in the form tag@attribute.

For example, the following command creates the index *myindex* on a column containing the XML files using the AUTO_SECTION_GROUP:

```
CREATE INDEX myindex ON xmldocs(xmlfile) INDEXTYPE IS ctxsys.context PARAMETERS
('datastore ctxsys.default_datastore filter ctxsys.null_filter section group
ctxsys.auto_section_group');
```

Attribute Searching

You can search XML attribute text in one of two ways:

- Create attribute sections with CTX_DDL.ADD_ATTR_SECTION and then index with the XML_SECTION_GROUP. If you use AUTO_SECTION_GROUP when you index, attribute sections are created automatically. You can query attribute sections with the WITHIN operator.
- Index with the PATH_SECTION_GROUP and query attribute text with the INPATH operator.

Creating Attribute Sections

Consider an XML file that defines the BOOK tag with a TITLE attribute as follows:

```
<BOOK TITLE="Tale of Two Cities">
  It was the best of times.
</BOOK>
```

To define the title attribute as an attribute section, create an XML_SECTION_GROUP and define the attribute section as follows:

```
begin
ctx_ddl.create_section_group('myxmlgroup', 'XML_SECTION_GROUP');
ctx_ddl.add_attr_section('myxmlgroup', 'booktitle', 'book@title');
end;
```

To index:

```
CREATE INDEX myindex ON xmldocs(xmlfile) INDEXTYPE IS ctxsys.context PARAMETERS
('datastore ctxsys.default_datastore filter ctxsys.null_filter section group
myxmlgroup');
```

You can query the XML attribute section *booktitle* as follows:

```
'Cities within booktitle'
```

Searching Attributes with the INPATH Operator

You can search attribute text with the INPATH operator. To do so, you must index your XML document set with the PATH_SECTION_GROUP.

See Also: "Path Section Searching" in this chapter.

Creating Document Type Sensitive Sections

You have an XML document set that contains the <book> tag declared for different document types. You want to create a distinct book section for each document type.

Assume that mydocname1 is declared as an XML document type (root element) as follows:

```
<!DOCTYPE mydocname1 ... [...
```

Within mydocname1, the element <book> is declared. For this tag, you can create a section named mybooksec1 that is sensitive to the tag's document type as follows:

```
begin
ctx_ddl.create_section_group('myxmlgroup', 'XML_SECTION_GROUP');
ctx_ddl.add_zone_section('myxmlgroup', 'mybooksec1', 'mydocname1(book)');
end;
```

Assume that mydocname2 is declared as another XML document type (root element) as follows:

```
<!DOCTYPE mydocname2 ... [...
```

Within mydocname2, the element <book> is declared. For this tag, you can create a section named mybooksec2 that is sensitive to the tag's document type as follows:

```
begin
ctx_ddl.create_section_group('myxmlgroup', 'XML_SECTION_GROUP');
ctx_ddl.add_zone_section('myxmlgroup', 'mybooksec2', 'mydocname2(book)');
end;
```

To query within the section mybooksec1, use WITHIN as follows:

```
'oracle within mybooksec1'
```

Path Section Searching

XML documents can have parent-child tag structures such as the following:

```
<A> <B> <C> dog </C> </B> </A>
```

In this example, tag C is a child of tag B which is a child of tag A.

With Oracle Text, you can do path searching with PATH_SECTION_GROUP. This section group enables you to specify direct parentage in queries, such as to find all documents that contain the term *dog* in element C which is a child of element B and so on.

With PATH_SECTION_GROUP, you can also perform attribute value searching and attribute equality testing.

The new operators associated with this feature are

- INPATH
- HASPATH

Creating Index with PATH_SECTION_GROUP

To enable path section searching, index your XML document set with PATH_SECTION_GROUP.

Create the preference:

```
begin
ctx_ddl.create_section_group('xmlpathgroup', 'PATH_SECTION_GROUP');
end;
```

Create the index:

```
CREATE INDEX myindex ON xmldocs(xmlfile) INDEXTYPE IS ctxsys.context PARAMETERS
('datastore ctxsys.default_datastore filter ctxsys.null_filter section group
xmlpathgroup');
```

When you create the index, you can use the INPATH and HASPATH operators.

Top-Level Tag Searching

To find all documents that contain the term *dog* in the top-level tag <A>:

```
dog INPATH (/A)
```

or

```
dog INPATH(A)
```

Any-Level Tag Searching

To find all documents that contain the term *dog* in the <A> tag at any level:

```
dog INPATH(//A)
```

This query finds the following documents:

```
<A>dog</A>
```

and

```
<C><B><A>dog</A></B></C>
```

Direct Parentage Searching

To find all documents that contain the term *dog* in a B element that is a direct child of a top-level A element:

```
dog INPATH(A/B)
```

This query finds the following XML document:

```
<A><B>My dog is friendly.</B></A>
```

but does not find:

```
<C><B>My dog is friendly.</B></C>
```

Tag Value Testing

You can test the value of tags. For example, the query:

```
dog INPATH(A[B="dog"])
```

Finds the following document:

```
<A><B>dog</B></A>
```

But does not find:

```
<A><B>My dog is friendly.</B></A>
```

Attribute Searching

You can search the content of attributes. For example, the query:

```
dog INPATH(//A/@B)
```

Finds the document

```
<C><A  B="snoop dog"> </A> </C>
```

Attribute Value Testing

You can test the value of attributes. For example, the query

```
California INPATH (//A[@B = "home address"])
```

Finds the document:

```
<A B="home address">San Francisco, California, USA</A>
```

But does not find:

```
<A B="work address">San Francisco, California, USA</A>
```

Path Testing

You can test if a path exists with the HASPATH operator. For example, the query:

```
HASPATH(A/B/C)
```

finds and returns a score of 100 for the document

```
<A><B><C>dog</C></B></A>
```

without the query having to reference *dog* at all.

Section Equality Testing with HASPATH

You can use the HASPATH operator to do section quality tests. For example, consider the following query:

```
dog INPATH A
```

finds

```
<A>dog</A>
```

but it also finds

```
<A>dog park</A>
```

To limit the query to the term *dog* and nothing else, you can use a section equality test with the HASPATH operator. For example,

```
HASPATH(A="dog")
```

finds and returns a score of 100 only for the first document, and not the second.

See Also: *Oracle Text Reference* to learn more about using the INPATH and HASPATH operators.

9

Working With a Thesaurus

This chapter describes how to improve your query application with a thesaurus. The following topics are discussed in this chapter:

- Overview of Thesauri
- Defining Thesaural Terms
- Using a Thesaurus in a Query Application
- About the Supplied Knowledge Base

Overview of Thesauri

Users of your query application looking for information on a given topic might not know which words have been used in documents that refer to that topic.

Oracle Text enables you to create case-sensitive or case-insensitive thesauri which define synonym and hierarchical relationships between words and phrases. You can then retrieve documents that contain relevant text by expanding queries to include similar or related terms as defined in the thesaurus.

You can create a thesaurus and load it into the system.

Note: The Oracle Text thesauri formats and functionality are compliant with both the ISO-2788 and ANSI Z39.19 (1993) standards.

Thesaurus Creation and Maintenance

Thesauri and thesaurus entries can be created, modified, and deleted by all Oracle Text users with the CTXAPP role.

CTX_THES Package

To maintain and browse your thesaurus programatically, you can use the PL/SQL package, CTX_THES. With this package, you can browse terms and hierarchical relationships, add and delete terms, and add and remove thesaurus relations.

Thesaurus Operators

You can also use the thesaurus operators in the CONTAINS clause to expand query terms according to your loaded thesaurus. For example, you can use the SYN operator to expand a term such as *dog* to its synonyms as follows:

'syn(dog)'

ctxload Utility

The ctxload utility can be used for loading thesauri from a plain-text file into the thesaurus tables, as well as dumping thesauri from the tables into output (dump) files.

The thesaurus dump files created by ctxload can be printed out or used as input for other applications. The dump files can also be used to load a thesaurus into the thesaurus tables. This can be useful for using an existing thesaurus as the basis for creating a new thesaurus.

Case-sensitive Thesauri

In a case-sensitive thesaurus, terms (words and phrases) are stored exactly as entered. For example, if a term is entered in mixed-case (using either the CTX_THES package or a thesaurus load file), the thesaurus stores the entry in mixed-case.

Note: To take full advantage of query expansions that result from a case-sensitive thesaurus, your index must also be case-sensitive.

When loading a thesaurus, you can specify that the thesaurus be loaded case-sensitive using the -thescase parameter.

When creating a thesaurus with CTX_THES.CREATE_THESAURUS, you can specify that the thesaurus created be case-sensitive.

In addition, when a case-sensitive thesaurus is specified in a query, the thesaurus lookup uses the query terms exactly as entered in the query. Therefore, queries that use case-sensitive thesauri allow for a higher level of precision in the query

expansion, which helps lookup when and only when you have a case-sensitive index.

For example, a case-sensitive thesaurus is created with different entries for the distinct meanings of the terms *Turkey* (the country) and *turkey* (the type of bird). Using the thesaurus, a query for *Turkey* expands to include only the entries associated with *Turkey*.

Case-insensitive Thesauri

In a case-insensitive thesaurus, terms are stored in all-uppercase, regardless of the case in which they were entered.

The ctxload program loads a thesaurus case-insensitive by default.

When creating a thesaurus with CTX_THES.CREATE_THESAURUS, the thesaurus is created case-insensitive by default.

In addition, when a case-insensitive thesaurus is specified in a query, the query terms are converted to all-uppercase for thesaurus lookup. As a result, Oracle Text is unable to distinguish between terms that have different meanings when they are in mixed-case.

For example, a case-insensitive thesaurus is created with different entries for the two distinct meanings of the term *TURKEY* (the country or the type of bird). Using the thesaurus, a query for either *Turkey* or *turkey* is converted to *TURKEY* for thesaurus lookup and then expanded to include all the entries associated with both meanings.

Default Thesaurus

If you do not specify a thesaurus by name in a query, by default, the thesaurus operators use a thesaurus named *DEFAULT*. However, Oracle Text does not provide a *DEFAULT* thesaurus.

As a result, if you want to use a default thesaurus for the thesaurus operators, you must create a thesaurus named *DEFAULT*. You can create the thesaurus through any of the thesaurus creation methods supported by Oracle Text:

- CTX_THES.CREATE_THESAURUS (PL/SQL)
- ctxload

> **See Also:** *Oracle Text Reference* to learn more about using `ctxload` and the CTX_THES package.

Supplied Thesaurus

Although Oracle Text does not provide a default thesaurus, Oracle Text does supply a thesaurus, in the form of a `ctxload` load file, that can be used to create a general-purpose, English-language thesaurus.

The thesaurus load file can be used to create a default thesaurus for Oracle Text or it can be used as the basis for creating thesauri tailored to a specific subject or range of subjects.

> **See Also:** *Oracle Text Reference* to learn more about using `ctxload` and the `CTX_THES` package.

Supplied Thesaurus Structure and Content

The supplied thesaurus is similar to a traditional thesaurus, such as Roget's Thesaurus, in that it provides a list of synonymous and semantically related terms.

The supplied thesaurus provides additional value by organizing the terms into a hierarchy that defines real-world, practical relationships between narrower terms and their broader terms.

Additionally, cross-references are established between terms in different areas of the hierarchy.

Supplied Thesaurus Location

The exact name and location of the thesaurus load file is operating system dependent; however, the file is generally named `dr0thsus` (with an appropriate extension for text files) and is generally located in the following directory structure:

```
<Oracle_home_directory>
    <interMedia_Text_directory>
       sample
           thes
```

> **See Also:** For more information about the directory structure for Oracle Text, see the Oracle Database installation documentation specific to your operating system.

Defining Thesaural Terms

You can create synonyms, related terms, and hierarchical relationships with a thesaurus. The following sections give examples.

Defining Synonyms

If you have a thesaurus of computer science terms, you might define a synonym for the term *XML* as *extensible markup language*. This enables queries on either of these terms to return the same documents.

```
XML
SYN Extensible Markup Language
```

You can thus use the SYN operator to expand XML into its synonyms:

```
'SYN(XML)'
```

is expanded to:

```
'XML, Extensible Markup Language'
```

Defining Hierarchical Relations

If your document set is made up of news articles, you can use a thesaurus to define a hierarchy of geographical terms. Consider the following hierarchy that describes a geographical hierarchy for the U.S state of California:

```
California
   NT Northern California
       NT San Francisco
       NT San Jose
   NT Central Valley
       NT Fresno
   NT Southern California
       NT Los Angeles
```

You can thus use the NT operator to expand a query on California as follows:

```
'NT(California)'
```

expands to:

```
'California, Northern California, San Francisco, San Jose, Central Valley,
Fresno, Southern California, Los Angeles'
```

The resulting hitlist shows all documents related to the U.S. state of California regions and cities.

Using a Thesaurus in a Query Application

Defining a custom thesaurus enables you to process queries more intelligently. Since users of your application might not know which words represent a topic, you can define synonyms or narrower terms for likely query terms. You can use the thesaurus operators to expand your query into your thesaurus terms.

There are two ways to enhance your query application with a custom thesaurus so that you can process queries more intelligently:

- Load your custom thesaurus and issue queries with thesaurus operators
- Augment the knowledge base with your custom thesaurus (English only) and use the ABOUT operator to expand your query.

Each approach has its advantages and disadvantages.

Loading a Custom Thesaurus and Issuing Thesaural Queries

To build a custom thesaurus, follow these steps:

1. Create your thesaurus. See "Defining Thesaural Terms" in this chapter.
2. Load thesaurus with ctxload. For example, the following example imports a thesaurus named `tech_doc` from an import file named `tech_thesaurus.txt`:

```
ctxload -user jsmith/123abc -thes -name tech_doc -file tech_thesaurus.txt
```

1. Use THES operators to query. For example, you can find all documents that contain XML and its synonyms as defined in tech_doc:

```
'SYN(XML, tech_doc)'
```

Advantage

The advantage of using this method is that you can modify the thesaurus after indexing.

Limitations

This method requires you to use thesaurus expansion operators in your query. Long queries can cause extra overhead in the thesaurus expansion and slow your query down.

Augmenting Knowledge Base with Custom Thesaurus

You can add your custom thesaurus to a branch in the existing knowledge base. The knowledge base is a hierarchical tree of concepts used for theme indexing, ABOUT queries, and deriving themes for document services.

When you augment the existing knowledge base with your new thesaurus, you query with the ABOUT operator which implicitly expands to synonyms and narrower terms. You do not query with the thesaurus operators.

To augment the existing knowledge base with your custom thesaurus, follow these steps:

1. Create your custom thesaurus, linking new terms to existing knowledge base terms. See "Defining Thesaural Terms" and "Linking New Terms to Existing Terms".
2. Load thesaurus with `ctxload`. See "Loading a Thesaurus with ctxload".
3. Compile the loaded thesaurus with `ctxkbtc` compiler. "Compiling a Loaded Thesaurus" later in this section.
4. Index your documents. By default the system creates a theme component to your index.
5. Use ABOUT operator to query. For example, to find all documents that are related to the term politics including any synonyms or narrower terms as defined in the knowledge base, issue the query:

   ```
   'about(politics)'
   ```

Advantage

Compiling your custom thesaurus with the existing knowledge base before indexing enables faster and simpler queries with the ABOUT operator. Document services can also take full advantage of the customized information for creating theme summaries and Gists.

Limitations

Use of the ABOUT operator requires a theme component in the index, which requires slightly more disk space. You must also define the thesaurus before indexing your documents. If you make any change to the thesaurus, you must recompile your thesaurus and re-index your documents.

Linking New Terms to Existing Terms

When adding terms to the knowledge base, Oracle recommends that new terms be linked to one of the categories in the knowledge base for best results in theme proving.

> **See Also:** *Oracle Text Reference* for more information about the supplied English knowledge base.

If new terms are kept completely separate from existing categories, fewer themes from new terms will be proven. The result of this is poor precision and recall with ABOUT queries as well as poor quality of gists and theme highlighting.

You link new terms to existing terms by making an existing term the broader term for the new terms.

Example: Linking New Terms to Existing Terms You purchase a medical thesaurus `medthes` containing a a hierarchy of medical terms. The four top terms in the thesaurus are the following:

- Anesthesia and Analgesia
- Anti-Allergic and Respiratory System Agents
- Anti-Inflammatory Agents, Antirheumatic Agents, and Inflammation Mediators
- Antineoplastic and Immunosuppressive Agents

To link these terms to the existing knowledge base, add the following entries to the medical thesaurus to map the new terms to the existing *health and medicine* branch:

```
health and medicine
 NT Anesthesia and Analgesia
 NT Anti-Allergic and Respiratory System Agents
 NT Anti-Inflamammatory Agents, Antirheumatic Agents, and Inflamation Mediators
 NT Antineoplastic and Immunosuppressive Agents
```

Loading a Thesaurus with ctxload

Assuming the medical thesaurus is in a file called `med.thes`, you load the thesaurus as `medthes` with `ctxload` as follows:

```
ctxload -thes -thescase y -name medthes -file med.thes -user ctxsys/ctxsys
```

Compiling a Loaded Thesaurus

To link the loaded thesaurus `medthes` to the knowledge base, use `ctxkbtc` as follows:

```
ctxkbtc -user ctxsys/ctxsys -name medthes
```

About the Supplied Knowledge Base

Oracle Text supplies a knowledge base for English and French. The supplied knowledge contains the information used to perform theme analysis. Theme analysis includes theme indexing, ABOUT queries, and theme extraction with the CTX_DOC package.

The knowledge base is a hierarchical tree of concepts and categories. It has six main branches:

- science and technology
- business and economics
- government and military
- social environment
- geography
- abstract ideas and concepts

> **See Also:** *Oracle Text Reference* for the breakdown of the category hierarchy.

The supplied knowledge base is like a thesaurus in that it is hierarchical and contains broader term, narrower term, and related term information. As such, you can improve the accuracy of theme analysis by augmenting the knowledge base with your industry-specific thesaurus by linking new terms to existing terms.

> **See Also:** "Augmenting Knowledge Base with Custom Thesaurus" in this chapter.

You can also extend theme functionality to other languages by compiling a language-specific thesaurus into a knowledge base.

> **See Also:** "Adding a Language-Specific Knowledge Base" in this chapter.

Knowledge Base Character Set

Knowledge bases can be in any single-byte character set. Supplied knowledge bases are in WE8ISO8859P1. You can store an extended knowledge base in another character set such as US7ASCII.

Adding a Language-Specific Knowledge Base

You can extend theme functionality to languages other than English or French by loading your own knowledge base for any single-byte whitespace delimited language, including Spanish.

Theme functionality includes theme indexing, ABOUT queries, theme highlighting, and the generation of themes, gists, and theme summaries with CTX_DOC.

You extend theme functionality by adding a user-defined knowledge base. For example, you can create a Spanish knowledge base from a Spanish thesaurus.

To load your language-specific knowledge base, follow these steps:

1. Load your custom thesaurus using `ctxload`.
2. Set NLS_LANG so that the language portion is the target language. The charset portion must be a single-byte character set.
3. Compile the loaded thesaurus using `ctxkbtc`:

```
ctxkbtc -user ctxsys/ctxsys -name my_lang_thes
```

This command compiles your language-specific knowledge base from the loaded thesaurus. To use this knowledge base for theme analysis during indexing and ABOUT queries, specify the NLS_LANG language as the THEME_LANGUAGE attribute value for the BASIC_LEXER preference.

Limitations

The following limitations hold for adding knowledge bases:

- Oracle supplies knowledge bases in English and French only. You must provide your own thesaurus for any other language.
- You can only add knowledge bases for languages with single-byte character sets. You cannot create a knowledge base for languages which can be expressed only in multibyte character sets. If the database is a multibyte universal character set, such as UTF-8, the NLS_LANG parameter must still be set to a compatible single-byte character set when compiling the thesaurus.
- Adding a knowledge base works best for whitespace delimited languages.

- You can have at most one knowledge base for each NLS language.
- Obtaining hierarchical query feedback information such as broader terms, narrower terms and related terms does not work in languages other than English and French. In other languages, the knowledge bases are derived entirely from your thesauri. In such cases, Oracle recommends that you obtain hierarchical information directly from your thesauri.

See Also: *Oracle Text Reference* for more information about theme indexing, ABOUT queries, using the CTX_DOC package, and the supplied English knowledge base.

10

Administration

This chapter describes Oracle Text administration.The following topics are covered:

- Oracle Text Users and Roles
- DML Queue
- The CTX_OUTPUT Package
- The CTX_REPORT Package
- Servers
- Administration Tool

Oracle Text Users and Roles

While any user can create an Oracle Text index and issue a `CONTAINS` query, Oracle Text provides the `CTXSYS` user for administration and the `CTXAPP` role for application developers.

CTXSYS User

The `CTXSYS` user is created at install time. `CTXSYS` can do the following:

- View all indexes
- Sync all indexes
- Run `ctxkbtc`, the knowledge base extension compiler
- Query all system-defined views
- Perform all the tasks of a user with the `CTXAPP` role

> **Note:** In previous releases of Oracle Text, CTXSYS had DBA privileges, and only CTXSYS could perform certain functions, such as modifying system-defined preferences or setting system parameters. See also Chapter 11, "Migrating Applications from Earlier Releases" for more on changes to CTXSYS.

During a manual installation, after installation of the CTXSYS schema is complete, you may want to run `dr0lsys.sql` to lock and expire the CTXSYS schema for security reasons. Alternatively, you can choose a good password for CTXSYS when running `dr0csys.sql`.

CTXAPP Role

The CTXAPP role is a system-defined role that enables users to do the following:

- Create and delete Oracle Text preferences
- Use the Oracle Text PL/SQL packages

Any user can create an Oracle Text index and issue a Text query. The CTXAPP role enables users to create preferences and use the PL/SQL packages.

Granting Roles and Privileges to Users

The system uses the standard SQL model for granting roles to users. To grant a Text role to a user, use the GRANT statement.

In addition, to allow application developers to call procedures in the Oracle Text PL/SQL packages, you must explicitly grant to each user EXECUTE privileges for the Oracle Text package.

> **See Also:** "Creating an Oracle Text User" in Chapter 2, "Getting Started with Oracle Text"

DML Queue

When there are inserts, updates, or deletes to documents in your base table, the DML queue stores the requests for documents waiting to be indexed. When you synchronize the index with CTX_DDL.SYNC_INDEX, requests are removed from this queue.

Pending DML requests can be queried with the CTX_PENDING and CTX_USER_PENDING views.

DML errors can be queried with the CTX_INDEX_ERRORS or CTX_USER_INDEX_ERRORS view.

> **See Also:** *Oracle Text Reference* for more information about these views.

The CTX_OUTPUT Package

Use the CTX_OUTPUT PL/SQL package to log indexing and document service requests.

> **See Also:** *Oracle Text Reference* for more information about this package.

The CTX_REPORT Package

Use the CTX_REPORT package to produce reports on indexes and queries. These reports can help you fine-tune or troubleshoot your applications.

> **See Also:** The CTX_REPORT chapter in the *Oracle Text Reference*

The CTX_REPORT package contains the following procedures:

CTX_REPORT.DESCRIBE_INDEX
CTX_REPORT.DESCRIBE_POLICY
These procedures create reports that describe an existing index or policy, including the settings of the index metadata, the indexing objects used, the settings of the attributes of the objects, and (for CTX_REPORT.DESCRIBE_INDEX) index partition information, if any. These procedures are especially useful for diagnosing index-related problems.

This is sample output from DESCRIBE_INDEX, run on a simple context index:

```
================================================================
                        INDEX DESCRIPTION
================================================================
index name:                     "DR_TEST"."TDRBPRX0"
index id:                       1160
index type:                     context
```

```
base table:                     "DR_TEST"."TDRBPR"
primary key column:             ID
text column:                    TEXT2
text column type:               VARCHAR2(80)
language column:
format column:
charset column:
================================================================
                          INDEX OBJECTS
================================================================
datastore:                      DIRECT_DATASTORE
filter:                         NULL_FILTER
section group:                  NULL_SECTION_GROUP
lexer:                          BASIC_LEXER
wordlist:                       BASIC_WORDLIST
   stemmer:                        ENGLISH
   fuzzy_match:                    GENERIC
stoplist:                       BASIC_STOPLIST
   stop_word:                      teststopword
storage:                        BASIC_STORAGE
   r_table_clause:                 lob (data) store as (cache)
   i_index_clause:                 compress 2
```

CTX_REPORT.CREATE_INDEX_SCRIPT
CTX_REPORT.CREATE_POLICY_SCRIPT

CREATE_INDEX_SCRIPT creates a SQL*Plus script that can create a duplicate of a given text index. Use this when you have an index but don't have the original script (if any) used to create that script and want to be able to re-create the index. For example, if you accidentally drop a script, CREATE_INDEX_SCRIPT can re-create it; likewise, CREATE_INDEX_SCRIPT can be useful if you have inherited indexes from another user but not the scripts that created them.

CREATE_POLICY_SCRIPT does the same thing as CREATE_INDEX_SCRIPT, except that it enables you to re-create a policy instead of an index.

This is sample output from CREATE_INDEX_SCRIPT, run on a simple context index (not a complete listing):

```
begin
  ctx_ddl.create_preference('"TDRBPRX0_DST"','DIRECT_DATASTORE');
end;
/
...
/
```

```
begin
  ctx_ddl.create_section_group('"TDRBPRX0_SGP"','NULL_SECTION_GROUP');
end;
/
...
begin
  ctx_ddl.create_preference('"TDRBPRX0_WDL"','BASIC_WORDLIST');
  ctx_ddl.set_attribute('"TDRBPRX0_WDL"','STEMMER','ENGLISH');
  ctx_ddl.set_attribute('"TDRBPRX0_WDL"','FUZZY_MATCH','GENERIC');
end;
/
begin
  ctx_ddl.create_stoplist('"TDRBPRX0_SPL"','BASIC_STOPLIST');
  ctx_ddl.add_stopword('"TDRBPRX0_SPL"','teststopword');
end;
/
...
/
begin
  ctx_output.start_log('TDRBPRX0_LOG');
end;
/
create index "DR_TEST"."TDRBPRX0"
  on "DR_TEST"."TDRBPR"
      ("TEXT2")
  indextype is ctxsys.context
  parameters('
    datastore       "TDRBPRX0_DST"
    filter          "TDRBPRX0_FIL"
    section group   "TDRBPRX0_SGP"
    lexer           "TDRBPRX0_LEX"
    wordlist        "TDRBPRX0_WDL"
    stoplist        "TDRBPRX0_SPL"
    storage         "TDRBPRX0_STO"
  ')
/
```

CTX_REPORT.INDEX_SIZE

This procedure creates a report showing the names of the internal index objects, along with their tablespaces, allocated sizes, and used sizes. It is useful for DBAs who may need to monitor the size of their indexes (for example, when disk space is at a premium).

Sample output from this procedure looks like this (partial listing):

```
============================================================
               INDEX SIZE FOR DR_TEST.TDRBPRX10
============================================================
TABLE:                          DR_TEST.DR$TDRBPRX10$I
TABLESPACE NAME:                DRSYS
BLOCKS ALLOCATED:                                        4
BLOCKS USED:                                             1
BYTES ALLOCATED:                             8,192 (8.00 KB)
BYTES USED:                                  2,048 (2.00 KB)

INDEX (LOB):                    DR_TEST.SYS_IL0000023161C00006$$
TABLE NAME:                     DR_TEST.DR$TDRBPRX10$I
TABLESPACE NAME:                DRSYS
BLOCKS ALLOCATED:                                        5
BLOCKS USED:                                             2
BYTES ALLOCATED:                           10,240 (10.00 KB)
BYTES USED:                                  4,096 (4.00 KB)

INDEX (NORMAL):                 DR_TEST.DR$TDRBPRX10$X
TABLE NAME:                     DR_TEST.DR$TDRBPRX10$I
TABLESPACE NAME:                DRSYS
BLOCKS ALLOCATED:                                        4
BLOCKS USED:                                             2
BYTES ALLOCATED:                             8,192 (8.00 KB)
BYTES USED:                                  4,096 (4.00 KB)
```

CTX_REPORT.INDEX_STATS

INDEX_STATS produces a variety of calculated statistics about an index, such as how many documents are indexed, how many unique tokens the index contains, average size of its tokens, fragmentation information for the index, and so on. An example of a use of INDEX_STATS might be in optimizing stoplists.

See the *Oracle Text Reference* for an example of the output of this procedure.

CTX_REPORT.QUERY_LOG_SUMMARY

This procedure creates a report of logged queries, which you can use to perform simple analyses. With query analysis, you can find out:

- which queries were made
- which queries were successful
- which queries were unsuccessful
- how many times each query was made

You can combine these factors in various ways, such as determining the 50 most frequent unsuccessful queries made by your application.

See the *Oracle Text Reference* for an example of the output of this procedure.

CTX_REPORT.TOKEN_INFO
`TOKEN_INFO` is used mainly to diagnose query problems; for instance, to check that index data is not corrupted. As an example, you can use it to find out which documents are producing unexpected or bad tokens.

CTX_REPORT.TOKEN_TYPE
This is a lookup function, used mainly as input to other functions (`CTX_DDL.OPTIMIZE_INDEX`, `CTX_REPORT.TOKEN_INFO`, and so on).

Servers

You index documents and issue queries with standard SQL. No server is needed for performing batch DML. You can synchronize the `CONTEXT` index with the `CTX_DDL.SYNC_INDEX` procedure.

> **See Also:** For more information about indexing and index synchronization, see Chapter 3, "Indexing".

Administration Tool

The Oracle Text Manager is a Java application integrated with the Oracle Enterprise Manager.

The Text Manager enables administrators to create preferences, stoplists, sections, and indexes. This tool also enables administrators to perform DML.

> **See Also:** for more information about the Oracle Text Manager, see the online help shipped with this tool.

11

Migrating Applications from Earlier Releases

This chapter covers issues relating to migrating your applications from previous releases of Oracle Text. It also contains a note on migrating *back* from the current release.

Security Improvements in Oracle Text

In previous versions of Oracle Text, CTXSYS had DBA privileges. To tighten security and protect the database in the case of unauthorized access, CTXSYS now has only CONNECT and RESOURCE roles, and only limited, necessary direct grants on some system views and packages. Some applications using Oracle Text may therefore require minor changes in order to work properly with this security change. Here are the major effects of the security improvements, their possible effects on Oracle Text applications, and the steps needed to ensure proper operation in Oracle Database 10*g*.

CTXSYS No Longer Has DBA Permissions

CTXSYS no longer has DBA permissions. This may affect indexes using USER_DATASTORE, PROCEDURE_FILTER, or USER_LEXER objects. For example, suppose that you have an index using a USER_DATASTORE whose procedure is CTXSYS.PROC, and that that procedure refers to other schemas' objects:

```
create procedure proc(r in rowid, d in out nocopy clob)
is
begin
```

```
select text into l_data from scott.example ...
```

Previously, this user datastore would have worked properly because CTXSYS was able to select from any table—namely, SCOTT.EXAMPLE. However, in Oracle Database 10*g*, CTXSYS does not have DBA privileges and is not allowed to select from SCOTT.EXAMPLE. This makes the procedure PROC invalid, which leads to errors when indexing or sync is done for this index.

To resolve this problem, Oracle recommends migrating all user datastores, procedure filters, and user lexers from CTXSYS-owned procedures to index-owner-owned procedures (see "Migrating CTXSYS-Owned Procedures").

Migrating CTXSYS-Owned Procedures

Here are the steps to migrate an index using a CTXSYS-owned procedure to use an index-owner-owned procedure:

1. Create a procedure owned by the index owner that is equivalent to the CTXSYS-owned procedure. If your application's CTXSYS-owned procedure simply calls another procedure owned by the index owner, use that procedure for step 2. Otherwise, copy the code from the CTXSYS-owned procedure into a new procedure owned by the index owner, making any needed changes for the change in schema.
2. Create a new user datastore, procedure filter, or user lexer preference that uses the index-owner-owned procedure. Alternatively, you can modify the existing preference using CTX_DDL.SET_ATTRIBUTE, if the preference used to create the index still exists.
3. Replace the existing datastore or filter or lexer with the new, updated preference using the new REPLACE METADATA command. For instance, to replace a user datastore:

   ```
   alter index <myindex> rebuild parameters ('replace metadata datastore <new_
   datastore_preference>');
   ```

   REPLACE METADATA does not rebuild the index, so this command will not affect existing index data.

See Also: "Migrating Back to Previous Releases" on page 11-4

Effective User During Indexing

In previous versions of Oracle Text, the effective user during indexing or sync was CTXSYS. As a result, CTXSYS required execute permission on all BFILE directories,

execute permission on any procedures called from user datastores, procedure filters, or user lexers, and the CTXSYS user's TEMP tablespace was used during indexing. In Oracle Database 10*g*, the effective user during indexing is the index owner, which eliminates these caveats.

Procedures Do Not Need to Be Owned by CTXSYS

Previously, procedures used in user datastores, procedure filters, and user lexers had to be owned by CTXSYS. In Oracle Database 10*g*, these procedures can be owned by any schema, so long as the index owner has execute privileges on them.

This principally affects creation of preferences. In previous releases of Oracle Text, a user datastore created with:

```
begin
ctx_ddl.create_preference('example','user_datastore');
ctx_ddl.set_attribute('example','procedure','proc');
end;
```

would have used the procedure CTXSYS.PROC. However, in Oracle Database 10*g*, standard Oracle Database rules are applied to the input "PROC," and this resolves to USER.PROC. Any application code that creates user datastores, procedure filters, or user lexers should either create the preferences as the owner of the procedure, or prepend the correct owner name to the procedure name. For example:

```
ctx_ddl.set_attribute('example','procedure','user.proc');
```

Synching and Optimizing of Other Users' Indexes

In previous versions of Oracle Text, only the owner of the index and CTXSYS were allowed to sync or optimize an index through CTX_DDL.SYNC_INDEX and CTX_DDL.OPTIMIZE_INDEX. In Oracle Database 10*g*, any user with the ALTER ANY INDEX system privilege is also allowed to sync or optimize any index.

CTX Packages and Invoker's Rights

Most public CTX packages, such as CTX_DDL, CTX_QUERY, and CTX_REPORT, are now invoker's rights packages.

CREATE TABLE Permissions

In Oracle Database 10*g*, if a text index is created by one user for another user, or if the create index statement is issued from a PL/SQL block, the index owner must be

granted the CREATE TABLE privilege in order for the indexing to succeed. Even if the index owner has the RESOURCE role, CREATE TABLE must be specifically granted.

Migrating Back to Previous Releases

During the upgrade to Oracle Database 10*g*, Oracle Text drops a number of procedures belonging to CTXSYS. (These procedures are invalid under Oracle Database 10*g* and have the name format DR$*indexid*$U.) If you migrate back to a pre-10*g* release of Oracle Database, you must re-create these procedures in order for DML to work. To do this, *after* the backward migration—once all the pre-10*g* packages have been reinstalled—rename each CTXCAT index; the rename code will re-create that procedure. (You can rename the procedures back if you want to retain the original names).

A

CONTEXT Query Application

This appendix describes how to build a simple Web search application using the CONTEXT index type, whether by writing your own code or using the Oracle Text Wizard. The following topics are covered:

- Web Query Application Overview
- The PSP Web Application
- The JSP Web Application

Web Query Application Overview

A common use of Oracle Text is to index HTML files on Web sites and provide search capabilities to users. The sample application in this appendix indexes a set of HTML files stored in the database and uses a Web server connected to Oracle Database to provide the search service.

This appendix describes two versions of the Web query application:

- one using PL/SQL Server Pages (PSP)
- one using Java Server Pages (JSP).

Both versions of these applications can be produced by means of a query application wizard, which produces the necessary code automatically.

You can view and download both the PSP and JSP application code, as well as the text query application wizard, at the Oracle Technology Network Web site:

```
http://otn.oracle.com/products/text
```

The text query application wizard Web page also contains complete instructions on how to use the wizard.

Figure A–1 shows what the JSP version of the text query application looks like. This application was created with the Oracle Text application wizard.

Figure A–1 The Text Query Application

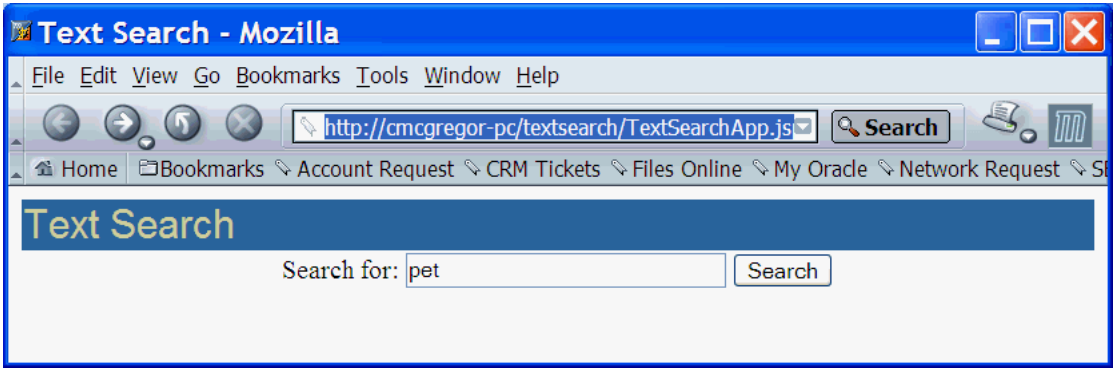
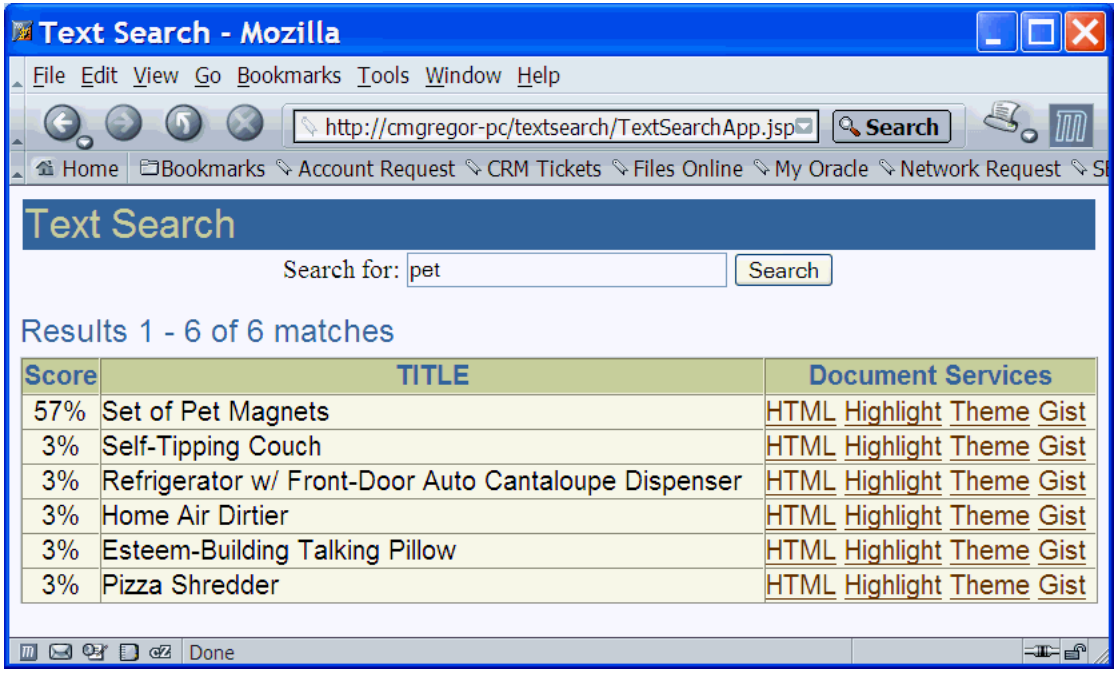


Figure A–2 shows the results of the text query.

Figure A–2 The Text Query Application with Results



The application returns links to documents containing the search term. Each document has four links:

- The **HTML** link displays the document.

  Graphics are not displayed in the filtered document. (You can see the source document for the first hit by looking at Figure 5–1 on page 5-9.)

- The **Highlight** link displays the document with the search term highlighted. Figure 5–2 on page 5-10 shows an example of highlighting.

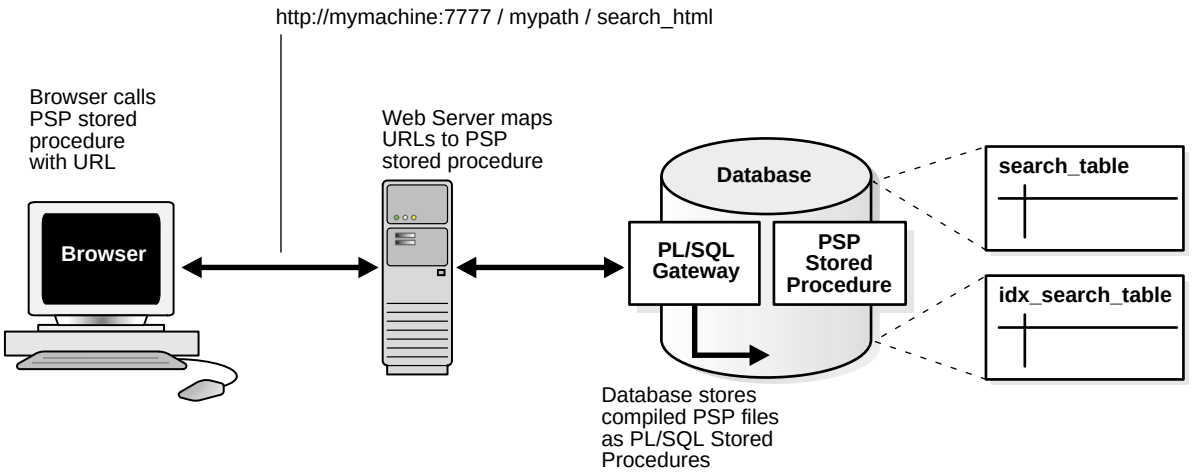
- The **Theme** link shows the top 50 themes associated with the document. Figure 5–3 on page 5-11 shows an example of theme extraction.

- The **Gist** link displays a short summary of the document. Figure 5–4 on page 5-12 shows an example of this gisting feature.

The PSP Web Application

This application is based on PL/SQL server pages. Figure A–3 illustrates how the browser calls the PSP-stored procedure on Oracle Database through a Web server.

Figure A–3 The PSP Web Application



Web Application Prerequisites

This application has the following requirements:

- Your Oracle Database (version 8.1.6 or higher) is up and running.
- You have the Oracle PL/SQL gateway running
- You have a Web server such as Apache up and running and correctly configured to send requests to the Oracle Database server.

Building the Web Application

This section describes how to build the PSP Web application.

Step 1 Create your Text Table

You must create a text table to store your HTML files. This example creates a table called `search_table` as follows:

```
create table search_table (tk numeric primary key, title varchar2(2000), text
clob);
```

Step 2 Load HTML Documents into Table Using SQL*Loader

You must load the text table with the HTML files. This example uses the control file loader.ctl to load the files named in loader.dat. The SQL*Loader command is as follows:

```
% sqlldr userid=scott/tiger control=loader.ctl
```

Step 3 Create the CONTEXT index

If you are using the text query wizard: The wizard produces a script to create an index. (See the instructions on the download Web page for the wizard.) Run that script.

If you are not using the wizard: Index the HTML files by creating a `CONTEXT` index on the text column as follows. Since you are indexing HTML, this example uses the `NULL_FILTER` preference type for no filtering and uses the `HTML_SECTION_GROUP` type:

```
create index idx_search_table on search_table(text)
  indextype is ctxsys.context parameters
  ('filter ctxsys.null_filter section group CTXSYS.HTML_SECTION_GROUP');
```

Step 4 Compile search_htmlservices Package in Oracle Database

The application must present selected documents to the user. To do so, Oracle Database must read the documents from the `CLOB` in `search_table` and output the result for viewing, This is done by calling procedures in the search_htmlservices package. The file search_htmlservices.sql must be compiled. You can do this at the SQL*Plus prompt:

```
SQL> @search_htmlservices.sql

Package created.
```

Step 5 Compile the search_html PSP page with loadpsp

The search page is invoked by calling search_html.psp from a browser. You compile search_html in Oracle Database with the `loadpsp` command-line program:

```
% loadpsp -replace -user scott/tiger search_html.psp
"search_html.psp": procedure "search_html" created.
```

See Also: *Oracle Database Application Developer's Guide - Fundamentals* for more information about using PSP.

Step 6 Configure Your Web Server

You must configure your Web server to accept client PSP requests as a URL. Your Web server forwards these requests to the Oracle Database server and returns server output to the browser. Refer to Figure A–3 on page A-4.

You can use the Oracle WebDB Web listener or Oracle Application Server, which includes the Apache Web server. See your Web server documentation for more information.

Step 7 Issue Query from Browser

You can access the query application from a browser using a URL. You configure the URL with your Web server. An example URL might look like:

```
http://mymachine:7777/mypath/search_html
```

The application displays a query entry box in your browser and returns the query results as a list of HTML links, as shown in Figure A–1 on page A-2 and Figure A–2 on page A-3.

PSP Sample Code

This section lists the code used to build the example Web application. It includes the following files:

- loader.ctl
- loader.dat
- search_htmlservices.sql
- search_html.psp

See Also: http://otn.oracle.com/products/text/

loader.ctl

This example shows a sample `loader.ctl` file. It is used by `sqlldr` to load the data file, `loader.dat`.

```
LOAD DATA
        INFILE 'loader.dat'
```

```
INTO TABLE search_table
REPLACE
FIELDS TERMINATED BY ';'
(tk             INTEGER,
 title          CHAR,
 text_file      FILLER CHAR,
 text           LOBFILE(text_file) TERMINATED BY EOF)
```

loader.dat

This example shows a sample `loader.dat` file. Each row contains three fields: a reference number for the document, a label (or "title"), and the name of the HTML document to load into the text column of `search_table`. The file has been truncated for this example.

```
1;   Pizza Shredder;Pizza.html
2;   Refrigerator w/ Front-Door Auto Cantaloupe Dispenser;Cantaloupe.html
3;   Self-Tipping Couch;Couch.html
4;   Home Air Dirtier;Mess.html
5;   Set of Pet Magnets;Pet.html
6;   Esteem-Building Talking Pillow;Snooze.html
      . . .
28;   Shaggy Found Inspiration For Success In Jamaica ;shaggy_found.html
29;   Solar Flare Eruptions Likely ;solar_flare.html
30;   Supersonic Plane Breaks Food Barrier ;food_barrier.html
31;   SOUNDSCAN REPORT: Recipe for An Aspiring Top Ten;urban_groove_1.html
      . . .
```

search_htmlservices.sql

```
set define off
create or replace package search_htmlServices as
  procedure showHTMLDoc (p_id in numeric);
  procedure showDoc  (p_id in varchar2, p_query in varchar2);
end;
/
show errors;

create or replace package body search_htmlServices as

  procedure showHTMLDoc (p_id in numeric) is
    v_clob_selected   CLOB;
    v_read_amount     integer;
    v_read_offset     integer;
```

```
    v_buffer          varchar2(32767);
   begin


     select text into v_clob_selected from search_table where tk = p_id;
     v_read_amount := 32767;
     v_read_offset := 1;
   begin
    loop
      dbms_lob.read(v_clob_selected,v_read_amount,v_read_offset,v_buffer);
      htp.print(v_buffer);
      v_read_offset := v_read_offset + v_read_amount;
      v_read_amount := 32767;
    end loop;
   exception
   when no_data_found then
     null;
   end;
 end showHTMLDoc;


procedure showDoc (p_id in varchar2, p_query in varchar2) is

 v_clob_selected   CLOB;
 v_read_amount     integer;
 v_read_offset     integer;
 v_buffer          varchar2(32767);
 v_query           varchar(2000);
 v_cursor          integer;

 begin
   htp.p('<html><title>HTML version with highlighted terms</title>');
   htp.p('<body bgcolor="#ffffff">');
   htp.p('<b>HTML version with highlighted terms</b>');

   begin
     ctx_doc.markup (index_name => 'idx_search_table',
                     textkey    => p_id,
                     text_query => p_query,
                     restab     => v_clob_selected,
                     starttag   => '<i><font color=red>',
                     endtag     => '</font></i>');

     v_read_amount := 32767;
     v_read_offset := 1;
```

```
     begin
      loop
        dbms_lob.read(v_clob_selected,v_read_amount,v_read_offset,v_buffer);
        htp.print(v_buffer);
        v_read_offset := v_read_offset + v_read_amount;
        v_read_amount := 32767;
      end loop;
     exception
      when no_data_found then
         null;
     end;

     exception
      when others then
        null; --showHTMLdoc(p_id);
   end;
end showDoc;
end;
/
show errors


set define on
```

search_html.psp

```
<%@ plsql procedure="search_html" %>
<%@ plsql parameter="query" default="null" %>
<%! v_results numeric := 0; %>

<html>
<head>
  <title>search_html Search </title>
</head>
<body>

<%

If query is null Then
%>

  <center>
    <form method=post action="search_html">
     <b>Search for: </b>
```

```
    <input type=text name="query" size=30> 
    <input type=submit value=Search>
  </center>
<hr>

<%
  Else
%>

   <p>
   <%!
      color varchar2(6) := 'ffffff';
   %>

   <center>
     <form method=post action="search_html">
      <b>Search for:</b>
      <input type=text name="query" size=30 value="<%= query %>">
      <input type=submit value=Search>
     </form>
   </center>
   <hr>
   <p>

   <%
     -- select statement
    for doc in (
                select /*+ FIRST_ROWS */ rowid, tk, title, score(1) scr
                from search_table
                where contains(text, query,1) >0
                order by score(1) desc
               )
         loop
           v_results := v_results + 1;
           if v_results = 1 then

   %>

             <center>
              <table border="0">
                <tr bgcolor="#6699CC">
                  <th>Score</th>
                  <th>Title</th>
                </tr>
```

```
<%      end if; %>
        <tr bgcolor="#<%= color %>">
         <td> <%= doc.scr %>% </td>
         <td> <%= doc.title %>
         [<a href="search_htmlServices.showHTMLDoc?p_id=
                <%= doc.tk %>">HTML</a>]
         [<a href="search_htmlServices.showDoc?p_id=
                <%= doc.tk %>&p_query=<%= query %>">Highlight</a>]
         </td>
        </tr>

 <%
        if (color = 'ffffff') then
             color := 'eeeeee';
           else
             color := 'ffffff';
        end if;

   end loop;
 %>

  </table>
 </center>

<%
  end if;
%>
</body></html>
```

The JSP Web Application

Creating the JSP-based Web application involves most of the same steps as those used in building the PSP-based application (see "Building the Web Application" on page A-4). You can use the same `loader.dat` and `loader.ctl` files. However, with the JSP-based application, you do not need to do the following:

- compile the `search_htmlservices` package
- compile the `search_html` PSP page with `loadpsp`

Web Application Prerequisites

This application has the following requirements:

- Your Oracle database (version 8.1.6 or higher) is up and running.
- You have a Web server such as Apache up and running and correctly configured to send requests to the Oracle Database server.

JSP Sample Code

This section lists the Java code used to build the example Web application. It includes the following files:

- search_html.jsp

  The code for this file was generated by the text query application wizard. (Some longer lines have been split to make the code easier to read.)

search_html.jsp

```
<%@ page import="java.sql.*, java.util.*, java.net.*, oracle.jdbc.*,
oracle.jsp.dbutil.*" %>
<%@ page contentType="text/html;charset=UTF-8" %>
<% oracle.jsp.util.PublicUtil.setReqCharacterEncoding(request, "UTF-8"); %>
<jsp:useBean id="name" class="oracle.jsp.jml.JmlString" scope ="request" >
<jsp:setProperty name="name" property="value" param="query" />
</jsp:useBean>

<%
String connStr="jdbc:oracle:thin:@mburtch-pc.us.oracle.com:1521:zippy922";

java.util.Properties info=new java.util.Properties();
Connection conn = null;
ResultSet rset = null;
OracleCallableStatement callStmt = null;
Statement stmt = null;
String userQuery = null;
String myQuery = null;
URLEncoder myEncoder;
int count=0;
int loopNum=0;
int startNum=0;
if (name.isEmpty()) {
%>
  <html>
    <title>Text Search</title>
    <body>
      <table width="100%">
        <tr bgcolor="#336699">
```

```
          <td><font face="arial, helvetica" align="left"
          color="#CCCC99" size=+2>Text Search</td>
        </tr>
      </table>
    <center>
      <form method = post>
      Search for:
      <input type=text name=query size = 30>
      <input type=submit value="Search">
      </form>
    </center>
    </body>
  </html>

<%
}

else {
%>
  <html>
    <title>Text Search</title>
    <body text="#000000" bgcolor="#FFFFFF" link="#663300"
          vlink="#996633" alink="#ff6600">
      <table width="100%">
        <tr bgcolor="#336699">
          <td><font face="arial, helvetica" align="left"
                color="#CCCC99" size=+2>Text Search</td>
        </tr>
      </table>
    <center>
      <form method = post action="TextSearchApp.jsp">
      Search for:
      <input type=text name="query" value="<%=name.getValue() %>" size = 30>
      <input type=submit value="Search">
      </form>
    </center>
<%
  try {
    DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver() );
    info.put ("user", "mburtch");
    info.put ("password","welcome");
    conn = DriverManager.getConnection(connStr,info);
    stmt = conn.createStatement();
    userQuery =   request.getParameter("query");
    myQuery =   URLEncoder.encode(userQuery);
```

```
    String numStr =   request.getParameter("sn");
    if(numStr!=null)
      startNum=Integer.parseInt(numStr);
    String theQuery =   translate(userQuery);
    callStmt =(OracleCallableStatement)conn.prepareCall("begin "+
         "?:=ctx_query.count_hits(index_name=>'ULTRA_IDX1', "+
         "text_query=>?"+
         "); " +
         "end; ");
    callStmt.setString(2,theQuery);
    callStmt.registerOutParameter(1, OracleTypes.NUMBER);
    callStmt.execute();
    count=((OracleCallableStatement)callStmt).getNUMBER(1).intValue();
    if(count>=(startNum+20)){
%>
    <font color="#336699" FACE="Arial,Helvetica" SIZE=+1>Results
           <%=startNum+1%> - <%=startNum+20%> of <%=count%> matches
<%
    }
    else if(count>0){
%>
    <font color="#336699" FACE="Arial,Helvetica" SIZE=+1>Results
           <%=startNum+1%> - <%=count%> of <%=count%> matches
<%
    }
    else {
%>
    <font color="#336699" FACE="Arial,Helvetica" SIZE=+1>No match found
<%
    }
%>
  <table width="100%">
  <TR ALIGN="RIGHT">
<%
  if((startNum>0)&(count<=startNum+20))
  {
%>
    <TD ALIGN="RIGHT">
    <a href="TextSearchApp.jsp?sn=<%=startNum-20 %>&query=
            <%=myQuery %>">previous20</a>
    </TD>
<%
  }
  else if((count>startNum+20)&(startNum==0))
  {
```

```
%>
    <TD ALIGN="RIGHT">
    <a href="TextSearchApp.jsp?sn=<%=startNum+20
          %>&query=<%=myQuery %>">next20</a>
    </TD>
<%
  }
  else if((count>startNum+20)&(startNum>0))
  {
%>
    <TD ALIGN="RIGHT">
    <a href="TextSearchApp.jsp?sn=<%=startNum-20 %>&query=
              <%=myQuery %>">previous20</a>
    <a href="TextSearchApp.jsp?sn=<%=startNum+20 %>&query=
              <%=myQuery %>">next20</a>
    </TD>
<%
  }
%>
  </TR>
  </table>
<%
    String ctxQuery = "select /*+ FIRST_ROWS */ rowid, 'TITLE',
     score(1) scr from 'ULTRA_TAB1' where contains('TEXT', '"+theQuery+"',1 )
     > 0 order by score(1) desc";
    rset = stmt.executeQuery(ctxQuery);
    String color = "ffffff";
    String rowid = null;
    String fakeRowid = null;
    String[] colToDisplay = new String[1];
    int myScore = 0;
    int items = 0;
    while (rset.next()&&items< 20) {
      if(loopNum>=startNum)
      {
        rowid = rset.getString(1);
        fakeRowid = URLEncoder.encode(rowid);
        colToDisplay[0] = rset.getString(2);
        myScore = (int)rset.getInt(3);
        items++;
        if (items == 1) {
%>
        <center>
          <table BORDER=1 CELLSPACING=0 CELLPADDING=0 width="100%"
            <tr bgcolor="#CCCC99">
```

```
              <th><font face="arial, helvetica" color="#336699">Score</th>
              <th><font face="arial, helvetica" color="#336699">TITLE</th>
              <th> <font face="arial, helvetica"
                       color="#336699">Document Services</th>
            </tr>
<%   } %>
      <tr bgcolor="#FFFFE0">
        <td ALIGN="CENTER"> <%= myScore %>%</td>
        <td> <%= colToDisplay[0] %>
        <td>
        </td>
      </tr>
<%
      if (color.compareTo("ffffff") == 0)
        color = "eeeeee";
      else
        color = "ffffff";
      }
      loopNum++;
    }
} catch (SQLException e) {
%>
    <b>Error: </b> <%= e %><p>
<%
} finally {
  if (conn != null) conn.close();
  if (stmt != null) stmt.close();
  if (rset != null) rset.close();
  }
%>
  </table>
  </center>
  <table width="100%">
  <TR ALIGN="RIGHT">
<%
  if((startNum>0)&(count<=startNum+20))
  {
%>
    <TD ALIGN="RIGHT">
    <a href="TextSearchApp.jsp?sn=<%=startNum-20 %>&query=
              <%=myQuery %>">previous20</a>
    </TD>
<%
  }
  else if((count>startNum+20)&(startNum==0))
```

```
  {
%>
    <TD ALIGN="RIGHT">
    <a href="TextSearchApp.jsp?sn=<%=startNum+20 %>&query=
          <%=myQuery %>">next20</a>
    </TD>
<%
  }
  else if((count>startNum+20)&(startNum>0))
  {
%>
    <TD ALIGN="RIGHT">
    <a href="TextSearchApp.jsp?sn=<%=startNum-20 %>&query=
          <%=myQuery %>">previous20</a>
    <a href="TextSearchApp.jsp?sn=<%=startNum+20 %>&query=
          <%=myQuery %>">next20</a>
    </TD>
<%
  }
%>
  </TR>
  </table>
  </body></html>
<%
}

%>
<%!
   public String translate (String input)
   {
      Vector reqWords = new Vector();
      StringTokenizer st = new StringTokenizer(input, " '", true);
      while (st.hasMoreTokens())
      {
        String token = st.nextToken();
        if (token.equals("'"))
        {
           String phrase = getQuotedPhrase(st);
           if (phrase != null)
           {
              reqWords.addElement(phrase);
           }
        }
        else if (!token.equals(" "))
        {
```

```
            reqWords.addElement(token);
          }
        }
        return getQueryString(reqWords);
    }

    private String getQuotedPhrase(StringTokenizer st)
    {
        StringBuffer phrase = new StringBuffer();
        String token = null;
        while (st.hasMoreTokens() && (!(token = st.nextToken()).equals("'")))
        {
          phrase.append(token);
        }
        return phrase.toString();
    }


    private String getQueryString(Vector reqWords)
    {
        StringBuffer query = new StringBuffer("");
        int length = (reqWords == null) ? 0 : reqWords.size();
        for (int ii=0; ii < length; ii++)
        {
           if (ii != 0)
           {
             query.append(" & ");
           }
           query.append("{");
           query.append(reqWords.elementAt(ii));
           query.append("}");
        }
        return query.toString();
    }
%>
```

B

CATSEARCH Query Application

This appendix describes how to build a simple Web-search application using the CATSEARCH index type, whether by writing your own code or using the Oracle Text Wizard. The following topics are covered:

- CATSEARCH Web Query Application Overview
- The JSP Web Application

CATSEARCH Web Query Application Overview

The CTXCAT index type is well suited for merchandise catalogs that have short descriptive text fragments and associated structured data. This appendix describes how to build a browser based bookstore catalog that users can search to find titles and prices.

This application is written in Java Server Pages (JSP).

The application can be produced by means of a catalog query application wizard, which produces the necessary code automatically. You can view and download the JSP application code, as well as the catalog query application wizard, at the Oracle Technology Network Web site:

```
http://otn.oracle.com/products/text
```

This Web site also has complete instructions on how to use the catalog query wizard.

The JSP Web Application

This application is based on Java Server pages and has the following requirements:

- Your Oracle Database (version 8.1.7 or higher) is up and running.

- You have a Web server such as Apache up and running and correctly configured to send requests to the Oracle Database server.

Building the JSP Web Application

This application models an online bookstore where you can look up book titles and prices.

Step 1 Create Your Table

You must create the table to store book information such as title, publisher, and price. From SQL*Plus:

```
sqlplus>create table book_catalog (
          id        numeric,
          title     varchar2(80),
          publisher varchar2(25),
          price     numeric )
```

Step 2 Load data using SQL*Loader

You load the book data from the operating system command line with SQL*Loader:

```
% sqlldr userid=ctxdemo/ctxdemo control=loader.ctl
```

Step 3 Create index set

You can create the index set from SQL*Plus:

```
sqlplus>begin
          ctx_ddl.create_index_set('bookset');
          ctx_ddl.add_index('bookset','price');
          ctx_ddl.add_index('bookset','publisher');
        end;
/
```

Step 4 Index creation

You can create the CTXCAT index from SQL*Plus as follows:

```
sqlplus>create index book_idx on book_catalog (title)
        indextype is ctxsys.ctxcat
        parameters('index set bookset');
```

Step 5 Try a simple search using CATSEARCH

You can test the newly created index in SQL*Plus as follows:

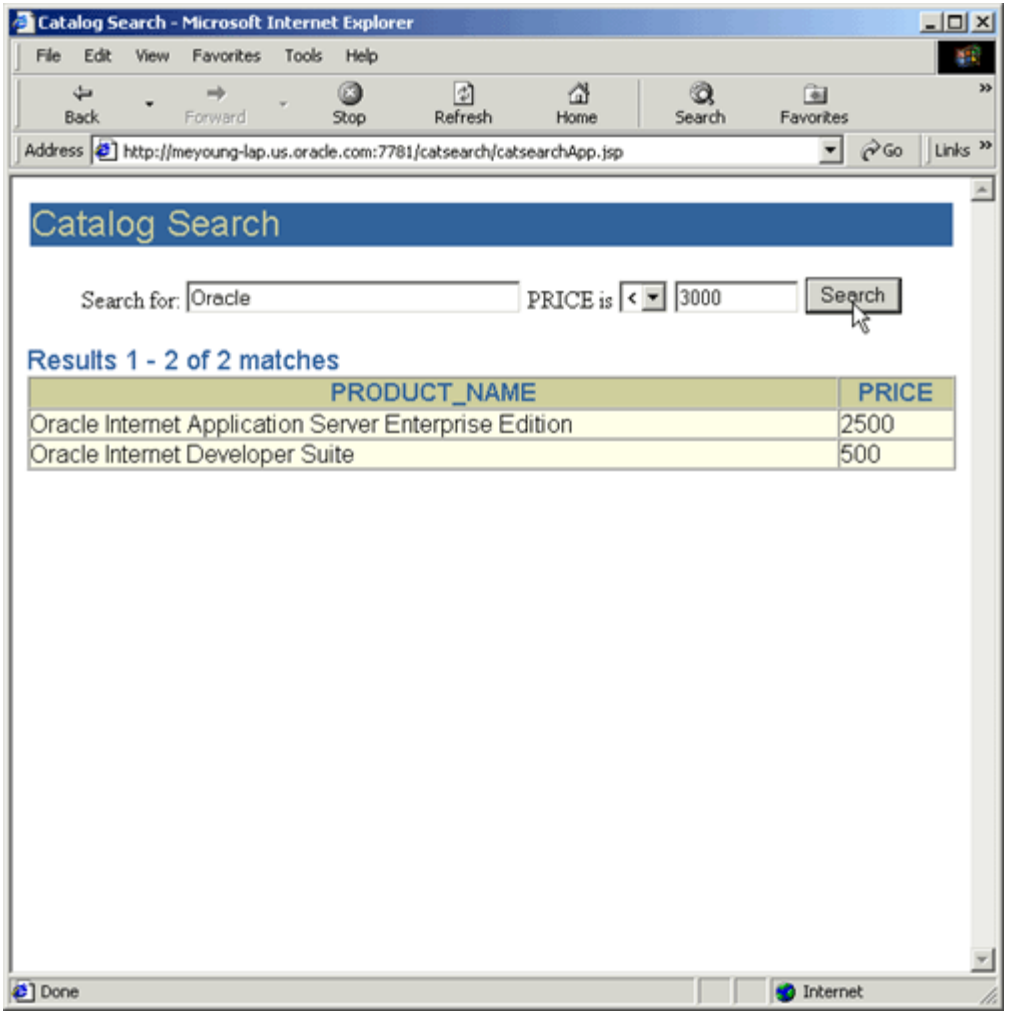
```
sqlplus>select id, title from book_catalog
        where catsearch(title,'Java','price > 10 order by price') > 0
```

Step 6 Copy the catalogSearch.jsp file to your Web site JSP directory.

When you do so, you can access the application from a browser. The URL should be `http://`*localhost*`:`*port*`/`*path*`/catalogSearch.jsp`

The application displays a query entry box in your browser and returns the query results as a list of HTML links. See Figure B–1.

Figure B–1 Screen shot of Web Query Application



JSP Sample Code

This section lists the code used to build the example Web application. It includes the following files:

- loader.ctl
- loader.dat

- catalogSearch.jsp

See Also: http://otn.oracle.com/products/text/

loader.ctl

```
LOAD DATA
  INFILE 'loader.dat'
  INTO TABLE book_catalog
  REPLACE
  FIELDS TERMINATED BY ';'
  (id, title, publisher, price)
```

loader.dat

```
1; A History of Goats; SPINDRIFT BOOKS;50
2; Robust Recipes Inspired by Eating Too Much;SPINDRIFT BOOKS;28
3; Atlas of Greenland History; SPINDRIFT BOOKS; 35
4; Bed and Breakfast Guide to Greenland; SPINDRIFT BOOKS; 37
5; Quitting Your Job and Running Away; SPINDRIFT BOOKS;25
6; Best Noodle Shops of Omaha ;SPINDRIFT BOOKS; 28
7; Complete Book of Toes; SPINDRIFT BOOKS;16
8; Complete Idiot's Guide to Nuclear Technology;SPINDRIFT BOOKS; 28
9; Java Programming for Woodland Animals; LOW LIFE BOOK CO; 10
10; Emergency Surgery Tips and Tricks;SPOT-ON PUBLISHING;10
11; Programming with Your Eyes Shut; KLONDIKE BOOKS; 10
12; Forest Fires of North America, 1858-1882; CALAMITY BOOKS; 11
13; Spanish in Twelve Minutes; WRENCH BOOKS 11
14; Better Sex and Romance Through C++; CALAMITY BOOKS; 12
15; Oracle Internet Application Server Enterprise Edition; KANT BOOKS; 12
16; Oracle Internet Developer Suite; SPAMMUS BOOK CO;13
17; Telling the Truth to Your Pets; IBEX BOOKS INC; 13
18; Go Ask Alice's Restaurant;HUMMING BOOKS;    13
19; Life Begins at 93;  CALAMITY BOOKS;   17
20; Dating While Drunk;  BALLAST BOOKS;  14
21; The Second-to-Last Mohican; KLONDIKE BOOKS;  14
22; Eye of Horus; An Oracle of Ancient Egypt; BIG LITTLE BOOKS;     15
23; Introduction to Sitting Down; IBEX BOOKS INC;  15
```

catalogSearch.jsp

```
<%@ page import="java.sql.* , oracle.jsp.dbutil.*" %>
<jsp:useBean id="name" class="oracle.jsp.jml.JmlString" scope="request" >
<jsp:setProperty name="name" property="value" param="v_query" />
```

```
</jsp:useBean>

<%
  String connStr="jdbc:oracle:thin:@machine-domain-name:1521:betadev";

  java.util.Properties info = new java.util.Properties();

  Connection conn = null;
  ResultSet  rset = null;
  Statement  stmt = null;



       if (name.isEmpty() ) {

%>
           <html>
             <title>Catalog Search</title>
             <body>
             <center>
               <form method=post>
               Search for book title:
               <input type=text name="v_query" size=10>
               where publisher is
               <select name="v_publisher">
                  <option value="ADDISON WESLEY">ADDISON WESLEY
                  <option value="HUMMING BOOKS">HUMMING BOOKS
                  <option value="WRENCH BOOKS">WRENCH BOOKS
                  <option value="SPOT-ON PUBLISHING">SPOT-ON PUBLISHING
                  <option value="SPINDRIFT BOOKS">SPINDRIFT BOOKS
                  <option value="LOW LIFE BOOK CO">LOW LIFE BOOK CO
                  <option value="KLONDIKE BOOKS">KLONDIKE BOOKS
                  <option value="CALAMITY BOOKS">CALAMITY BOOKS
                  <option value="IBEX BOOKS INC">IBEX BOOKS INC
                  <option value="BIG LITTLE BOOKS">BIG LITTLE BOOKS
               </select>
               and price is
               <select name="v_op">
                 <option value="=">=
                 <option value="&lt;">&lt;
                 <option value="&gt;">&gt;
               </select>
               <input type=text name="v_price" size=2>
               <input type=submit value="Search">
               </form>
```

```
            </center>
            <hr>
            </body>
          </html>

<%
      }
      else {

         String v_query = request.getParameter("v_query");
 String v_publisher = request.getParameter("v_publisher");
         String v_price = request.getParameter("v_price");
         String v_op    = request.getParameter("v_op");
%>

         <html>
           <title>Catalog Search</title>
           <body>
           <center>
            <form method=post action="catalogSearch.jsp">
            Search for book title:
            <input type=text name="v_query" value=
            <%= v_query %>
            size=10>
            where publisher is
            <select name="v_publisher">
                  <option value="ADDISON WESLEY">ADDISON WESLEY
                  <option value="HUMMING BOOKS">HUMMING BOOKS
                  <option value="WRENCH BOOKS">WRENCH BOOKS
                  <option value="SPOT-ON PUBLISHING">SPOT-ON PUBLISHING
                  <option value="SPINDRIFT BOOKS">SPINDRIFT BOOKS
                  <option value="LOW LIFE BOOK CO">LOW LIFE BOOK CO
                  <option value="KLONDIKE BOOKS">KLONDIKE BOOKS
                  <option value="CALAMITY BOOKS">CALAMITY BOOKS
                  <option value="IBEX BOOKS INC">IBEX BOOKS INC
                  <option value="BIG LITTLE BOOKS">BIG LITTLE BOOKS
            </select>
            and price is
            <select name="v_op">
               <option value="=">=
               <option value="&lt;">&lt;
               <option value="&gt;">&gt;
            </select>
            <input type=text name="v_price" value=
            <%= v_price %> size=2>
```

```
            <input type=submit value="Search">
            </form>
            </center>

<%
     try {

       DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver() );
       info.put ("user", "ctxdemo");
       info.put ("password","ctxdemo");
       conn = DriverManager.getConnection(connStr,info);

         stmt = conn.createStatement();
         String theQuery = request.getParameter("v_query");
         String thePrice = request.getParameter("v_price");

 // select id,title
 // from book_catalog
 // where catsearch (title,'Java','price >10 order by price') > 0

 // select title
 // from book_catalog
 // where catsearch(title,'Java','publisher = ''CALAMITY BOOKS''
          and price < 40 order by price' )>0

         String myQuery = "select title, publisher, price from book_catalog
             where catsearch(title, '"+theQuery+"',
             'publisher = '''+v_publisher+"''' and price "+v_op+thePrice+"
             order by price' ) > 0";
         rset = stmt.executeQuery(myQuery);

         String color = "ffffff";

         String myTitle = null;
         String myPublisher = null;
         int myPrice = 0;
         int items = 0;

         while (rset.next()) {
            myTitle     = (String)rset.getString(1);
    myPublisher = (String)rset.getString(2);
            myPrice     = (int)rset.getInt(3);
            items++;

            if (items == 1) {
```

```
%>
               <center>
                  <table border="0">
                     <tr bgcolor="#6699CC">
                       <th>Title</th>
       <th>Publisher</th>
       <th>Price</th>
                     </tr>
<%
            }
%>
            <tr bgcolor="#<%= color %>">
             <td> <%= myTitle %></td>
             <td> <%= myPublisher %></td>
     <td> $<%= myPrice %></td>
            </tr>
<%
            if (color.compareTo("ffffff") == 0)
               color = "eeeeee";
             else
               color = "ffffff";

      }

   } catch (SQLException e) {

%>

      <b>Error: </b> <%= e %><p>

<%

  } finally {
       if (conn != null) conn.close();
       if (stmt != null) stmt.close();
       if (rset != null) rset.close();
   }

%>
    </table>
    </center>
   </body>
   </html>
<%
 }
```

```
%>
```

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

Z