Oracle® Text

Application Developer's Guide

11*g* Release 2 (11.2)

E16594-01

August 2010

ORACLE®

Oracle Text Application Developer's Guide, 11*g* Release 2 (11.2)

E16594-01

Copyright © 2004, 2010, Oracle and/or its affiliates. All rights reserved.

Primary Author: Cathy Shea, Paul Lane

Contributors: Mohammad Faisal, Roger Ford, Wesley Lin, Yasuhiro Matsuda

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this software or related documentation is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications which may create a risk of personal injury. If you use this software in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use of this software. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software in dangerous applications.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

This software and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Contents

Preface xiii

Audience xiii
Documentation Accessibility xiii
Related Documents xiv
Conventions xiv

1 Understanding Oracle Text Application Development

Introduction to Oracle Text 1-1
Document Collection Applications 1-1
Flowchart of Text Query Application 1-2
Catalog Information Applications 1-3
Flowchart for Catalog Query Application 1-3
Document Classification Applications 1-4
XML Search Applications 1-5
Using Oracle Text 1-5
Using the Oracle XML DB Framework 1-6
Combining Oracle Text features with Oracle XML DB 1-6
Using the Text-on-XML Method 1-6
Using the XML-on-Text Method 1-7

2 Getting Started with Oracle Text

Overview of Getting Started with Oracle Text 2-1
Creating an Oracle Text User 2-1
Query Application Quick Tour 2-2
Building Web Applications with the Oracle Text Wizard 2-4
Oracle JDeveloper 2-5
Oracle Text Wizard Addins 2-5
Oracle Text Wizard Instructions 2-5
Catalog Application Quick Tour 2-5
Classification Application Quick Tour 2-7
Steps for Creating a Classification Application 2-8

3 Indexing with Oracle Text

About Oracle Text Indexes 3-1
Types of Oracle Text Indexes 3-1

Structure of the Oracle Text CONTEXT Index 3-5
Merged Word and Theme Index 3-5
The Oracle Text Indexing Process 3-5
Datastore Object 3-6
Filter Object 3-6
Sectioner Object 3-6
Lexer Object 3-7
Indexing Engine 3-7
Partitioned Tables and Indexes 3-7
Querying Partitioned Tables 3-7
Creating an Index Online 3-8
Parallel Indexing 3-8
Indexing and Views 3-8
Considerations For Indexing 3-9
Location of Text 3-9
Supported Column Types 3-10
Storing Text in the Text Table 3-10
Storing File Path Names 3-11
Storing URLs 3-11
Storing Associated Document Information 3-11
Format and Character Set Columns 3-11
Supported Document Formats 3-12
Summary of DATASTORE Types 3-12
Document Formats and Filtering 3-12
No Filtering for HTML 3-13
Filtering Mixed-Format Columns 3-13
Custom Filtering 3-13
Bypassing Rows for Indexing 3-13
Document Character Set 3-13
Character Set Detection 3-14
Language Detection 3-14
Mixed Character Set Columns 3-14
Document Language 3-14
Language Features Outside BASIC_LEXER 3-14
Indexing Multi-language Columns 3-15
Indexing Special Characters 3-15
Printjoin Characters 3-15
Skipjoin Characters 3-15
Other Characters 3-16
Case-Sensitive Indexing and Querying 3-16
Language-Specific Features 3-16
Automatic Language Detection with AUTO_LEXER 3-16
Indexing Themes 3-16
Base-Letter Conversion for Characters with Diacritical Marks 3-17
Alternate Spelling 3-17
Composite Words 3-17
Korean, Japanese, and Chinese Indexing 3-18

Fuzzy Matching and Stemming 3-18
Better Wildcard Query Performance 3-18
Document Section Searching 3-19
Stopwords and Stopthemes 3-19
Automatic Language Detection and Stoplists 3-19
Multi-Language Stoplists 3-19
Index Performance 3-19
Query Performance and Storage of LOB Columns 3-20
Mixed Query Performance 3-20
Creating Oracle Text Indexes 3-20
Summary of Procedure for Creating a Text Index 3-20
Creating Preferences 3-21
Datastore Examples 3-21
NULL_FILTER Example: Indexing HTML Documents 3-22
PROCEDURE_FILTER Example 3-22
BASIC_LEXER Example: Setting Printjoin Characters 3-23
MULTI_LEXER Example: Indexing a Multi-Language Table 3-23
BASIC_WORDLIST Example: Enabling Substring and Prefix Indexing 3-24
Creating Section Groups for Section Searching 3-24
Example: Creating HTML Sections 3-24
Using Stopwords and Stoplists 3-24
Multi-Language Stoplists 3-25
Stopthemes and Stopclasses 3-25
PL/SQL Procedures for Managing Stoplists 3-25
Creating a CONTEXT Index 3-25
CONTEXT Index and DML 3-26
Default CONTEXT Index Example 3-26
Incrementally Creating an Index with ALTER INDEX and CREATE INDEX 3-26
Creating a CONTEXT Index Incrementally with POPULATE_PENDING 3-27
Custom CONTEXT Index Example: Indexing HTML Documents 3-27
CONTEXT Index Example: Query Processing with FILTER BY and ORDER BY 3-27
Creating a CTXCAT Index 3-28
CTXCAT Index and DML 3-28
About CTXCAT Sub-Indexes and Their Costs 3-28
Creating CTXCAT Sub-indexes 3-28
Creating CTXCAT Index 3-30
Creating a CTXRULE Index 3-30
Step One: Create a Table of Queries 3-31
Step Two: Create the CTXRULE Index 3-31
Step Three: Classify a Document 3-31
Maintaining Oracle Text Indexes 3-32
Viewing Index Errors 3-32
Dropping an Index 3-32
Resuming Failed Index 3-33
Re-creating an Index 3-33
Re-creating a Global Index 3-33
Rebuilding an Index 3-35

Dropping a Preference 3-35
Managing DML Operations for a CONTEXT Index 3-36
Viewing Pending DML 3-36
Synchronizing the Index 3-36
Example 3-36
Maxtime Parameter for SYNC_INDEX 3-37
Locking Parameter for SYNC_INDEX 3-37
Optimizing the Index 3-37
CONTEXT Index Structure 3-38
Index Fragmentation 3-38
Document Invalidation and Garbage Collection 3-38
Single Token Optimization 3-38
Viewing Index Fragmentation and Garbage Data 3-38
Examples: Optimizing the Index 3-39

4 Querying with Oracle Text

Overview of Queries 4-1
Querying with CONTAINS 4-1
CONTAINS SQL Example 4-1
CONTAINS PL/SQL Example 4-2
Structured Query with CONTAINS 4-2
Querying with CATSEARCH 4-3
CATSEARCH SQL Query 4-3
CATSEARCH Example 4-3
Querying with MATCHES 4-4
MATCHES SQL Query 4-4
MATCHES PL/SQL Example 4-6
Word and Phrase Queries 4-7
CONTAINS Phrase Queries 4-7
CATSEARCH Phrase Queries 4-7
Querying Stopwords 4-8
ABOUT Queries and Themes 4-8
Querying Stopthemes 4-8
Query Expressions 4-9
CONTAINS Operators 4-9
CATSEARCH Operator 4-9
MATCHES Operator 4-9
Case-Sensitive Searching 4-9
Word Queries 4-10
ABOUT Queries 4-10
Query Feedback 4-10
Query Explain Plan 4-10
Using a Thesaurus in Queries 4-11
Document Section Searching 4-11
Using Query Templates 4-11
Query Rewrite 4-11
Query Relaxation 4-12

Query Language 4-13
Alternative and User-defined Scoring 4-13
Alternative Grammar 4-14
Query Analysis 4-14
Other Query Features 4-14
The CONTEXT Grammar 4-15
ABOUT Query 4-16
Logical Operators 4-16
Section Searching 4-17
Proximity Queries with NEAR and NEAR_ACCUM Operators 4-17
Fuzzy, Stem, Soundex, Wildcard and Thesaurus Expansion Operators 4-18
Using CTXCAT Grammar 4-18
Stored Query Expressions 4-18
Defining a Stored Query Expression 4-18
SQE Example 4-19
Calling PL/SQL Functions in CONTAINS 4-19
Optimizing for Response Time 4-19
Other Factors that Influence Query Response Time 4-19
Counting Hits 4-20
SQL Count Hits Example 4-20
Counting Hits with a Structured Predicate 4-20
PL/SQL Count Hits Example 4-20
Using DEFINESCORE and DEFINEMERGE for User-defined Scoring 4-21
The CTXCAT Grammar 4-21
Using CONTEXT Grammar with CATSEARCH 4-21

5 Presenting Documents in Oracle Text

Highlighting Query Terms 5-1
Text highlighting 5-1
Theme Highlighting 5-1
CTX_DOC Highlighting Procedures 5-1
Markup Procedure 5-2
Highlight Procedure 5-3
Concordance 5-3
Obtaining Lists of Themes, Gists, and Theme Summaries 5-3
Lists of Themes 5-4
In-Memory Themes 5-4
Result Table Themes 5-4
Gist and Theme Summary 5-4
In-Memory Gist 5-5
Result Table Gists 5-5
Theme Summary 5-5
Document Presentation and Highlighting 5-6
Highlighting Example 5-7
Document List of Themes Example 5-8
Gist Example 5-9

6 Classifying Documents in Oracle Text

Overview of Document Classification 6-1
Classification Applications 6-1
Classification Solutions 6-2
Rule-Based Classification 6-3
Rule-based Classification Example 6-3
CTXRULE Parameters and Limitations 6-6
Supervised Classification 6-7
Decision Tree Supervised Classification 6-7
Decision Tree Supervised Classification Example 6-8
SVM-Based Supervised Classification 6-10
SVM-Based Supervised Classification Example 6-11
Unsupervised Classification (Clustering) 6-12
Clustering Example 6-13

7 Tuning Oracle Text

Optimizing Queries with Statistics 7-1
Collecting Statistics 7-2
Example 7-3
Re-Collecting Statistics 7-3
Deleting Statistics 7-3
Optimizing Queries for Response Time 7-4
Other Factors that Influence Query Response Time 7-4
Improved Response Time with FIRST_ROWS(n) Hint for ORDER BY Queries 7-4
About the DOMAIN_INDEX_SORT Hint 7-5
Improved Response Time using Local Partitioned CONTEXT Index 7-5
Range Search on Partition Key Column 7-6
ORDER BY Partition Key Column 7-6
Improved Response Time with Local Partitioned Index for Order by Score 7-6
Optimizing Queries for Throughput 7-7
CHOOSE and ALL ROWS Modes 7-7
FIRST_ROWS(n) Mode 7-7
Composite Domain Index (CDI) in Oracle Text 7-7
Performance Tuning with CDI 7-8
Solving Index and Query Bottlenecks Using Tracing 7-8
Using Parallel Queries 7-9
Parallel Queries on a Local Context Index 7-9
Parallelizing Queries Across Oracle RAC Nodes 7-10
Tuning Queries with Blocking Operations 7-10
Frequently Asked Questions About Query Performance 7-11
What is *Query Performance*? 7-11
What is the fastest type of text query? 7-11
Should I collect statistics on my tables? 7-12
How does the size of my data affect queries? 7-12
How does the format of my data affect queries? 7-12
What is a *functional* versus an *indexed* lookup? 7-12
What tables are involved in queries? 7-12

Does sorting the results slow a text-only query? 7-13
How do I make an ORDER BY score query faster? 7-13
Which Memory Settings Affect Querying? 7-13
Does out of line LOB storage of wide base table columns improve performance? 7-13
How can I make a CONTAINS query on more than one column faster? 7-14
Is it OK to have many expansions in a query? 7-14
How can local partition indexes help? 7-15
Should I query in parallel? 7-15
Should I index themes? 7-15
When should I use a CTXCAT index? 7-16
When is a CTXCAT index NOT suitable? 7-16
What optimizer hints are available, and what do they do? 7-16
Frequently Asked Questions About Indexing Performance 7-16
How long should indexing take? 7-16
Which index memory settings should I use? 7-17
How much disk overhead will indexing require? 7-17
How does the format of my data affect indexing? 7-18
Can parallel indexing improve performance? 7-18
How can I improve index performance for creating local partitioned index? 7-18
How can I tell how much indexing has completed? 7-19
Frequently Asked Questions About Updating the Index 7-19
How often should I index new or updated records? 7-19
How can I tell when my indexes are getting fragmented? 7-19
Does memory allocation affect index synchronization? 7-20

8 Searching Document Sections in Oracle Text

About Oracle Text Document Section Searching 8-1
Enabling Oracle Text Section Searching 8-1
Create a Section Group 8-1
Define Your Sections 8-3
Index Your Documents 8-3
Section Searching with the WITHIN Operator 8-3
Path Searching with INPATH and HASPATH Operators 8-3
Oracle Text Section Types 8-4
Zone Section 8-4
Field Section 8-6
Stop Section 8-7
MDATA Section 8-7
NDATA Section 8-9
SDATA Section 8-10
Attribute Section 8-11
Special Sections 8-11
HTML Section Searching with Oracle Text 8-12
Creating HTML Sections 8-12
Searching HTML Meta Tags 8-12
Example: Creating Sections for `<META>`Tags 8-12
XML Section Searching with Oracle Text 8-13

Automatic Sectioning 8-13
Attribute Searching 8-13
Creating Attribute Sections 8-14
Searching Attributes with the INPATH Operator 8-14
Creating Document Type Sensitive Sections 8-14
Path Section Searching 8-15
Creating an Index with PATH_SECTION_GROUP 8-15
Top-Level Tag Searching 8-15
Any-Level Tag Searching 8-16
Direct Parentage Searching 8-16
Tag Value Testing 8-16
Attribute Searching 8-16
Attribute Value Testing 8-16
Path Testing 8-17
Section Equality Testing with HASPATH 8-17

9 Using Oracle Text Name Search

Overview of Name Search 9-1
Examples of Using Name Search 9-1

10 Working With a Thesaurus in Oracle Text

Overview of Oracle Text Thesaurus Features 10-1
Oracle Text Thesaurus Creation and Maintenance 10-1
CTX_THES Package 10-1
Thesaurus Operators 10-1
ctxload Utility 10-2
Using a Case-sensitive Thesaurus 10-2
Using a Case-insensitive Thesaurus 10-2
Default Thesaurus 10-3
Supplied Thesaurus 10-3
Supplied Thesaurus Structure and Content 10-3
Supplied Thesaurus Location 10-3
Defining Terms in a Thesaurus 10-4
Defining Synonyms 10-4
Defining Hierarchical Relations 10-4
Using a Thesaurus in a Query Application 10-4
Loading a Custom Thesaurus and Issuing Thesaurus-based Queries 10-5
Advantage 10-5
Limitations 10-5
Augmenting Knowledge Base with Custom Thesaurus 10-5
Advantage 10-6
Limitations 10-6
Linking New Terms to Existing Terms 10-6
Loading a Thesaurus with ctxload 10-7
Compiling a Loaded Thesaurus 10-7
About the Supplied Knowledge Base 10-7
Adding a Language-Specific Knowledge Base 10-8

Limitations 10-8

11 Using XML Query Result Set Interface

Overview of the XML Query Result Set Interface 11-1
Using the XML Query Result Set Interface 11-1

12 Extracting Entities Using Oracle Text

How to Use Entity Extraction 12-1
Examples of Entity Extraction 12-2

13 Administering Oracle Text

Oracle Text Users and Roles 13-1
CTXSYS User 13-1
CTXAPP Role 13-1
Granting Roles and Privileges to Users 13-2
DML Queue 13-2
The CTX_OUTPUT Package 13-2
The CTX_REPORT Package 13-2
Text Manager in Oracle Enterprise Manager 13-5
Using Text Manager 13-6
Viewing General Information for a Text Index 13-6
Checking Text Index Health 13-6
Servers and Indexing 13-7
Database Feature Usage Tracking in Oracle Enterprise Manager 13-7
Package Usage Statistics 13-7
Index Usage Statistics 13-8
SQL Operator Usage Statistics 13-8
Oracle Text on Oracle Real Application Clusters 13-8

14 Migrating Oracle Text Applications

Migrating to Oracle Text 11*g* Release 2 (11.2) 14-1
Migrating to Oracle Text 11*g* Release 1 (11.1) 14-1
Migrating to Oracle Text 10*g* Release 2 (10.2) 14-1
New Filter (INSO_FILTER versus AUTO_FILTER) 14-1
Migrating to the AUTO_FILTER Filter Type 14-2

A CONTEXT Query Application

Web Query Application Overview A-1
The PSP Web Application A-3
Web Application Prerequisites A-3
Building the Web Application A-3
PSP Sample Code A-5
loader.ctl A-5
loader.dat A-5
search_htmlservices.sql A-5

search_html.psp A-7
The JSP Web Application A-8
Web Application Prerequisites A-9
JSP Sample Code A-9
search_html.jsp A-9

B CATSEARCH Query Application

CATSEARCH Web Query Application Overview B-1
The JSP Web Application B-1
Building the JSP Web Application B-1
JSP Sample Code B-3
loader.ctl B-3
loader.dat B-4
catalogSearch.jsp B-4

Glossary

Index

Preface

This Preface contains these topics:

- Audience
- Documentation Accessibility
- Related Documents
- Conventions

Audience

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

- maintain an Oracle Text system

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

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible to all users, including users that are disabled. 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. Accessibility 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 more information, visit the Oracle Accessibility Program Web site at `http://www.oracle.com/accessibility/`.

Accessibility of Code Examples in Documentation

Screen readers 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, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

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

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/support/contact.html or visit http://www.oracle.com/accessibility/support.html if you are hearing impaired.

Related Documents

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 Language Reference*
- *Oracle Database Reference*
- *Oracle Database Advanced Application Developer's Guide*

For more information about PL/SQL, refer to:

- *Oracle Database PL/SQL Language Reference*

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Understanding Oracle Text Application Development

This chapter discusses the following topics:

- Introduction to Oracle Text
- Document Collection Applications
- Catalog Information Applications
- Document Classification Applications
- XML Search Applications

Introduction to Oracle Text

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

To design an Oracle Text application, first determine the type of queries you expect to run. This enables you to choose the most suitable index for the task.

Consider the following three categories of applications for Oracle Text:

- Document Collection Applications
- Catalog Information Applications
- Document Classification Applications

Document Collection Applications

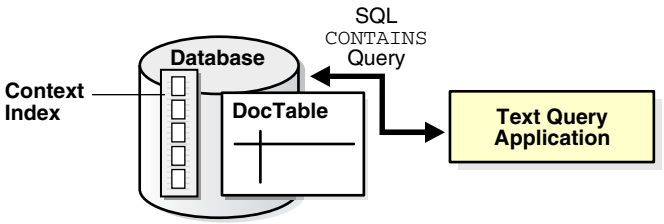
A text query application enables users to search *document collections*, such as Web sites, digital libraries, or document warehouses. The collection is typically static with no significant change in content after the initial indexing run. Documents can be of any size and of different formats, such as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

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 can be used to improve the search results, such as stemming, proximity searching, and wildcarding.

An important factor for this type of application is retrieving documents relevant to a 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, the 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

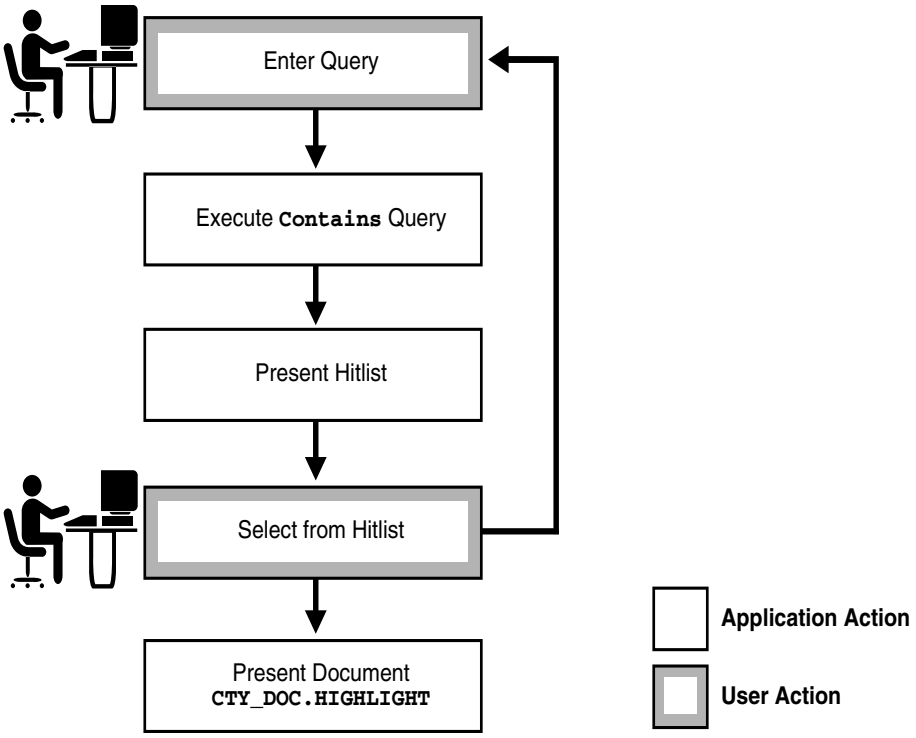
A typical text query application on a document collection enables the user to enter a query. The application enters a `CONTAINS` query and returns a list, called a *hitlist*, of documents that satisfy the query. The results are usually ranked by relevance. The application enables the user to view one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

1. The user enters a query.
2. The application runs 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



Catalog Information Applications

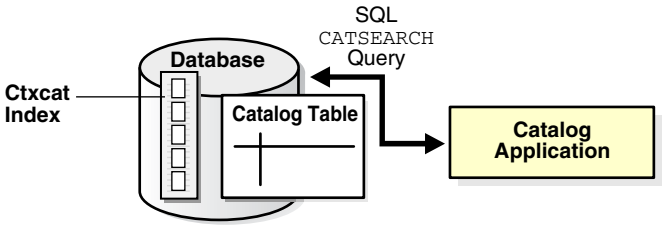
Catalog information consists of inventory type information, such as that of an online book store or auction site. The stored information consists of text information, such as book titles, and related structured information, such as price. The information is usually updated regularly to keep the online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

Queries are usually a combination of a text component and a structured component. 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. 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 that uses the `CATSEARCH` operator to query the index.

Figure 1–3 A Catalog Query Application



Flowchart for Catalog Query Application

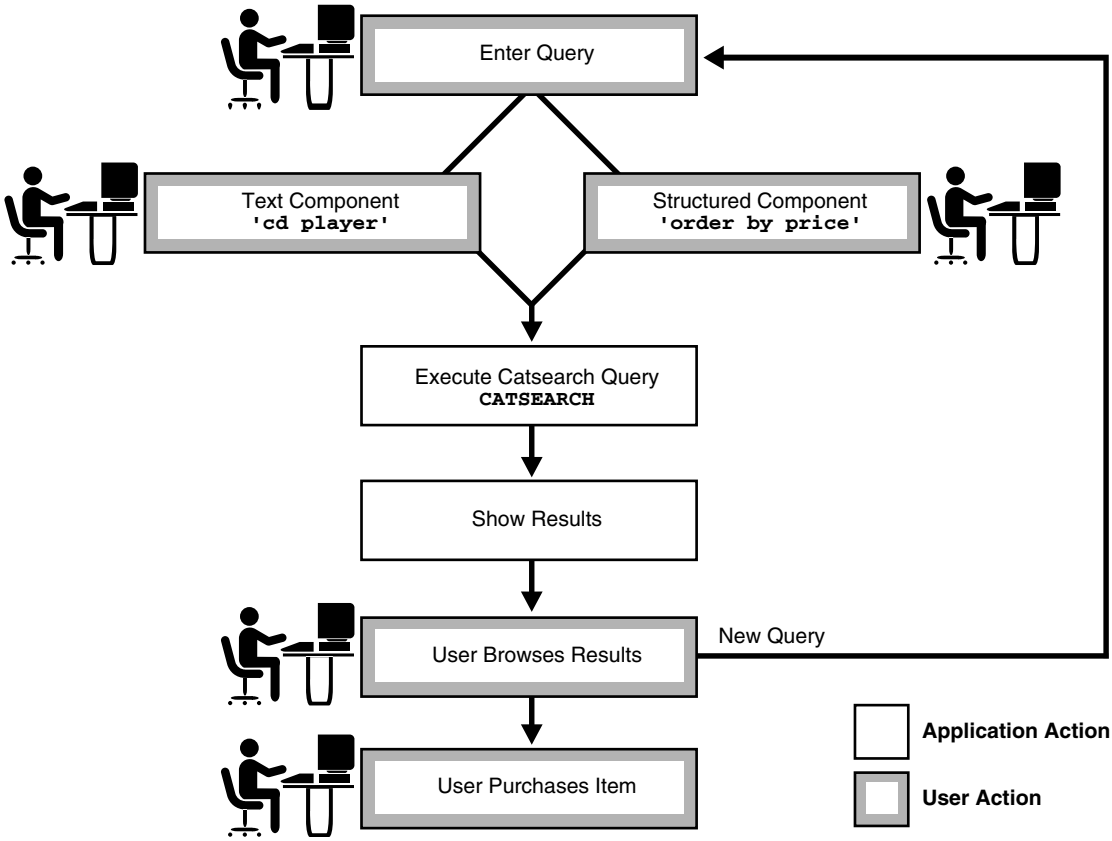
A catalog application enables users to search for specific items in catalogs. For example, an online store application enables users to search for and purchase items in

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 enters another query or performs an action, such as purchasing the item.

Figure 1–4 Flowchart of a Catalog Query Application



Document Classification Applications

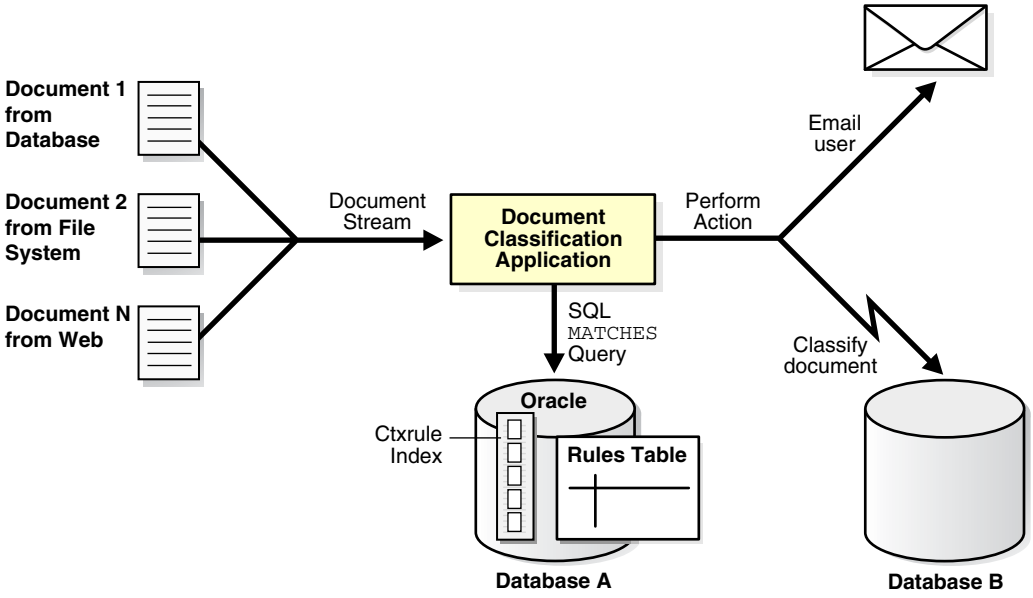
In a document classification application, an incoming stream or a set of documents is compared to a pre-defined set of rules. When a document matches one or more rules, the application performs some action.

For example, assume there is an incoming stream of news articles. You 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 e-mailing one or more users.

To create a document classification application, 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. See Figure 1–5 for the general flow of a classification application.

Figure 1–5 Overview of a Document Classification Application



XML Search Applications

An XML search application performs searches over XML documents. A regular document search usually searches across a set of documents to return documents that satisfy a text predicate; an XML search often uses 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, then you can also benefit from the following characteristics of Oracle Text searches:

- Searches are token-based, whitespace-normalized
- Hitlists are ranked by relevance

- Case-sensitive searching
- Section searching
- Linguistic features such as stemming and fuzzy searching
- Performance-optimized queries for large document sets

See Also: "XML Section Searching with Oracle Text" on page 8-13

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:

- It is a non-text search with equality and range on dates and numbers.
- It is a string search that is character-based, where all characters are treated the same.
- It 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:

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

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

See Also: *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 where you want to do a full-text retrieval, leveraging the XML structure by entering queries such as "find all nodes that contain the word Pentium." Do so in one of two ways:

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

See Also: *Oracle XML DB Developer's Guide* and "XML Section Searching with Oracle Text" on page 8-13

Using the Text-on-XML Method

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

With the Text-on-XML method, 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, 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, use the `WITHIN` operator:

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

Specify more complex criteria with `XPATH` expressions using the `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
- Creating an Oracle Text User
- Query Application Quick Tour
- Catalog Application Quick Tour
- Classification Application Quick Tour

Overview of Getting Started with Oracle Text

This chapter provides basic information about how to create an Oracle Text developer user account, and how to build simple text query and catalog applications. It also provides information about basic SQL statements for each type of application to load, index, and query tables.

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

> **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, do the following as the system administrator user:

Step 1 Create User

The following SQL statement 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 statement 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

Oracle Text includes several packages that let you perform actions ranging from synchronizing an Oracle Text index to highlighting documents. For example, the CTX_DDL package includes the SYNC_INDEX procedure, which enables you to synchronize your index. The *Oracle Text Reference* describes each of these packages in its own chapter.

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, enter the following SQL statements:

```
GRANT EXECUTE ON CTXSYS.CTX_CLS TO myuser;
GRANT EXECUTE ON CTXSYS.CTX_DDL TO myuser;
GRANT EXECUTE ON CTXSYS.CTX_DOC TO myuser;
GRANT EXECUTE ON CTXSYS.CTX_OUTPUT TO myuser;
GRANT EXECUTE ON CTXSYS.CTX_QUERY TO myuser;
GRANT EXECUTE ON CTXSYS.CTX_REPORT TO myuser;
GRANT EXECUTE ON CTXSYS.CTX_THES TO myuser;
GRANT EXECUTE ON CTXSYS.CTX_ULEXER 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 to load the text table, index the documents, and query the 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

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" on page A-3 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. Because you are indexing HTML, this example uses the NULL_FILTER preference type for no filtering and 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, then use the CTXSYS.AUTO_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

First, set the format of the SELECT statement's output so it is easily readable. Set the width of the text column to 40 characters:

```
COLUMN text FORMAT a40;
```

Now query the table with the SELECT statement with CONTAINS. This retrieves the document IDs that satisfy the query. 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>');
```

Because 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 rerun 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 add-on 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 Oracle Technology Network.

Oracle JDeveloper

Obtain the latest JDeveloper software from the following URL:

http://www.oracle.com/technology/software/products/jdev

See "Building the JSP Web Application" on page B-1 for an example.

Oracle Text Wizard Addins

Obtain the Text, Catalog, and Classification Wizard addins from the following URL:

http://www.oracle.com/technology/software/products/text

Oracle Text Wizard Instructions

Find instructions on using the Oracle Text Wizard and setting up your JSP files to run in a Web server environment from the following URL:

http://www.oracle.com/technology/software/products/text

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 to create a suitable `CTXCAT` index. Queries on this type of index are entered 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

Connect as the `CTXAPP` role the user `myuser`:

```
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

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" on page A-3 for an example on how to use SQL*Loader to load a text table from a data file

Step 1 Determine your Queries

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, because 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.

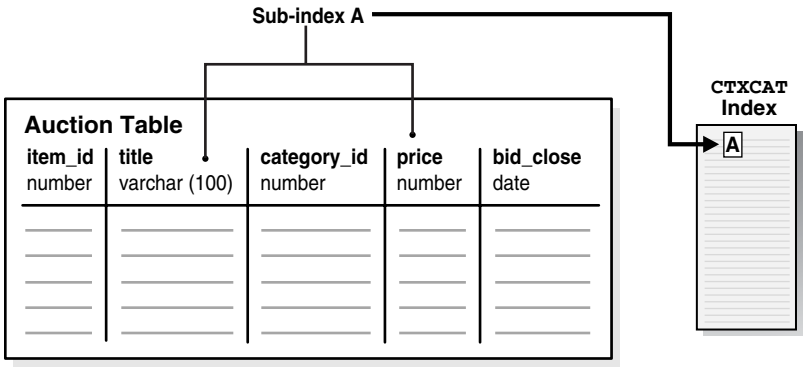
Step 3 Create the CTXCAT Index

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 run 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

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 rerun 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.

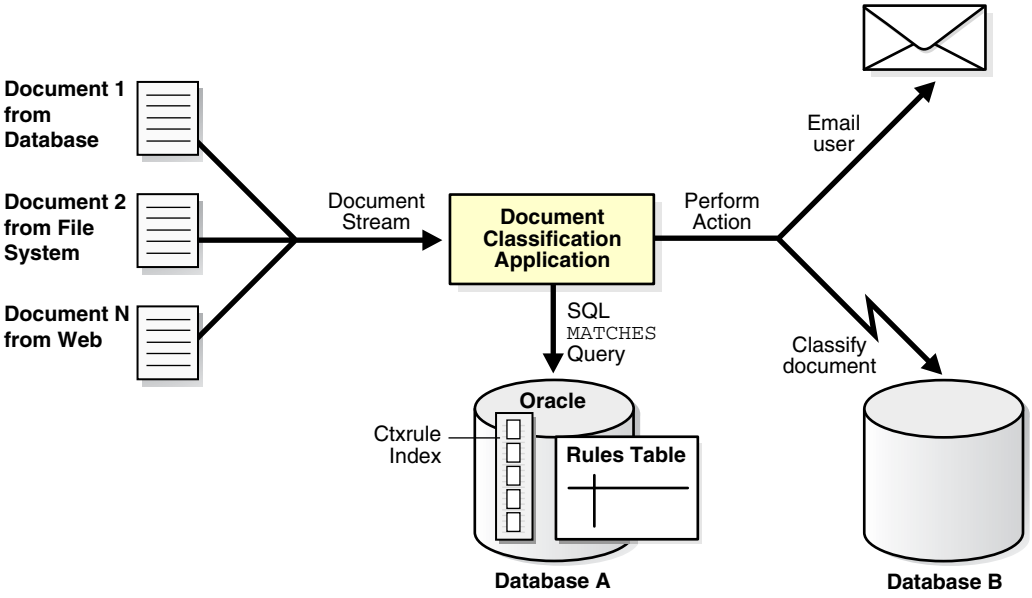
Documents are classified according to predefined rules. These rules select for a category. For instance, a query rule of *'presidential elections'* might select documents for a category about politics.

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 "Overview of Document Classification" on page 6-1.)

To create simple classification for document content using Oracle Text, create rules. Rules are essentially a table of queries that categorize document content. 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. See 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 shows how to classify documents by defining simple categories, creating a `CTXRULE` index, and using `MATCHES`, using the `CTXAPP` role user `myuser`.

Step 1 Connect As the Appropriate User

Connect as the `CTXAPP` role user `myuser`:

```
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 e-mailing a user.

> **See Also:** Chapter 6, "Classifying Documents in Oracle Text" for more extended classification examples

3

Indexing with Oracle Text

This chapter provides an introduction to Oracle Text indexing. The following topics are discussed:

- About Oracle Text Indexes
- Considerations For Indexing
- Creating Oracle Text Indexes
- Maintaining Oracle Text Indexes
- Managing DML Operations for a CONTEXT Index

About Oracle Text Indexes

This section discusses the different types of Oracle Text indexes, their structure, the indexing process, and limitations. The following subjects are included:

- Types of Oracle Text Indexes
- Structure of the Oracle Text CONTEXT Index
- The Oracle Text Indexing Process
- Partitioned Tables and Indexes
- Creating an Index Online
- Parallel Indexing
- Indexing and Views

Types of Oracle Text Indexes

With Oracle Text, you can create indexes of several types, using `CREATE INDEX`. Table 3–1 describes each index type, its purpose, and the features it supports.

Table 3–1 Oracle Text Index Types

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. You can customize the 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. Supports FILTER BY and ORDER BY clauses of CREATE INDEX to also index structured column values for more efficient processing of mixed queries.

Table 3–1 (Cont.) Oracle Text Index Types

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, the number of indexes in the index set, and the 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. This index is created on a table of queries, where the queries define the classification or routing criteria.	See "CTXRULE Parameters and Limitations" on page 6-6.	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.

Table 3–1 (Cont.) Oracle Text Index Types

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()	This indextype is deprecated and is only supported for backward compatibility with older releases of Oracle Database where XMLIndex is not available. If you are building a new application, Oracle recommends that you use XMLIndex. 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 Search Applications" on page 1-5.

See Also: "Creating Oracle Text Indexes" on page 3-20

An Oracle Text index is an Oracle Database domain index. To build your query application, you can create an index of type CONTEXT with a mixture of text and structured data columns, 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 CONTEXT index with FILTER BY and/or ORDER BY clauses to specify relational columns that will be used in structured criteria of the mixed query. Query this index with the CONTAINS operator.

Use the CTXCAT 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. Query this index with the CATSEARCH operator.

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

If you are working with XMLtype columns, then create an XMLIndex index to speed up queries with existsNode.

> **Note:** The CTXXPATH index type is deprecated. It is only supported for backward compatibility with older releases of Oracle Database where XMLIndex is not available. If you are building a new application, then Oracle recommends that you use XMLIndex.

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.

> **Note:** Because a transparent data encryption-enabled column does not support domain indexes, it cannot be used with Oracle Text. However, you can create an Oracle Text index on a column in a table stored in TDE-enabled tablespace.

See Also: "Creating Oracle Text Indexes" on page 3-20

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.

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: "Creating Preferences" on page 3-21 to learn more about indexing theme information

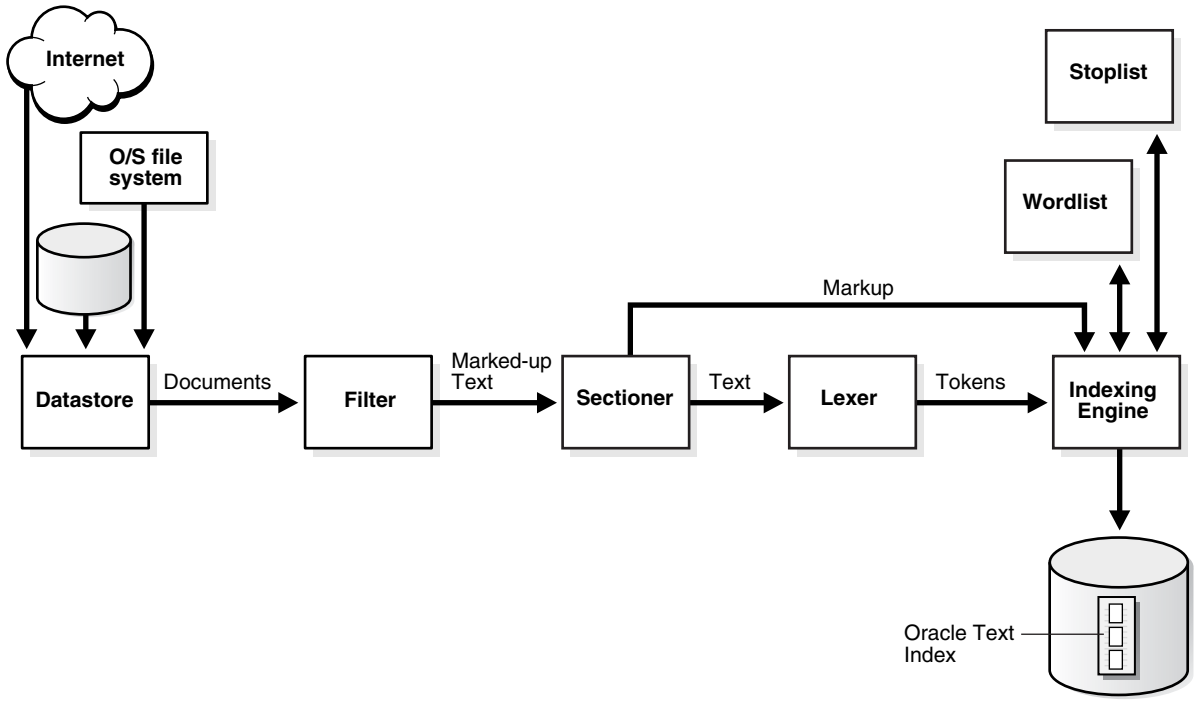
The Oracle Text Indexing Process

This section describes the Oracle Text indexing process. Initiate the indexing process with the CREATE INDEX statement. Initiate the indexing process using the CREATE INDEX statement to create an Oracle Text index of tokens, organized 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 following sections 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`, then the stream starts by reading the files from the operating system. You can also store your documents on the Internet or in Oracle Database. Wherever your files reside physically, you must always have a text table in 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 `AUTO_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

You create a lexer preference using one of the Oracle Text lexer types to specify the language of the text to be indexed. 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 analyzes 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, because querying, DML, and backup and recovery can act on a single partition.

On local `CONTEXT` indexes with multiple table sets, Oracle Text supports the number of partitions supported by Oracle Database.

Note: The number of partitions supported in Oracle Text is approximately 1024K-1. This limit, which should be more than adequate, is not applicable to a `CONTEXT` index on partitioned tables.

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

Because 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.

Because parallel indexing decreases the *initial* indexing time, it is useful for the following scenarios:

- 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:**
>
> - "Parallel Queries on a Local Context Index" on page 7-9
> - "Frequently Asked Questions About Indexing Performance" on page 7-16

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

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. This sections contains the following topics:

- Location of Text
- Document Formats and Filtering
- Bypassing Rows for Indexing
- Document Character Set
- Document Language
- Indexing Special Characters
- Case-Sensitive Indexing and Querying
- Language-Specific Features
- Fuzzy Matching and Stemming
- Better Wildcard Query Performance
- Document Section Searching
- Stopwords and Stopthemes
- Index Performance
- Query Performance and Storage of LOB Columns
- Mixed Query Performance

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.

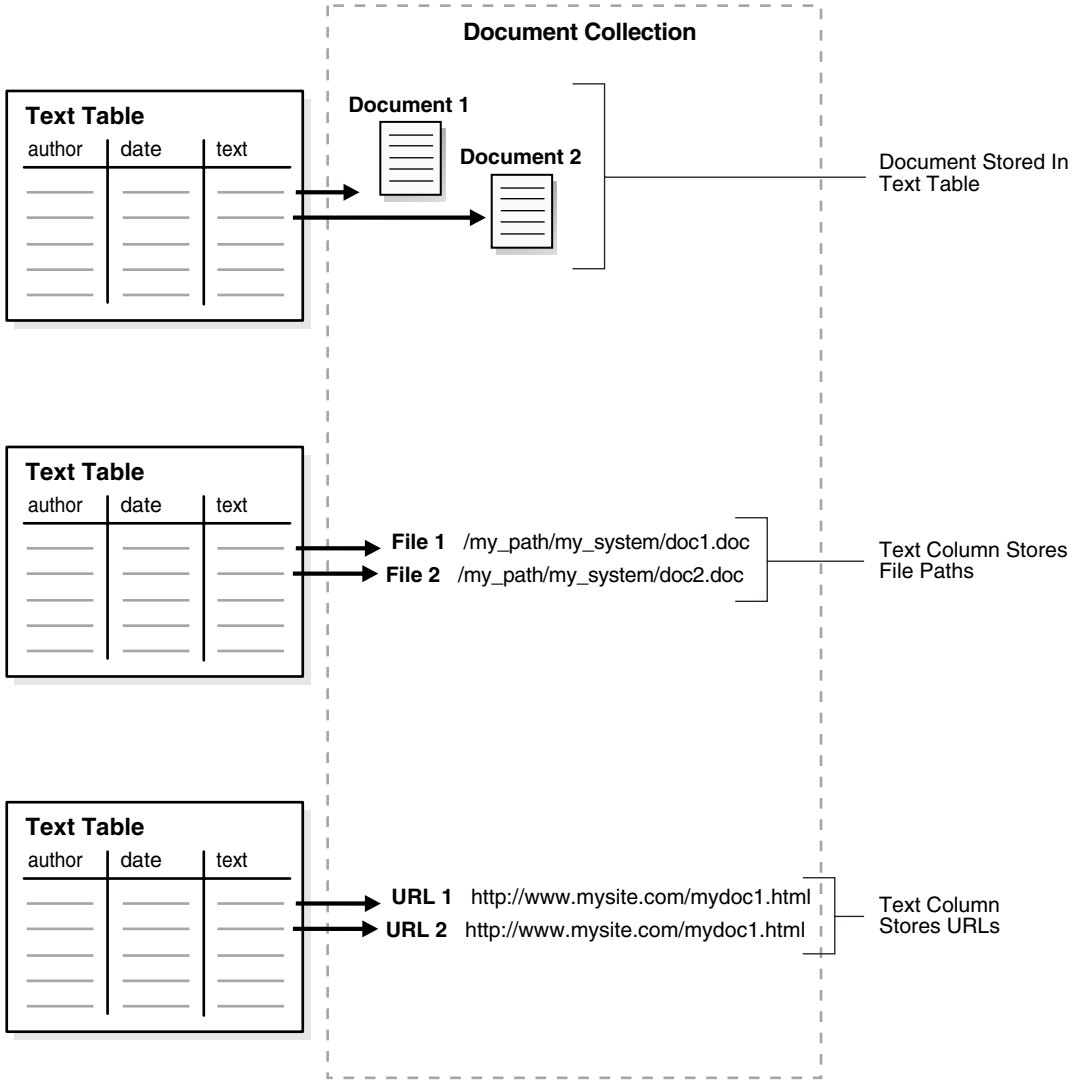
When you create a CONTEXT index, you can populate rows in your text table with one of the following elements:

- 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 for each row.

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). If you mix HTML and XML documents in 1 index, you might not be able to configure your index to your needs; you cannot prevent stylesheet information from being added to the index.

> **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.

Table 3–2 summarizes the different ways you can store your text with the datastore preference type.

Table 3–2 Summary of DATASTORE Types

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. 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, because the system must retrieve the document from the network.

> **See Also:** "Datastore Examples" on page 3-21

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 `AUTO_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 `AUTO_FILTER` type. For best results, use the `NULL_FILTER` preference type.

> **See Also:** "NULL_FILTER Example: Indexing HTML Documents" on page 3-22

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.

A third format column type, `IGNORE`, is provided for when you do not want a document to be indexed at all. This is useful, for example, when you have a mixed-format table that includes plain-text documents in both Japanese and English, but you only want to process the English documents; another example might be that of a mixed-format table that includes both plain-text documents and images. Because `IGNORE` is implemented at the datastore level, it can be used with all filters.

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 3-22

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 `AUTO_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, then you can use the `AUTO_FILTER` or `CHARSET_FILTER` filter type to convert your text for indexing.

Character Set Detection

When the CHARSET column is set to AUTO, the AUTO_FILTER filter detects the character set of the document and converts it from the detected character set to the database character set, if there is a difference.

Language Detection

The lexer preference AUTO_LEXER performs language detection and any necessary transformations. At index time, the lexer automatically detects the language of the document and tokenizes and stems the document appropriately. At query time, the language of the query is inherited from the query template, or if no language is specified through the query template, then the language of the query is inherited from the session language.

> **See Also:** *Oracle Text Reference* for more information about AUTO_LEXER

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 for each row. 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. Depending on the language of your documents, use one of the following lexer types:

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.

Use the AUTO_LEXER preference type to index columns that contain documents of different languages.

Use the MULTI_LEXER preference type for indexing tables containing documents of different languages such as English, German, and Japanese.

Use the USER_LEXER preference type to create your own lexer for indexing a particular language.

Use the WORLD_LEXER preference type for indexing tables containing documents of different languages and to autodetect the languages in the document.

You can also use other lexer types that are designed specifically to tokenize and index Japanese, Chinese, and Korean.

> **See Also:** *Oracle Text Reference* to learn more about indexing languages and lexer types

Language 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 the user-defined 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" on page 3-23

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 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:

- Printjoin Characters
- Skipjoin Characters
- Other Characters

> **Note:** A new lexer type `AUTO_LEXER` automatically detects the language of the document and tokenizes the document appropriately.

Printjoin Characters

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 Printjoin Characters" on page 3-23

Skipjoin Characters

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:

- Automatic Language Detection with AUTO_LEXER
- Indexing Themes
- Base-Letter Conversion for Characters with Diacritical Marks
- Alternate Spelling
- Composite Words
- Korean, Japanese, and Chinese Indexing

Automatic Language Detection with AUTO_LEXER

The `AUTO_LEXER` lexer type automatically detects the language of the document. At index creation, `AUTO_LEXER` automatically detects the language of the document and tokenizes and stems the document appropriately.

> **See Also:** *Oracle Text Reference* for more information about `AUTO_LEXER`

Indexing Themes

For English and French, you can index document *theme* information. A document theme 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" on page 4-8

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

Index these languages with specific lexers:

Table 3–3 Lexers for Asian Languages

Language	Lexer
Korean	`KOREAN_MORPH_LEXER`
Japanese	`JAPANESE_LEXER`, `JAPANESE_VGRAM_LEXER`
Chinese	`CHINESE_LEXER`, `CHINESE_VGRAM_LEXER`

These lexers have their own sets of attributes to control 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 automatically detect the language of a document and to have the necessary transformations performed, create a stem index by enabling the index_stems attribute of the `AUTO_LEXER`. The stemmer that corresponds to the document language will be used and the stemmer will always be configured to maximize document recall. Additionally, for documents in languages that use compound words such as Finnish, Swedish, and Dutch, if `index_stems` is set to `YES`, then compound word stemming will automatically be performed. Compounds are always separated into their components stems.

To improve the performance of stem queries, 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 enter 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" on page 3-24

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, "Searching Document Sections in Oracle Text"

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.

Automatic Language Detection and Stoplists

At index time, the lexer preference `AUTO_LEXER` automatically detects the language of the document and tokenizes and stems the document appropriately. At query time, the language of the query is inherited from the query template, or from the session language (if no language is specified through the query template).

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 index creation, 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" on page 7-16

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.

Note: You cannot map attributes to a remote LOB column.

See Also: "Does out of line LOB storage of wide base table columns improve performance?" on page 7-13

Mixed Query Performance

If your `CONTAINS()` query also has structured predicates on the non-text columns, then you may consider having the Text index also index those column values. To do so, specify those columns in the `FILTER BY` clause of the `CREATE INDEX` statement. This enables Oracle Text to determine whether to have the structured predicate(s) also be processed by the Text index for better performance.

Additionally, if your `CONTAINS()` query has `ORDER BY` criteria on one or more structured columns, then the Text index can also index those column values. Specify those columns in the `ORDER BY` clause of the `CREATE INDEX` statement. This enables Oracle Text to determine whether to push the sort into the Text index for better query response time.

See Also: "CONTEXT Index Example: Query Processing with FILTER BY and ORDER BY" on page 3-27

Creating Oracle Text Indexes

You can create four types of indexes with Oracle Text: `CONTEXT`, `CTXCAT`, and `CTXRULE`, and `CTXXPATH`. The section contains these topics:

- Summary of Procedure for Creating a Text Index
- Creating Preferences
- Creating Section Groups for Section Searching
- Using Stopwords and Stoplists
- Creating a CONTEXT Index
- Creating a CTXCAT Index
- Creating a CTXRULE Index

Summary of Procedure for Creating a Text 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 statement 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: Default CONTEXT Index Example on page 3-26

To create an Oracle Text index:

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?

2. Optionally, create your own custom preferences, section groups, or stoplists. See "Creating Preferences" on page 3-21.
3. Create the Text index with the SQL statement `CREATE INDEX`, naming your index and optionally specifying preferences. See "Creating a CONTEXT Index" on page 3-25.

See Also: "Considerations For Indexing" on page 3-9 and CREATE INDEX topic in *Oracle Text Reference*

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 in HTML, then 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 Printjoin 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, enter 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');
```

Because 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, "Searching Document Sections in Oracle Text"

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 statements

Creating a CONTEXT Index

The CONTEXT index type is well suited for indexing large coherent documents in formats such as Microsoft 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 a record is deleted, the index change is immediate. That is, your own session will no longer find the record from the moment you make the change, and other users will not find the record as soon as you commit. For inserts and updates, the new information will not be visible to text searches until an index synchronization has occurred. Therefore, when you perform inserts or updates on the base table, you must explicitly synchronize the index with CTX_DDL.SYNC_INDEX.

See Also: "Synchronizing the Index" on page 3-36

Default CONTEXT Index Example

The following statement 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 actions by default for all languages:

- 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 AUTO_FILTER filter.

  To learn more about configuring your environment to use the AUTO_FILTER 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.

Incrementally Creating an Index with ALTER INDEX and CREATE INDEX

The ALTER INDEX and CREATE INDEX statements support incrementally creating a global CONTEXT index.

- For creating a global index, CREATE INDEX supports the NOPOPULATE keyword of the REBUILD clause. Using the NOPOPULATE keyword in the REPLACE parameter, you can create indexes incrementally. This is valuable for creating text indexes in large installations that cannot afford to have the indexing process running continuously.

- For creating a local index partition, `ALTER INDEX ... REBUILD partition ... parameters ('REPLACE ...')` parameter string is modified to support the `NOPOPULATE` keyword.
- For creating a partition on a local index, `CREATE INDEX ... LOCAL ... (partition ... parameters ('NOPOPULATE'))` is supported. The partition-level `POPULATE` or `NOPOPULATE` keywords override any `POPULATE` or `NOPOPULATE` specified at the index level.

> **See Also:** *Oracle Text Reference* to learn more about the syntax for the `ALTER INDEX` and `CREATE INDEX` statements

Creating a CONTEXT Index Incrementally with POPULATE_PENDING

For large installations that cannot afford to have the indexing process run continuously, you can use the `CTX_DDL.POPULATE_PENDING` procedure. This also provides finer control over creating the indexes. The preferred method is to create an empty index, place all the rowids into the pending queue, and build the index through `CTX_DDL.SYNC_INDEX`.

This procedure populates the pending queue with every rowid in the base table or table partition.

> **See Also:** *Oracle Text Reference* for information about `CTX.DDL.POPULATE_PENDING`

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" on page 3-21 for more examples on creating a custom `context` index

CONTEXT Index Example: Query Processing with FILTER BY and ORDER BY

To enable more efficient query processing and better response time for mixed queries, you can use `FILTER BY` and `ORDER BY` clauses as shown in the following example.

```
CREATE INDEX myindex on docs(text) INDEXTYPE is CTXSYS.CONTEXT
```

```
FILTER BY category, publisher, pub_date
ORDER BY pub_date desc;
```

In this example, by specifying the clause `FILTER BY category, publisher, pub_date` at query time, Oracle Text will also consider pushing any relational predicate on any of these columns into the Text index row source for more efficient query processing.

Also, when the query has matching `ORDER BY` criteria, by specifying `ORDER BY pub_date desc`, Oracle Text will determine whether to push the `SORT` into the Text index row source for better response time.

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 provide 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 in this section.

About CTXCAT Sub-Indexes and Their Costs

A `CTXCAT` index comprises 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 the following factors:

- 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, because 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 that 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

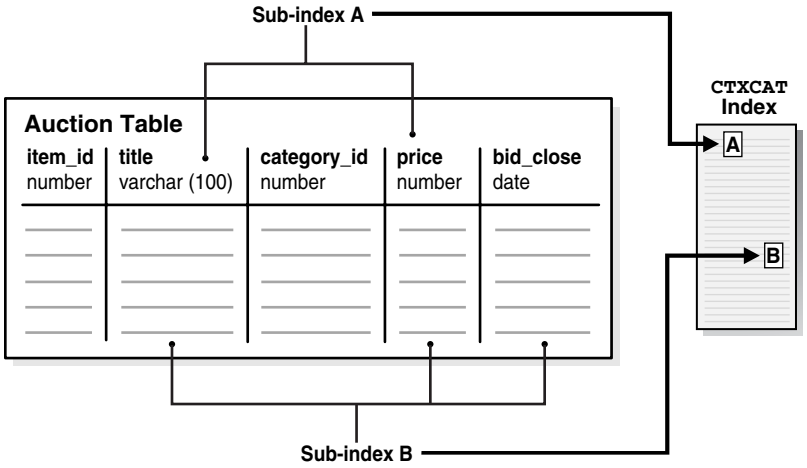


Figure 3–3 shows a table called `AUCTION` with the following schema:

```
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 enter. 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 3-29 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 `CREATEINDEX`

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. 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.

This section contains the following steps toward creating a CTXRULE index and a simple document classification application:

- Step One: Create a Table of Queries
- Step Two: Create the CTXRULE Index
- Step Three: Classify a Document

> **See Also:** Chapter 6, "Classifying Documents in Oracle Text" for more information on document classification and the CTXRULE index

Step One: 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 (or 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

Step Two: 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');
```

Step Three: Classify a Document

With a CTXRULE index created on a 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;
```

Maintaining Oracle Text Indexes

This section describes maintaining your index in the event of an error or indexing failure. The following topics are included:

- Viewing Index Errors
- Dropping an Index
- Resuming Failed Index
- Re-creating an Index
- Rebuilding an Index
- Dropping a Preference

Viewing Index Errors

Sometimes an indexing operation might fail or not complete successfully. When the system encounters an error during 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, enter the following statement:

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

To clear the view of errors, enter:

```
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.

Drop an index using the DROP INDEX statement in SQL.

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

For example, to drop an index called newsindex, enter the following SQL statement:

```
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 statement

Resuming Failed Index

You can sometimes resume a failed index creation operation using the ALTER INDEX statement. 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, then all optimization work up to the commit point has already been saved.

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

Example: Resuming a Failed Index

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

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

Re-creating an Index

This section describes the procedures available for re-creating an index, which enable index settings to be changed. During the re-creation process, the index can be queried normally.

Re-creating a Global Index

Oracle Text provides RECREATE_INDEX_ONLINE to re-create a CONTEXT index with new preferences, while preserving the base table DML and query capability during the re-create process. You can use RECREATE_INDEX_ONLINE in a one step procedure to re-create a CONTEXT index online for global indexes. Because the new index is created alongside the existing index, this operation requires storage roughly equal to the size of the existing index. Also, because the RECREATE_INDEX_ONLINE operation is performed online, you may issue DML on the base table during the operation. All DML that occurs during re-creation is logged into an online pending queue.

- After the re-create index operation is complete, any new information resulting from DML during the re-creation process may not be immediately reflected. As with creating an index online, the index should be synchronized after the re-create index operation is complete in order to bring it fully up-to-date.
- Synchronizations issued against the index during the re-creation are processed against the old, existing data. Synchronizations are blocked during this time when queries return errors.

- Optimize commands issued against the index during the re-creation return immediately without error and without processing.
- During RECREATE_INDEX_ONLINE, the index can be queried normally most of the time. Queries return results based on the existing index and policy until after the final swap. Also, if you issue DML statements and synchronize them, then you will be able to see the new rows when you query on the existing index.

See Also: *Oracle Text Reference* to learn more about the RECREATE_INDEX_ONLINE procedure

---

Note: Transactional query is not supported with RECREATE_INDEX_ONLINE.

Re-creating a Local Partitioned Index

If the index is locally partitioned, you cannot re-create the index in one step. You must first create a shadow policy, and then run the RECREATE_INDEX_ONLINE procedure for every partition. You can specify SWAP or NOSWAP, which indicates whether re-creating the index for the partition will swap the index partition data and index partition metadata.

This procedure can also be used to update the metadata (for example the storage preference) of each partition when you specify NOPOPULATE in the parameter string. This is useful for incremental building of a shadow index through time-limited synchronization. If NOPOPULATE is specified, then NOSWAP is silently enforced.

- When all of the partitions use NOSWAP, the storage requirement is approximately equal to the size of the existing index. During the re-creation of the index partition, since no swapping is performed, queries on the partition are processed normally. Queries spanning multiple partitions return consistent results across partitions until the swapping stage is reached.
- When the partitions are rebuilt with SWAP, the storage requirement for the operation is equal to the size of the existing index partition. Since index partition data and metadata are swapped after re-creation, queries spanning multiple partitions will not return consistent results from partition to partition, but will always be correct with respect to each index partition.
- If SWAP is specified, then DML and synchronization on the partition are blocked during the swap process.

See Also: *Oracle Text Reference* for complete information about RECREATE_INDEX_ONLINE

Re-creating a Global Index with Time Limit for Synch

You can control index re-creation to set a time limit for SYNC_INDEX during non-business hours and incrementally re-create the index. You use the CREATE_SHADOW_INDEX procedure with POPULATE_PENDING and maxtime.

See Also: *Oracle Text Reference* for information and examples for CREATE_SHADOW_INDEX

Re-creating a Global Index with Scheduled Swap

With CTX_DDL.EXCHANGE_SHADOW_INDEX you can perform index re-creation during non-business hours when query failures and DML blocking can be tolerated.

> **See Also:** *Oracle Text Reference* for information and examples for CTX_DDL.EXCHANGE_SHADOW_INDEX

Re-creating a Local Index with All-at-Once Swap

You can re-create a local partitioned index online to create or change preferences. The swapping of the index and partition metadata occurs at the end of the process. Queries spanning multiple partitions return consistent results across partitions when re-create is in process, except at the end when EXCHANGE_SHADOW_INDEX is running.

Scheduling Local Index Re-creation with All-at-Once Swap

With RECREATE_INDEX_ONLINE of the CTX.DDL package, you can incrementally re-create a local partitioned index, where partitions are all swapped at the end.

> **See Also:** *Oracle Text Reference* for complete information and the example for this process with RECREATE_INDEX_ONLINE

Re-creating a Local Index with Per-Partition Swap

Instead of swapping all partitions at once, you can re-create the index online with new preferences, with each partition being swapped as it is completed. Queries across all partitions may return inconsistent results during this process. This procedure uses CREATE_SHADOW_INDEX with RECREATE_INDEX_ONLINE.

> **See Also:** *Oracle Text Reference* for an example of swapping index partitions with the RECREATE_INDEX_ONLINE procedure

Rebuilding an Index

You can rebuild a valid index using ALTER INDEX. Rebuilding an index does not allow most index settings to be changed. 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:** "Re-creating an Index" on page 3-33 for information about changing index settings

Example: Rebuilding an Index

The following statement 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: Dropping a Preference

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. This section contains the following topics:

- Viewing Pending DML
- Synchronizing the Index
- Optimizing the Index

Note: CTXCAT indexes are transactional and, thus, they are updated immediately when there is a change 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, enter 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 sections describe how you can control the duration and locking behavior for synchronizing an index:

- Maxtime Parameter for SYNC_INDEX
- Locking Parameter for SYNC_INDEX

Example

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

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

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

Maxtime Parameter for SYNC_INDEX

The `sync_index` procedure includes a `maxtime` parameter that, like `optimize_index`, indicates a suggested time limit in minutes for the operation. The `sync_index` will process as many documents in the queue as possible within the given time limit.

- NULL `maxtime` is equivalent to `CTX_DDL.MAXTIME_UNLIMITED`.
- The time limit is approximate. The actual time taken may be somewhat less than, or greater than what you specify.
- There is no change to the `ALTER INDEX... sync` command, which is deprecated.
- The `maxtime` parameter is ignored when `sync_index` is invoked without an index name.
- The `maxtime` parameter cannot be communicated for automatic synchronizations (for example, `sync on commit` or `sync every`).

Locking Parameter for SYNC_INDEX

The locking parameter of `sync_index` enables you to configure how the synchronization deals with the scenario where another sync is already running on the index.

- The locking parameter is ignored when `sync_index` is invoked without an index name.
- The locking parameter cannot be communicated for automatic syncs (i.e. sync on commit/sync every)
- When locking mode is `LOCK_WAIT`, in the event of not being able to get a lock, it will wait forever and ignore the maxtime setting.

The options are as follows:

Option	Description
`CTX_DDL.LOCK_WAIT`	If another sync is running, wait until the running sync is complete, then begin new sync.
`CTX_DDL.LOCK_NOWAIT`	If another sync is running, immediately return without error.
`CTX_DDL.LOCK_NOWAIT_ERROR`	If another sync is running, immediately raise an error (DRG-51313: timeout while waiting for DML or optimize lock).

Optimizing the Index

Frequent index synchronization ultimately causes fragmentation of your `CONTEXT` index. Index fragmentation can adversely affect query response time. You should, therefore, allow time to optimize your `CONTEXT` index in order to reduce fragmentation and index size and to ensure optimal 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 (for example, 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 as follows:

```
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 mode.

Less frequent batch processing results in longer document lists, reducing the number of rows in the index and thus 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, 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, a list of most fragmented tokens, and the amount of garbage data in your index. Although this

report might take a long time 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` statement syntax and examples

4

Querying with Oracle Text

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

- Overview of Queries
- The CONTEXT Grammar
- The CTXCAT Grammar

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 enter 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 enter 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 enter logical, proximity, fuzzy, stemming, thesaurus and wildcard searches. With a correctly configured index, you can also enter 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 enter 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 enters 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 enter 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 enter 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 enter 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 enter depends on how you create your sub-indexes.

See Also: "Creating a CTXCAT Index" on page 3-28

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, enter 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, "Classifying Documents in Oracle Text"

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, enter 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.

Combining everything into 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 `AUTO_FILTER` filter. For the following example, we make two assumptions: one, that the document data is in the bind variable `:doc_blob`; and, two, that we have already defined a policy, `my_policy`, that `CTX_DOC.POLICY_FILTER` can use. For example:

```
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       => 'AUTO_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, because you need to get the text into a CLOB to enter a MATCHES query. It takes as one argument the name of a policy you have already created with CTX_DDL.CREATE_POLICY. (See *Oracle Text Reference* for information on CTX_DOC.POLICY_FILTER.)

If your file is text in the database character set, then 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 AUTO_FILTER filtering, then you can load the file into a BLOB instead, and call CTX_DOC.POLICY_FILTER, as previously shown.

> **See Also:** Chapter 6, "Classifying Documents in Oracle Text" 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, "Classifying Documents in Oracle Text" 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*, enter a query specifying *dog* as your query term.

You can enter 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*, enter 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" on page 4-15

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" on page 4-15

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" on page 4-4

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 entered 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 enter a query on a phrase containing stopwords and non-stopwords, then 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. Because 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 10, "Working With a Thesaurus in Oracle Text"

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, "Searching Document Sections in Oracle Text"

Using Query Templates

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

- Query Rewrite
- Query Relaxation
- Query Language
- Alternative and User-defined 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 enter the queries *{kukui nut}* and *{kukui or nut}* 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 entered in sequence as the following:

{black} {pen}

{black} NEAR {pen}

{black} AND {pen}

{black} ACCUM {pen}

The query relaxation 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 `DOMAIN_INDEX_SORT` hint, or the `FIRST_ROWS` hint, which has been deprecated, 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 and User-defined Scoring

You can use query templating to specify alternative scoring algorithms to use, other than the default, to customize how `CONTAINS` is scored, and to enable `SDATA` to be used as part of the scoring expressions. In this way, you can mathematically define the scoring expression using not only pre-defined scoring components, but also `SDATA` components.

With alternative user-defined scoring, you can:

- Specify the scoring expressions of terms by defining arithmetic expressions that define how the query should be scored, using
  - predefined scoring algorithms: `DISCRETE`, `OCCURRENCE`, `RELEVANCE`, and `COMPLETION`;
  - arithmetic operations: plus, minus, multiply, divide;
  - arithmetic functions: `ABS(n)`, finding the absolute value of *n* ; `LOG(n)`, finding the base-10 logarithmic value of *n*;
  - Numeric literals.
- Specify the scoring expressions at the term level.
- Specify terms that should not be taken into account when calculating the score.
- Specify how the score from child elements of `OR` and `AND` operators should be merged.
- Use `SDATA` that stores numeric or `DATETIME` values to affect the final score of the document.

The following example specifies an alternative scoring algorithm:

```
select id from docs where CONTAINS (text,
'<query>
 <textquery grammar="CONTEXT" lang="english"> mustang  </textquery>
```

```
 <score datatype="float" algorithm="DEFAULT"/>
</query>')>0
```

The following query templating example includes SDATA values as part of the final score:

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

See Also: "Using DEFINESCORE and DEFINEMERGE for User-defined Scoring" on page 4-21

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 Text enables 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 entered. 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 enter the query
Stemming	Searches for words with same root as specified term.	$ operator at when you enter the query
Fuzzy Searching	Searches for words that have similar spelling to specified term.	FUZZY operator when you enter 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.

- ABOUT Query
- Logical Operators
- Section Searching
- Proximity Queries with NEAR and NEAR_ACCUM Operators
- Fuzzy, Stem, Soundex, Wildcard and Thesaurus Expansion Operators
- Using CTXCAT Grammar
- Stored Query Expressions
- Calling PL/SQL Functions in CONTAINS
- Optimizing for Response Time
- Counting Hits
- Using DEFINESCORE and DEFINEMERGE for User-defined Scoring

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 enter a theme query, your index must have a theme component which is created by default in English and French.

Enter 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'
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, "Searching Document Sections in Oracle Text"

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*, enter 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?" on page 7-14

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 SQL 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 Language 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" on page 7-4

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 About Query Performance" on page 7-11

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*, enter 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`

Using DEFINESCORE and DEFINEMERGE for User-defined Scoring

The DEFINESCORE operator enables you to define how the score for a term or phrase is to be calculated. The DEFINEMERGE operator defines how to merge scores of child elements of AND and OR operators. You can also use the alternative scoring template with SDATA to affect the final scoring of the document.

See Also:

- "Alternative and User-defined Scoring" on page 4-13 for information about the alternative scoring template
- *Oracle Text Reference* to learn more about the syntax of DEFINESCORE and DEFINEMERGE

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:

Table 4–3 CATSEARCH Query Operator 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.
( )	(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 enter proximity, thesaurus, or ABOUT queries with a CTXCAT index.

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

5

Presenting Documents in Oracle Text

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
- Query offset information for the document
- A concordance of the document, in which occurrences of the query term are returned with their surrounding text

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

These are the highlighting procedures in `CTX_DOC`:

- `CTX_DOC.MARKUP` and `CTX_DOC.POLICY_MARKUP`
- `CTX_DOC.HIGHLIGHT` and `CTX_DOC.POLICY_HIGHLIGHT`
- `CTX_DOC.SNIPPET` and `CTX_DOC.POLICY_SNIPPET`

The `POLICY` and non-`POLICY` versions of the procedures are equivalent, except that the `POLICY` versions do not require an index.

Markup Procedure

The CTX_DOC.MARKUP and CTX_DOC.POLICY_MARKUP procedures take a document reference and a query, and return a marked-up version of the document. The output can be either marked-up plaintext or marked-up HTML. For example, you might specify that a marked-up document be returned with the query term surrounded by angle brackets (<<<*tansu*>>>) or HTML (<*b*>*tansu*</*b*>).

CTX_DOC.MARKUP and CTX_DOC.POLICY_MARKUP are equivalent, except that CTX_DOC.POLICY_MARKUP does not require an index.

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—in this case, the query term will display in red—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,
                     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` and `CTX_DOC.POLICY_SNIPPET`

Highlight Procedure

`CTX_DOC.HIGHLIGHT` and `CTX_DOC.POLICY_HIGHLIGHT` take a query and a document and return offset information for the query in either plaintext or HTML formats. This offset information can be used to write your own custom routines for displaying documents.

`CTX_DOC.HIGHLIGHT` and `CTX_DOC.POLICY_HIGHLIGHT` are equivalent, except that `CTX_DOC.POLICY_HIGHLIGHT` does not require an index.

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` and `CTX_DOC.POLICY_HIGHLIGHT`

Concordance

`CTX_DOC.SNIPPET` and `CTX_DOC.POLICY_SNIPPET` produce a *concordance* of the document, in which occurrences of the query term are returned with their surrounding text. This result is sometimes known as Key Word in Context, or KWIC, because instead of returning the entire document (with or without the query term highlighted), it returns the query term in text fragments, allowing a user to see it in context. You can control the way the query term is highlighted in the returned fragments.

`CTX_DOC.SNIPPET` and `CTX_DOC.POLICY_SNIPPET` are equivalent, except that `CTX_DOC.POLICY_SNIPPET` does not require an index.

> **See Also:** *Oracle Text Reference* for more information about `CTX_DOC.SNIPPET` and `CTX_DOC.POLICY_SNIPPET`

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, enter the following:

```
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, enter the following:

```
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 nondefault 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 nondefault 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.

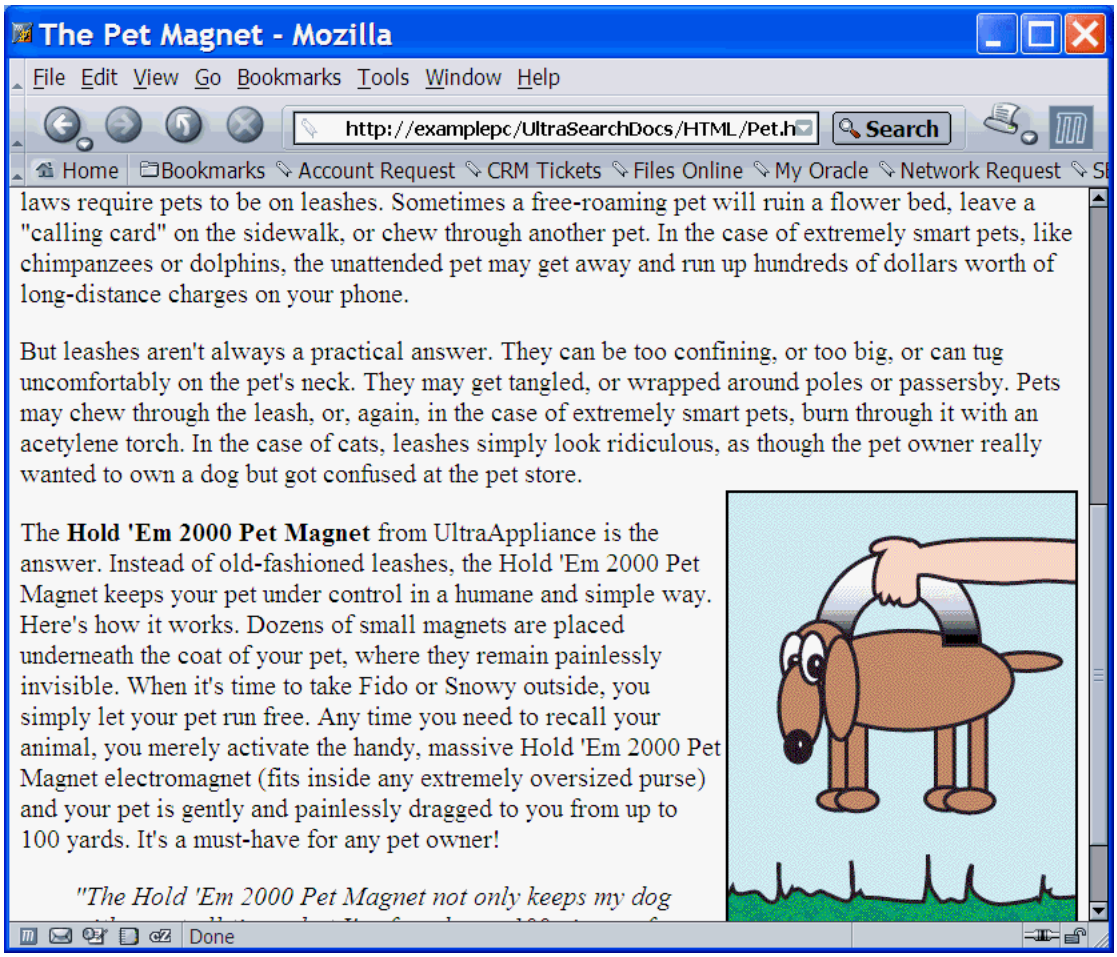
Table 5–2 CTX_DOC Output

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: *Oracle Text Reference*

Figure 5–1 shows an original document to which we can apply highlighting, gisting, and theme extraction in the following sections.

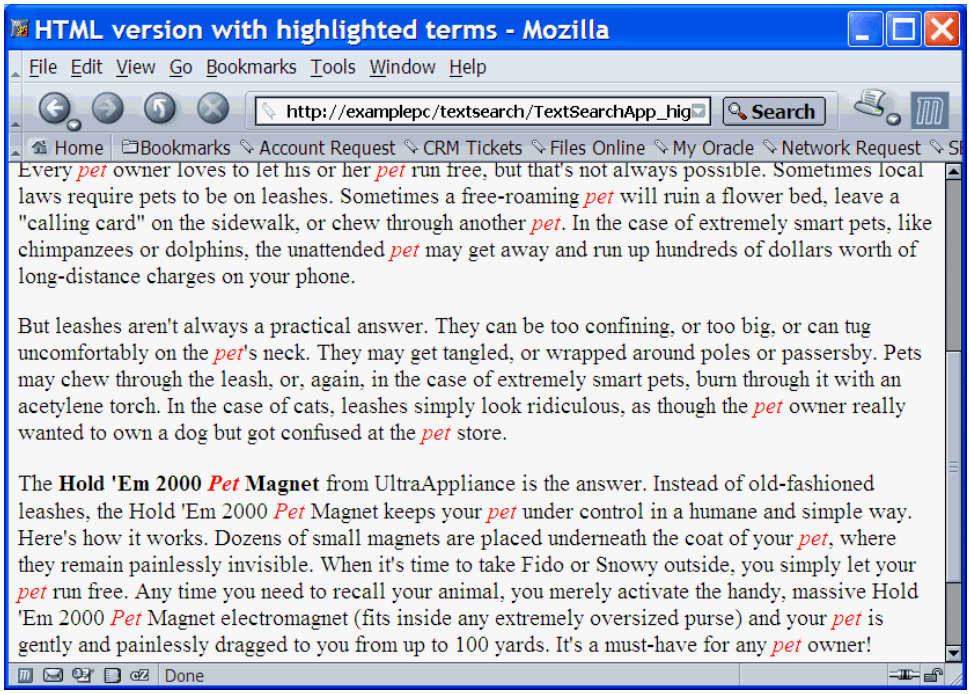
Figure 5–1 Sample Document for Highlighting, Gisting, and Theme Extraction



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".

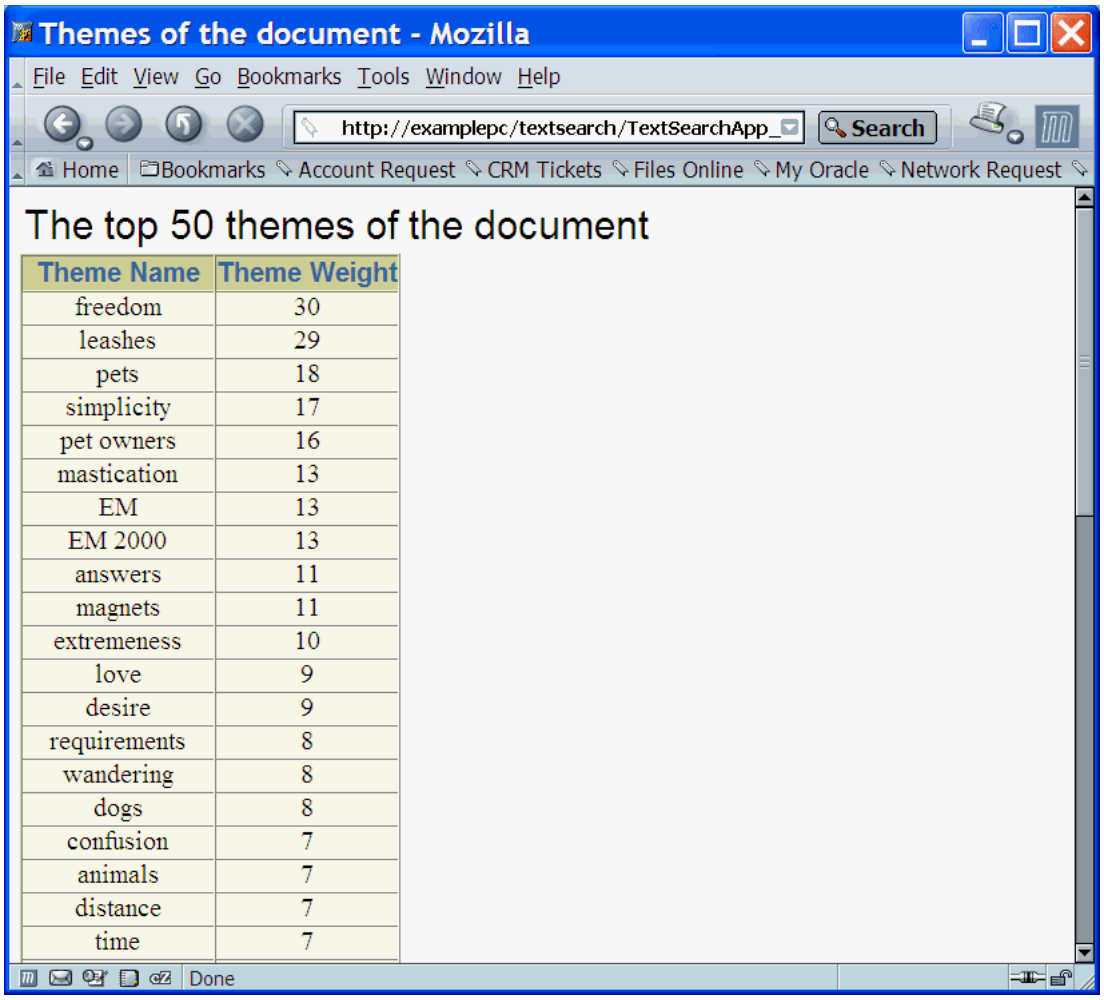
Figure 5–2 Pet Highlighted in Pet Magnet Document



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

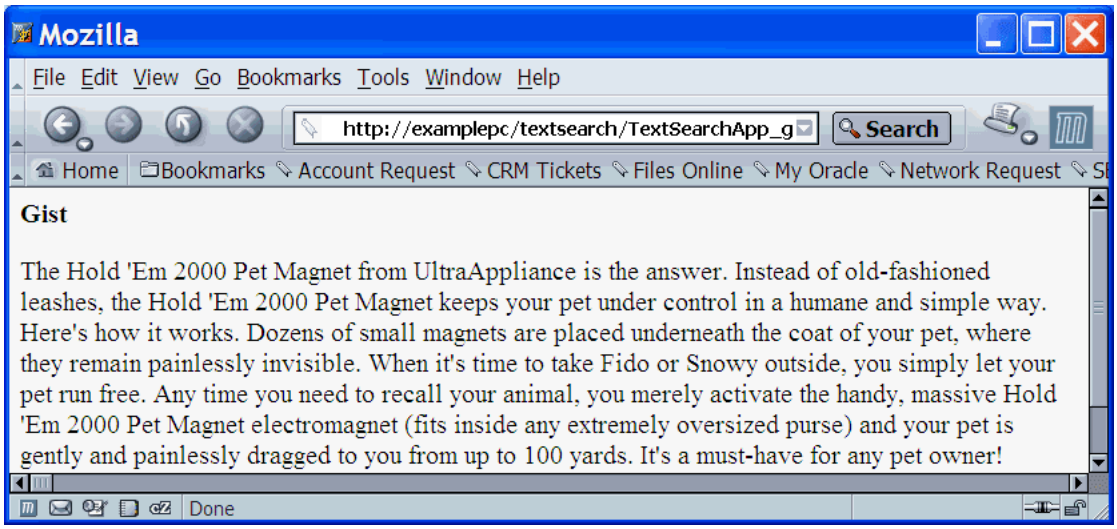


Theme Name	Theme Weight
freedom	30
leashes	29
pets	18
simplicity	17
pet owners	16
mastication	13
EM	13
EM 2000	13
answers	11
magnets	11
extremeness	10
love	9
desire	9
requirements	8
wandering	8
dogs	8
confusion	7
animals	7
distance	7
time	7

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

Classifying Documents in Oracle Text

This chapter includes the following topics:

- Overview of Document Classification
- Classification Solutions
- Rule-Based Classification
- Supervised Classification
- Unsupervised Classification (Clustering)

Overview of Document Classification

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 do not 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.

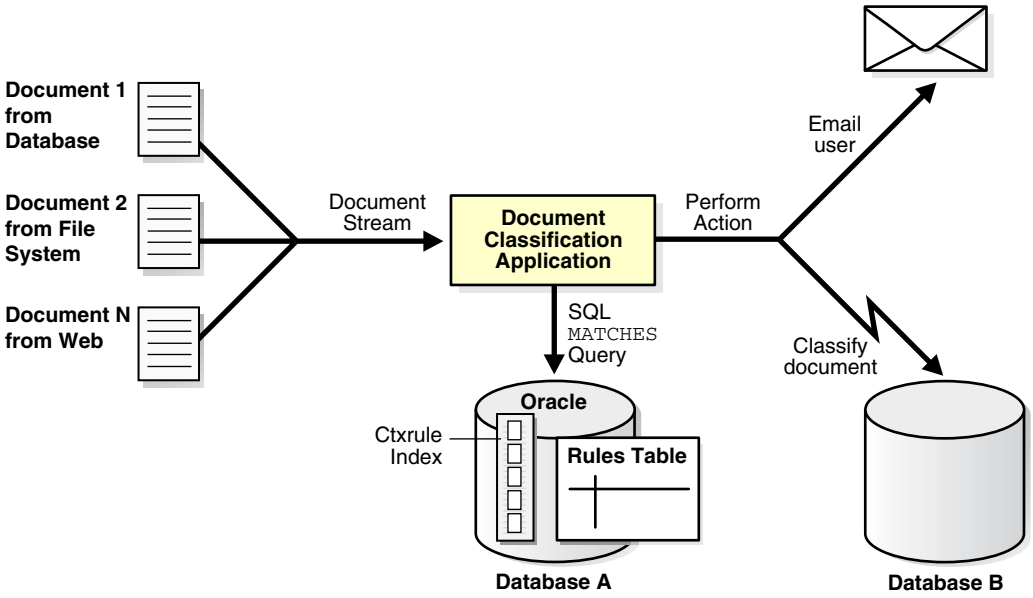
Oracle Text offers various approaches to document classification. Under *rule-based classification*, you write the classification rules yourself. With *supervised classification*, Oracle Text creates classification rules based on a set of sample documents that you pre-classify. Finally, with *unsupervised 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, because 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 6-7.

  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 correlates them with clusters according to content.

  Advantages:

  - You do not 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, because 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" on page 6-6 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 for each of our categories.

Defined Categories	-	-
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 as follows:

```
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.

- If the `SVM_CLASSIFIER` classifier is used, then you may use the `BASIC_LEXER`, `CHINESE_LEXER`, `JAPANESE_LEXER`, or `KOREAN_MORPH_LEXER` lexers. If `SVM_CLASSIFIER` is not used, only the `BASIC_LEXER` lexer type may be used for indexing your query set. (See *Oracle Text Reference* for more on lexer and classifier preferences.)
- 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. 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" on page 3-30

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 *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 enables 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 do not 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 populating.

```
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,
  type number(3) not null,
  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 do not even have to provide a training set of documents.

Clustering is performed with the CTX_CLS.CLUSTERING procedure. CTX_CLS.CLUSTERING creates a hierarchy of document groups, known as *clusters*, and, for each document, returns relevancy scores for all leaf clusters.

For example, suppose that you have a large collection of documents concerning animals. CTX_CLS.CLUSTERING might create one leaf cluster about dogs, another about cats, another about fish, and a fourth cluster about bears. (The first three might

be grouped under a node cluster concerning pets.) Suppose further that you have a document about one breed of dogs, such as chihuahuas. CTX_CLS.CLUSTERING would assign the dog cluster to the document with a very high relevancy score, while the cat cluster would be assigned with a lower score and the fish and bear clusters with still lower scores. When scores for all clusters have been assigned to all documents, an application can then take action based on the scores.

As noted in "Decision Tree Supervised Classification" on page 6-7, attributes used for determining clusters 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 showing how similar the document is to each leaf cluster. This information takes the form of document identification, cluster identification, and a similarity score between the document and a cluster.
- 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, and a quality score for the cluster.

CTX_CLS.CLUSTERING employs a K-MEAN algorithm to perform clustering. Use the KMEAN_CLUSTERING preference to determine how CTX_CLS.CLUSTERING works.

> **See Also:** *Oracle Text Reference* for more on cluster types and hierarchical clustering

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, 'Oracle Database 11g 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

Tuning Oracle Text

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
- Composite Domain Index (CDI) in Oracle Text
- Solving Index and Query Bottlenecks Using Tracing
- Using Parallel Queries
- Tuning Queries with Blocking Operations
- Frequently Asked Questions 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 *Oracle Database PL/SQL Packages and Types Reference*.

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 (CBO) to determine the best execution plan for a query. To enable the optimizer to better estimate costs, you can calculate the statistics on the table you query. To do so, enter 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, enter the `ANALYZE` statement as many times as necessary or use the `DBMS_STATS` package

By collecting statistics on the Text domain index, the cost-based optimizer in Oracle Database is able to perform the following tasks:

- 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.

Note: Oracle Text index tables, namely `DR$` tables (`$I`, `$R`, `$K`, `$N`, `$P`, `$S`, and, if the context index is local partitioned, then the names for the internal tables start with `DR#`), are excluded from `GATHER_DATABASE_STATS` and `GATHER_SCHEMA_STATS`.

Also, explicitly gathering statistics on these tables is strongly discouraged because internal SQL statements on these tables are hinted to use appropriate execution plans.

See Also:

- *Oracle Database SQL Language Reference* and *Oracle Database Performance Tuning Guide* for more information about the `ANALYZE` statement
- *Oracle Database 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, enter 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) Hint 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 About Query Performance" on page 7-11

Improved Response Time with FIRST_ROWS(n) Hint for ORDER BY Queries

When you need the first rows of an `ORDER BY` query, Oracle recommends that you use the cost-based `FIRST_ROWS(n)` hint.

> **Note:** As the `FIRST_ROWS(n)` hint is cost-based, Oracle recommends that you collect statistics on your tables before you use this hint. See "Collecting Statistics" on page 7-2.

You use the `FIRST_ROWS(n)` hint in cases where you want the first number (*n*) 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, 'Omophagia')>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 *omophagia* 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(n)` hint, the optimizer instructs the Text index to return rowids in score-sorted order when the cost of returning the top n hits is lower.

Without the hint, Oracle Database 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 this way takes time.

Because 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 DOMAIN_INDEX_SORT Hint

You can also optimize for response time using the related `DOMAIN_INDEX_SORT` 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 /*+ DOMAIN_INDEX_SORT */ 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 as follows:

```
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 that 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 `DOMAIN_INDEX_SORT` 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 `DOMAIN_INDEX_SORT` hint. Specifically, you can use the `DOMAIN_INDEX_SORT` 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 `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 /*+ DOMAIN_INDEX_SORT */ 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 `DOMAIN_INDEX_SORT`
- *Oracle Database Performance Tuning Guide* and *Oracle Database SQL Language Reference* for more information about the `EXPLAIN PLAN` statement

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(n) Mode

In `FIRST_ROWS(n)` mode, the optimizer in Oracle Database 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(n)` hint.

If you want to optimize for better throughput under `FIRST_ROWS(n)`, 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(10) 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(n)` and `CHOOSE`

Composite Domain Index (CDI) in Oracle Text

The Composite Domain Index feature of the Extensibility Framework in Oracle Database 11*g*, enables structured columns to be indexed by Oracle Text. Therefore, both text and one or more structured criteria can be satisfied by one single Oracle Text index row source. Performance for the following types of query are improved:

- Text query with structured criteria in the SQL `WHERE` clause.
- Text query with structured `ORDER BY` criteria.
- Combination of both of the previous two query types.

As with concatenated btree indexes or bitmap indexes, applications will experience slow-down in DML performance as the number of `FILTER BY` and `ORDER BY` columns increases. Where `SCORE`-sort push-down is optimized for response time, the

structured sort or combination of SCORE and structured sort push-down are also optimized for response time, and not for throughput. However, using DOMAIN_INDEX_SORT or FIRST_ROWS(n) hints to force the sort to be pushed into CDI while fetching the entire hitlist may result in poor query response time.

Performance Tuning with CDI

Support for mapping a FILTER BY column to MDATA enables query performance to be optimized for equality searches by restricting supported functionality of RANGE and LIKE. However, mapping a FILTER BY column to MDATA is not recommended if the FILTER BY column contains sequential values, or has very high cardinality. Doing so can result in a very long and narrow $I table and reduced $X performance. One example of such a sequential column might be one that uses DATE stamp. For such sequential columns, mapping to SDATA is recommended.

The following hints can be used to push or not push the SORT and FILTER BY predicates into the CDI:

- DOMAIN_INDEX_SORT. The query optimizer will try to push the applicable sorting criteria into the specified composite domain index.
- DOMAIN_INDEX_NO_SORT. The query optimizer will try NOT to push sorting criteria into the specified composite domain index.
- DOMAIN_INDEX_FILTER(*table name index name*). The query optimizer will try to push the applicable FILTER BY predicate(s) into the specified composite domain index.
- DOMAIN_INDEX_NO_FILTER(*table name index name*). The query optimizer will not try to push the applicable FILTER BY predicate(s) into the specified composite domain index.

Example 7–1 Performance Tuning a Text Query with CDI Hints

The following example performs an optimized query on the table books.

```
SELECT bookid, pub_date, source FROM
  (SELECT /*+ domain_index_sort domain_index_filter(books books_ctxcdi) */ bookid,
pub_date, source
      FROM books
      WHERE CONTAINS(text, 'aaa',1)>0 AND bookid >= 80
      ORDER BY PUB_DATE desc nulls last, SOURCE asc  nulls last, score(1) desc)
 WHERE rownum < 20;
```

Note: The domain_index_filter hint does not force the query optimizer to use CDI. Instead, if the cost-based optimizer chooses to use the CDI, then it should also push the filter predicate into the index. To force the query optimizer to choose CDI index, you additionally need to use the INDEX hint.

Solving Index and Query Bottlenecks Using 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.

See Also: *Oracle Text Reference*

Using Parallel Queries

In general, parallel queries are optimal for DSS, OLAP, or analytical systems with large data collection, multiple CPUs with a low number of concurrent users, or parallelized across Oracle Real Application Clusters (Oracle RAC) nodes.

Oracle Text supports parallel queries as follows:

- Parallel Queries on a Local Context Index
- Parallelizing Queries Across Oracle RAC Nodes

Parallel Queries on a Local Context Index

Parallel query refers to the parallelized processing of a local `CONTEXT` index. Based on the parallel degree of the index and various system attributes, Oracle determines the number of parallel query slaves to be spawned to process the index. Each parallel query slave processes one or more index partitions. This is the default query behavior for local indexes created in parallel.

However, for heavily loaded systems with high numbers of concurrent users, query throughput will generally be worse with parallel query because top N hits can usually be satisfied by the first few partitions, if the query is run serially. For example, typical top N text queries with an `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 an `ALTER INDEX` statement 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 specifying:

```
Alter index <text index name> paralllel < parallel degree >;
```

Parallelizing Queries Across Oracle RAC Nodes

When considering whether to use Oracle Real Application Clusters (Oracle RAC) to improve Oracle Text performance, it is essential to understand which performance issues that you are trying to solve. Oracle RAC is a great solution for improving query throughput. If you can get good performance from Oracle Text with a light query load, then you can expect to get excellent scalability from Oracle RAC as the query load increases.

Further improvements in Oracle Text performance in an Oracle RAC environment may be achieved by physically partitioning the text data and text indexes (using local partitioned indexes), and ensuring that partitions are handled by separate Oracle RAC nodes. This way, you avoid duplication of the cache contents across multiple nodes and, therefore, maximize the benefit of Oracle RAC cache fusion.

In Oracle 10g Release 1, each Oracle Text index partition must be forced into a separate database file when the index is created. This enables the use of the "re-mastering" feature in Oracle RAC to force database file affinity, in which each node concentrates on a particular database file and, therefore, a particular Oracle Text index partition.

In Oracle 10*g* Release 2 and forward, Oracle supports database object-level affinity, which makes it much easier to allocate index objects (`$I` and `$R` tables) to particular nodes.

While Oracle RAC offers solutions for improving query throughput and performance, is not a "magic bullet," and it will not necessarily enable you to continue to get the same performance improvements as you scale up the data volumes. You are more likely to see improvements by increasing the amounts of memory available to the SGA cache, or by partitioning your data in such a way that queries will normally not need to hit all of the partitions of a table in order to provide the required set of query results.

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 cost-based optimizer in Oracle Database 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
```

Because 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 running 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.

Because 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, enter:

```
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 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 each 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.

If your queries are always pure text queries (no structured predicate and no joins), you should delete statistics on your Oracle Text index.

See Also: "Optimizing Queries with Statistics" on page 7-1

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 e-mails, 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 an 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. Because 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 query completes more quickly. 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 requires 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 (for example, 10 to 20) does not cause difficulty, but avoid having large numbers of explansions (80 or 100) 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" on page 7-5

Should I query in parallel?

Answer: It depends on system load and server capacity. 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 particularly appropriate for DSS or analytical systems with large data collections, multiple CPUs, and low number of concurrent users.

> **See Also:** "Using Parallel Queries" on page 7-9

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" on page 4-8

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. Because he is 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 `DOMAIN_INDEX_SORT` hint in conjunction with `ORDER BY SCORE(n) DESC` tells the Oracle optimizer to accept a sorted set from the text index, and not to do a further sort.

See Also: "Optimizing Queries for Response Time" on page 7-4

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. The speed of indexing will depend on the power of the hardware involved. Indexing speed depends on CPU and I/O capacity. Given sufficient I/O capacity to read in the original data and write out index entries, then CPU will be the limiting factor.

Tests with Intel x86 (Core 2 architecture, 2.5GHz) CPUs have shown that Oracle Text can index around 100GB of text per CPU core, per day. This would be expected to increase as CPU clock speeds increase and/or CPU architectures become more efficient.

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.

Oracle recommends that you use a large index memory setting. Large settings, even up to hundreds of megabytes, can improve the speed of indexing, and reduce the fragmentation of the final indexes. However, if you set the index memory setting too high, then memory paging can occur that will reduce 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 want to get as much information cached in the system global area's (SGA) block buffer cache as possible. So you should allocate 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 e-mails or discussion notes, because 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 because 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, because 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 `AUTO_FILTER` 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.

Overnight indexing is acceptable for many systems. In this case, 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" on page 3-36

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
> - "Optimizing the Index" on page 3-37

Does memory allocation affect index synchronization?

Answer: Yes, the same way as for normal indexing. 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

Searching Document Sections in Oracle Text

This chapter describes how to use document sections in an Oracle Text query application.

The following topics are discussed in this chapter:

- About Oracle Text Document Section Searching
- HTML Section Searching with Oracle Text
- XML Section Searching with Oracle Text

About Oracle Text 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.

This section contains these topics:

- Enabling Oracle Text Section Searching
- Oracle Text Section Types

Enabling Oracle Text 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.

Table 8–1 lists the different types of section groups you can use:

Table 8–1 Types of Section Groups

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.
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 bytes. 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.

Note: Documents sent to the HTML, XML, AUTO and PATH sectioners must begin with \s*<, where \s* represents zero or more whitespace characters. Otherwise, the document is treated as a plaintext document, and no sections are recognized.

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, then you need not explicitly define sections. The system does this for you during indexing.

See Also:

- "Oracle Text Section Types" on page 8-4 for more information about sections
- "XML Section Searching with Oracle Text" on page 8-13 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 the 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, enter 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 enter queries, you can enter path queries with the INPATH and HASPATH operators.

See Also:

- "XML Section Searching with Oracle Text" on page 8-13 to learn more about using these operators
- *Oracle Text Reference* to learn more about using the `INPATH` operator

Oracle Text Section Types

All section types are blocks of text in a document. However, sections can differ in the way that they are delimited and the way that they are recorded in the index. Sections can be one of the following types:

- Zone Section
- Field Section
- Stop Section
- MDATA Section
- SDATA Section
- Attribute Section (for XML documents)
- Special Sections (sentence or paragraphs)

Table 8–2 shows which section types may be used with each kind of section group.

Table 8–2 Section Types and Section Groups

Section Group	ZONE	FIELD	STOP	MDATA	SDATA	ATTRIBUTE	SPECIAL
NULL	NO	NO	NO	NO	NO	NO	YES
BASIC	YES	YES	NO	YES	YES	NO	YES
HTML	YES	YES	NO	YES	YES	NO	YES
XML	YES	YES	NO	YES	YES	YES	YES
NEWS	YES	YES	NO	YES	YES	NO	YES
AUTO	NO	NO	YES	NO	NO	NO	NO
PATH	NO	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 separately 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. Also, 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.

After 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
- The CTX_DDL package chapter of *Oracle Text Reference* for information on adding and removing MDATA sections

NDATA Section

Fields containing data to be indexed for name searching can be specified exclusively by adding NDATA sections to section groups of type: BASIC_SECTION_GROUP, HTML_SECTION_GROUP, or XML_SECTION_GROUP.

Users can synthesize textual documents, which contain name data, using two possible datastores: MULTI_COLUMN_DATASTORE or USER_DATASTORE. The following is an example of using MULTI_COLUMN_DATASTORE to pick up relevant columns containing the name data for indexing:

```
create table people(firstname varchar2(80), surname varchar2(80));
insert into people values('John', 'Smith');
exec ctx_ddl.set_attribute('nameds', 'columns', 'firstname,surname');
```

This produces the following virtual text for indexing:

```
<FIRSTNAME>
John
</FIRSTNAME>
<SURNAME>
Smith
</SURNAME>
```

You can then create NDATA sections for FIRSTNAME and SURNAME sections:

```
begin
  ctx_ddl.create_section_group('namegroup', 'BASIC_SECTION_GROUP');
  ctx_ddl.add_ndata_section('namegroup', 'FIRSTNAME', 'FIRSTNAME');
  ctx_ddl.add_ndata_section('namegroup', 'SURNAME', 'SURNAME');
end;
/
```

Then create the index using the datastore preference and section group preference created earlier:

```
create index peopleidx on people(firstname) indextype is ctxsys.context
parameters('section group namegroup datastore nameds');
```

NDATA sections support both single- and multi-byte data, however, there are character- and term-based limitations. NDATA section data that is indexed is constrained as follows:

- the number of characters in a single, white space delimited term

511

- the number of white space delimited terms

  255

- the total number of characters, including white spaces

  511

SDATA Section

The value of an SDATA section is extracted from the document text like other sections, but is indexed as structured data, also referred to as SDATA. Using SDATA sections supports operations such as projection, range searches, and ordering. It also enables SDATA indexing of section data such as embedded tags, and detail table or function invocations. This enables you to perform various combinations of text and structured searches in one single SQL statement.

SDATA operators should be used only as descendants of AND operators that also have non-SDATA children. SDATA operators are meant to be used as secondary, checking or non-driving, criteria. For instance, "find documents with DOG that also have price > 5", rather than "find documents with rating > 4". Other uses will operate properly, but may not have optimal performance.

You use CTX_DDL.ADD_SDATA_SECTION to add an SDATA section to a section group. When querying within an SDATA section, you use the CONTAINS operator. The following example creates a table called items, and adds an SDATA section called my_sec_group, and then queries SDATA in the section.

Create the table items:

```
CREATE TABLE items
(id  NUMBER PRIMARY KEY,
 doc VARCHAR2(4000));

INSERT INTO items VALUES (1, '<description> Honda Pilot </description>
                              <category> Cars & Trucks </category>
                              <price> 27000 </price>');
INSERT INTO items VALUES (2, '<description> Toyota Sequoia </description>
                              <category> Cars & Trucks </category>
                              <price> 35000 </price>');
INSERT INTO items VALUES (3, '<description> Toyota Land Cruiser </description>
                              <category> Cars & Trucks </category>
                              <price> 45000 </price>');
INSERT INTO items VALUES (4, '<description> Palm Pilot </description>
                              <category> Electronics </category>
                              <price> 5 </price>');
INSERT INTO items VALUES (5, '<description> Toyota Land Cruiser Grill
</description>
                              <category> Parts & Accessories </category>
                              <price> 100 </price>');
COMMIT;
```

Add SDATA section my_sec_group:

```
BEGIN
  CTX_DDL.CREATE_SECTION_GROUP('my_sec_group', 'BASIC_SECTION_GROUP');
  CTX_DDL.ADD_SDATA_SECTION('my_sec_group', 'category', 'category', 'VARCHAR2');
  CTX_DDL.ADD_SDATA_SECTION('my_sec_group', 'price', 'price', 'NUMBER');
END;
```

Create the CONTEXT index:

```
CREATE INDEX items$doc
```

```
  ON items(doc)
  INDEXTYPE IS CTXSYS.CONTEXT
  PARAMETERS('SECTION GROUP my_sec_group');
```

Run a query:

```
SELECT id, doc
  FROM items
  WHERE contains(doc, 'Toyota
                       AND SDATA(category = ''Cars & Trucks'')
                       AND SDATA(price <= 40000 )') > 0;
```

Return the results:

```
  ID DOC
---- ----------------------------------------------------------------------
   2 <description> Toyota Sequoia </description>
                                   <category> Cars & Trucks </category>
                                   <price> 35000 </price>
```

See Also:

- The `CONTAINS` query operators chapter of the *Oracle Text Reference* for information on the `SDATA` operator
- The `CTX_DDL` package chapter of *Oracle Text Reference* for information on adding `SDATA` 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 with Oracle Text" on page 8-13

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 8–3 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 with Oracle Text

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, enter the following query:

```
'oracle within headings'
```

This section contains these topics:

- Creating HTML Sections
- Searching HTML Meta Tags

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 with Oracle Text

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.

The following sections describe the different types of XML searching:

- Automatic Sectioning
- Attribute Searching
- Creating 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 statement 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 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" on page 8-15

Creating Document Type Sensitive Sections

For an XML document set that contains the <book> tag declared for different document types, you may want to create a distinct book section for each document type. The following scenario shows how to create book sections for each document type to improve search capability.

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:

```
<A> <B> <C> dog </C> </B> </A>
```

In this scenario, 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 an Index with PATH_SECTION_GROUP

To enable path section searching, index your XML document set with PATH_SECTION_GROUP. For example:

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

Using Oracle Text Name Search

This chapter describes how to use the name search facility, and includes:

- Overview of Name Search
- Examples of Using Name Search

Overview of Name Search

Someone accustomed to the spelling rules of one culture can have difficulty applying those same rules to a name originating from a different culture. Name searching (also called name matching) provides a solution to match proper names that might differ in spelling due to orthographic variation. It also enables you to search for somewhat inaccurate data, such as might occur when a record's first name and surname are not properly segmented. This ability to handle somewhat inaccurate data is the main advantage of name searching.

Examples of Using Name Search

The following example illustrates how to use NDATA sections to search on names:

```
drop table people;

create table people (
  full_name varchar2(2000)
);

insert into people values
('John Black Smith');

-- multi_column datastore is a convenient way of adding section tags around our
data
exec ctx_ddl.drop_preference('name_ds')
begin
  ctx_ddl.create_preference('name_ds', 'MULTI_COLUMN_DATASTORE');
  ctx_ddl.set_attribute('name_ds', 'COLUMNS', 'full_name');
end;
/

exec ctx_ddl.drop_section_group('name_sg');
begin
  ctx_ddl.create_section_group('name_sg', 'BASIC_SECTION_GROUP');
  ctx_ddl.add_ndata_section('name_sg', 'full_name', 'full_name');
end;
/
-- You can optionally load a thesaurus of nicknames
```

```
-- HOST ctxload -thes -name nicknames -file nicknames.txt

exec ctx_ddl.drop_preference('name_wl');
begin
  ctx_ddl.create_preference('name_wl', 'BASIC_WORDLIST');
  ctx_ddl.set_attribute('name_wl', 'NDATA_ALTERNATE_SPELLING', 'FALSE');
  ctx_ddl.set_attribute('name_wl', 'NDATA_BASE_LETTER', 'TRUE');
  -- Include the following line only if you have loaded the thesaurus
  -- file nicknames.txt:
  -- ctx_ddl.set_attribute('name_wl', 'NDATA_THESAURUS', 'nicknames');
  ctx_ddl.set_attribute('name_wl', 'NDATA_JOIN_PARTICLES',
   'de:di:la:da:el:del:qi:abd:los:la:dos:do:an:li:yi:yu:van:jon:un:sai:ben:al');
end;
/

create index people_idx on people(full_name) indextype is ctxsys.context
  parameters ('datastore name_ds section group name_sg wordlist name_wl');

-- Now you can do name searches with the following SQL:

var name varchar2(80);
exec :name := 'Jon Blacksmith'

select /*+ FIRST_ROWS */ full_name, score(1)
  from people
  where contains(full_name,  'ndata( full_name, '||:name||') ',1)>0
  order by score(1) desc
/
```

The following example illustrates a more complicated version of using NDATA sections to search on names:

```
create table emp (
    first_name    varchar2(30),
    middle_name   varchar2(30),
    last_name     varchar2(30),
    email         varchar2(30),
    phone         varchar2(30));

insert into emp values
('John', 'Black', 'Smith', 'john.smith@example.org', '123-456-7890');

-- user datastore procedure
create or replace procedure empuds_proc
   (rid in rowid, tlob in out nocopy clob) is
     tag varchar2(30);
     phone varchar2(30);
begin
  for c1 in (select FIRST_NAME, MIDDLE_NAME, LAST_NAME, EMAIL, PHONE
             from emp
             where rowid = rid)
  loop
     tag :='<email>';
     dbms_lob.writeappend(tlob, length(tag), tag);
     if (c1.EMAIL is not null) then
         dbms_lob.writeappend(tlob, length(c1.EMAIL), c1.EMAIL);
     end if;
     tag :='</email>';
     dbms_lob.writeappend(tlob, length(tag), tag);
     tag :='<phone>';
     dbms_lob.writeappend(tlob, length(tag), tag);
```

```
      if (c1.PHONE is not null) then
        phone := nvl(REGEXP_SUBSTR(c1.PHONE, '\d\d\d\d($|\s)'), ' ');
        dbms_lob.writeappend(tlob, length(phone), phone);
      end if;
      tag :='</phone>';
      dbms_lob.writeappend(tlob, length(tag), tag);
      tag :='<fullname>';
      dbms_lob.writeappend(tlob, length(tag), tag);
      if (c1.FIRST_NAME is not null) then
        dbms_lob.writeappend(tlob, length(c1.FIRST_NAME), c1.FIRST_NAME);
        dbms_lob.writeappend(tlob, length(' '), ' ');
      end if;
      if (c1.MIDDLE_NAME is not null) then
        dbms_lob.writeappend(tlob, length(c1.MIDDLE_NAME), c1.MIDDLE_NAME);
        dbms_lob.writeappend(tlob, length(' '), ' ');
      end if;
      if (c1.LAST_NAME is not null) then
        dbms_lob.writeappend(tlob, length(c1.LAST_NAME), c1.LAST_NAME);
      end if;
      tag :='</fullname>';
      dbms_lob.writeappend(tlob, length(tag), tag);
    end loop;
  end;
  /

--list
show errors

exec ctx_ddl.drop_preference('empuds');
begin
  ctx_ddl.create_preference('empuds', 'user_datastore');
  ctx_ddl.set_attribute('empuds', 'procedure', 'empuds_proc');
  ctx_ddl.set_attribute('empuds', 'output_type', 'CLOB');
end;
/

exec ctx_ddl.drop_section_group('namegroup');
begin
  ctx_ddl.create_section_group('namegroup', 'BASIC_SECTION_GROUP');
  ctx_ddl.add_ndata_section('namegroup', 'fullname', 'fullname');
  ctx_ddl.add_ndata_section('namegroup', 'phone', 'phone');
  ctx_ddl.add_ndata_section('namegroup', 'email', 'email');
end;
/

-- Need to load nicknames thesaurus
-- ctxload -thes -name nicknames -file dr0thsnames.txt
-- You can find sample nicknames thesaurus file, dr0thsnames.txt, under
-- $ORACLE_HOME/ctx/sample/thes directory.

exec ctx_ddl.drop_preference('ndata_wl');
begin
   ctx_ddl.create_preference('NDATA_WL', 'BASIC_WORDLIST');
   ctx_ddl.set_attribute('NDATA_WL', 'NDATA_ALTERNATE_SPELLING', 'FALSE');
   ctx_ddl.set_attribute('NDATA_WL', 'NDATA_BASE_LETTER', 'TRUE');
   ctx_ddl.set_attribute('NDATA_WL', 'NDATA_THESAURUS', 'NICKNAMES');
   ctx_ddl.set_attribute('NDATA_WL', 'NDATA_JOIN_PARTICLES',
    'de:di:la:da:el:del:qi:abd:los:la:dos:do:an:li:yi:yu:van:jon:un:sai:ben:al');
end;
/
```

```
exec ctx_output.start_log('emp_log');
create index name_idx on emp(first_name) indextype is ctxsys.context
parameters ('datastore empuds section group namegroup wordlist ndata_wl
  memory 500M');

exec ctx_output.end_log;

-- Now you can do name searches with the following SQL:
var name varchar2(80);
exec :name := 'Jon Blacksmith'

select first_name, middle_name, last_name, phone, email, scr from
   (select /*+ FIRST_ROWS */
          first_name, middle_name, last_name, phone, email, score(1) scr
    from emp
    where contains(first_name,
          'ndata(phone, '||:name||') OR  ndata(email,'||:name||') OR
           ndata(fullname, '||:name||') ',1)>0
    order by score(1) desc
   ) where rownum <= 10;
```

10

Working With a Thesaurus in Oracle Text

This chapter describes how to improve your query application with a thesaurus. The following topics are discussed in this chapter:

- Overview of Oracle Text Thesaurus Features
- Defining Terms in a Thesaurus
- Using a Thesaurus in a Query Application
- About the Supplied Knowledge Base

Overview of Oracle Text Thesaurus Features

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 thesauruses that 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: Oracle Text thesaurus formats and functionality are compliant with both the ISO-2788 and ANSI Z39.19 (1993) standards.

Oracle Text Thesaurus Creation and Maintenance

Thesauruses 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 thesauruses from a plain-text file into the thesaurus tables, as well as dumping thesauruses from the tables into output (or 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.

> **Caution: To ensure sound security practices, Oracle recommends that you enter the password for `ctxload` using the interactive mode, which prompts you for the user password. Oracle strongly recommends that you do not enter a password on the command line.**

Using a Case-sensitive Thesaurus

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 thesauruses 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*.

Using a Case-insensitive Thesaurus

In a case-insensitive thesaurus, terms are stored in all-uppercase, regardless of the case in which they were originally entered.

The `ctxload` program loads a thesaurus in case-insensitive mode by default.

When creating a thesaurus with `CTX_THES.CREATE_THESAURUS`, the thesaurus is created as 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.

> **See Also:** The "ctxload Utility" on page 10-2

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` utility

> **See Also:** *Oracle Text Reference* to learn more about using `ctxload` and the `CTX_THES` package, and "ctxload Utility" on page 10-2 in this chapter

Supplied Thesaurus

Although Oracle Text does not provide a default thesaurus, Oracle Text does supply a thesaurus, in the form of a file that you load with `ctxload`, 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 thesauruses 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, and "ctxload Utility" on page 10-2 in this chapter

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:** Oracle Database installation documentation specific to your operating system for more information about the directory structure of Oracle Text

Defining Terms in a Thesaurus

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. Because 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 enter 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 Thesaurus-based Queries

To build a custom thesaurus, follow these steps:

1. Create your thesaurus. See "Defining Terms in a Thesaurus" on page 10-4.
2. Load thesaurus with `ctxload`. The following example imports a thesaurus named `tech_doc` from an import file named `tech_thesaurus.txt`:

   ```
   ctxload -thes -name tech_doc -file tech_thesaurus.txt
   ```

3. At the prompt, enter username and password. To ensure security, do not enter a password at the command line.
4. 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 Terms in a Thesaurus" and "Linking New Terms to Existing Terms".
2. Load thesaurus with `ctxload`. See "Loading a Thesaurus with ctxload" on page 10-7.
3. Compile the loaded thesaurus with `ctxkbtc` compiler. Refer to "Compiling a Loaded Thesaurus" on page 10-7.

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, enter 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
```

When you enter the `ctxload` command line, you are prompted for the user password. For best security practices, never enter the password at the command line. Alternatively, you may omit the `-user` and let `ctxload` prompt you for username and password, respectively.

Compiling a Loaded Thesaurus

To link the loaded thesaurus `medthes` to the knowledge base, use `ctxkbtc` as follows:

```
ctxkbtc -user ctxsys -name medthes
```

When you enter the `ctxkbtc` command line, you are prompted for the user password. As with `ctxload`, for best security practices, do not enter the password at the command line.

IMPORTANT: In order to ensure sound security practices, Oracle recommends that you enter the password for `ctxload` and `ctxkbtc` using the interactive mode, which prompts you for the user password. Oracle strongly recommends that you do not enter a password on the command line.

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" on page 10-5

Note: Oracle Text supplied knowledge bases may not necessarily be installed when Oracle Text is installed. You may need to separately install the knowledge bases if they have not been installed. For more information, refer to Oracle Database installation documentation specific to your operating system.

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" on page 10-8

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`. Refer to "Loading a Thesaurus with ctxload" on page 10-7.
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` as follows:

   ```
   ctxkbtc -user ctxsys -name my_lang_thes
   ```

   Enter the password for `-user` when prompted. Refer to "Compiling a Loaded Thesaurus" on page 10-7.

   This statement 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 apply 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_LANG` 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 thesauruses. In such cases, Oracle recommends that you obtain hierarchical information directly from your thesauruses.

> **See Also:** *Oracle Text Reference* for more information about theme indexing, `ABOUT` queries, using the `CTX_DOC` package, and the supplied English knowledge base

11

Using XML Query Result Set Interface

This chapter describes how to use the XML query result set interface, and includes:

- Overview of the XML Query Result Set Interface
- Using the XML Query Result Set Interface

Overview of the XML Query Result Set Interface

A page of search results in applications can consist of many disparate elements — metadata of the first few documents, total hit counts, per-word hit counts, and so on. Generating these results in earlier versions of Oracle Text required several queries and calls — perhaps a query on the base table, a call to `CTX_QUERY.COUNT_HITS`, and so on. Each extra call takes time to reparse the query and look up index metadata. Additionally, some search operations, such as iterative query refinement or breakdown top ten, are difficult for SQL. If it is even possible to construct a SQL statement to produce the desired results, such SQL is usually suboptimal.

The result set interface is able to produce the various kinds of data needed for a page of search results all at once, thus improving performance by sharing overhead. The result set interface can also return data views that are difficult to express in SQL, such as top *N* by category queries.

Using the XML Query Result Set Interface

The `CTX_QUERY.RESULT_SET()` API enables you to obtain query results with a single query, rather than running multiple `CONTAINS()` queries to achieve the same result. For example, in order to display a search result page, the following information needs to be obtained first:

- top 20 hit list sorted by date and relevancy
- total number of hits for the given Text query
- counts group by publication date
- counts group by author

Assume the following table definition for storing documents to be searched:

```
create table docs (
  docid    number,
  author   varchar2(30),
  pubdate  date,
  title    varchar2(60),
  doc      clob);
```

Assume the following Oracle Text Index definition:

```
create index docidx on docs(doc) indextype is ctxsys.context
filter by author, pubdate, title,
order by pubdate;
```

With these definitions, you can obtain the four pieces of information for displaying the search result page by issuing four SQL statements:

```
-- Get top 20 hits sorted by date and relevancy
select * from
  (select /*+ first_rows */ rowid, title, author, pubdate
   from docs where contains(doc, 'oracle',1)>0
   order by pubdate desc, score(1) desc)
where rownum < 21;

-- Get total number of hits for the given Text query
select count(*) from docs where contains(doc, 'oracle',1)>0;

-- Get counts group by publication date
select pubdate, count(*) from docs where contains(doc, 'oracle',1)>0
group by pubdate;

-- Get counts group by author
select author, count(*) from docs where contains(doc, 'oracle',1)>0 group by
author;
```

As you can see, using separate SQL statements results in a resource-intensive query, as you run the same query four times. However, by using `CTX_QUERY.RESULT_SET()`, you can enter all this information in one single Oracle Text query:

```
declare
   rs clob;
begin
   dbms_lob.createtemporary(rs, true, dbms_lob.session);
   ctx_query.result_set('docidx', 'oracle text performance tuning', '
   <ctx_result_set_descriptor>
    <count/>
    <hitlist start_hit_num="1" end_hit_num="20" order="pubDate desc,
        score desc">
      <score/>
      <rowid/>
         <sdata name="title"/>
      <sdata name="author"/>
      <sdata name="pubDate"/>
    </hitlist>
    <group sdata="pubDate">
      <count/>
    </group>
   <group sdata="author">
     <count/>
   </group>
  </ctx_result_set_descriptor>
 ', rs);

-- Put in your code here to process the Output Result Set XML
   dbms_lob.freetemporary(rs);
exception
   when others then
    dbms_lob.freetemporary(rs);
    raise;
```

```
end;
/
```

The result set output will be an XML containing all the necessary information required to construct the search result page:

```
<ctx_result_set>
  <hitlist>
    <hit>
      <score>90</score>
      <rowid>AAAPoEAABAAAMWsAAC</rowid>
      <sdata name="TITLE"> Article 8 </sdata>
      <sdata name="AUTHOR">John</sdata>
      <sdata name="PUBDATE">2001-01-03 00:00:00</sdata>
    </hit>
    <hit>
      <score>86</score>
      <rowid>AAAPoEAABAAAMWsAAG</rowid>
      <sdata name="TITLE"> Article 20 </sdata>
      <sdata name="AUTHOR">John</sdata>
      <sdata name="PUBDATE">2001-01-03 00:00:00</sdata>
    </hit>
    <hit>
      <score>78</score>
      <rowid>AAAPoEAABAAAMWsAAK</rowid>
      <sdata name="TITLE"> Article 17 </sdata>
      <sdata name="AUTHOR">John</sdata>
      <sdata name="PUBDATE">2001-01-03 00:00:00</sdata>
    </hit>
    <hit>
      <score>77</score>
      <rowid>AAAPoEAABAAAMWsAAO</rowid>
      <sdata name="TITLE"> Article 37 </sdata>
      <sdata name="AUTHOR">John</sdata>
      <sdata name="PUBDATE">2001-01-03 00:00:00</sdata>
    </hit>
...
    <hit>
      <score>72</score>
      <rowid>AAAPoEAABAAAMWsAAS</rowid>
      <sdata name="TITLE"> Article 56 </sdata>
      <sdata name="AUTHOR">John</sdata>
      <sdata name="PUBDATE">2001-01-03 00:00:00</sdata>
    </hit>
  </hitlist>

  <count>100</count>

  <groups sdata="PUBDATE">
    <group value="2001-01-01 00:00:00"><count>25</count></group>
    <group value="2001-01-02 00:00:00"><count>50</count></group>
    <group value="2001-01-03 00:00:00"><count>25</count></group>
  </groups>

  <groups sdata="AUTHOR">
    <group value="John"><count>50</count></group>
    <group value="Mike"><count>25</count></group>
    <group value="Steve"><count>25</count></group>
  </groups>

</ctx_result_set>
```

See Also: *Oracle Text Reference* for syntax details and more information

12

Extracting Entities Using Oracle Text

Entity extraction is the process of locating phrases in an input document and classifying these phrases into a set of categories. This information can then be used to better understand the content of the document. Entities can be concrete objects, such as people or companies, or more abstract objects, such as percentages. Table 12–1, " Entities and Entity Types" illustrates example entities and the entity types they belong to.

Table 12–1 Entities and Entity Types

Entity	Entity Types
United States of America	Country
25%	Percentage
Oracle Corporation	Organization_Company
John Doe	Person

How to Use Entity Extraction

Application developers can use entity extraction in a variety of ways. For example, a search application could use entity extraction to provide the capability of finding all documents in a given corpus that contain a company name, even if the user does not know the exact name of the company he is searching for. Entity extraction can be used to examine a corpus and see what names are most frequently mentioned (that is, most popular), which is something that cannot be done with simple keyword searches. Entity extraction can be used to markup documents with useful links. As an example, an application could use entity extraction to find all the places in their documents, and link the text of each place to a map showing the absolute location.

Oracle Text's entity extraction feature provides an out-of-the-box solution that enables application developers to use entity extraction on their documents inside of the database. In addition, entity extraction allows developers to improve their results by adding user-defined rules and dictionaries to drive the extraction. Oracle provides a large set of supplied entity types, and, in addiation, you can define your own.

The examples in this chapter show how to use and tune entity extraction for a given application.

See Also:

- *Oracle Text Reference* for syntax details and additional examples

Examples of Entity Extraction

The following examples illustrate different cases of using entity extraction.

Example 12–1 Running Entity Extraction

This example illustrates how to use entity extraction to analyze a set of documents in a database. Suppose that our documents are contained in a table named docs, with the following definition:

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

Then, we create a second table called entities that will contain the extracted entities for the given documents:

```
create table entities(id number primary key, ents clob);
```

We now create an entity extraction policy that will govern how the extraction runs. An entity extraction policy represents the various knobs the user has to tune the results of the extraction. For example, the user can set the policy to enable or disable the Oracle-supplied rules or the Oracle-supplied dictionary. Additionally, users can add their own custom rules or dictionary to a given policy.

For now, we will create a policy `p1` that uses the default options, which includes both the Oracle-supplied rules and Oracle-supplied dictionary:

```
exec ctx_entity.create_extract_policy('p1');
```

We then run entity extraction on our input document in the docs table using the PL/SQL procedure `CTX_ENTITY.EXTRACT`. We insert the extracted entities back into the entities table for further inspection:

```
declare
  mydoc clob;
begin
  --put input document into mydoc
  select txt into mydoc from docs where id=1;
  --run extraction using policy p1
  ctx_entity.extract('p1', mydoc, null, myresults);
  --save entities to table
  insert into entities values(1, myresults);
  commit;
end;
/
```

Finally, we can examine the entities by reading the entities table. Below is a sample of entity extraction output from an example document:

```
<entities>
  <entity id="0" offset="4068" length="7" source="SuppliedDictionary">
  <text>Chicago</text>
  <type>city</type>
</entity>
<entity id="1" offset="1102" length="8" source="SuppliedDictionary">
  <text>New York</text>
  <type>city</type>
</entity>
<entity id="2" offset="4326" length="8" source="SuppliedDictionary">
  <text>New York</text>
  <type>city</type>
</entity>
<entity id="7" offset="49" length="28" source="SuppliedRule">
```

```
  <text>American International Group</text>
  <type>company</type></entity>
<entity id="8" offset="3494" length="10" source="SuppliedRule">
  <text>Imax Corp.</text>
  <type>company</type>
</entity>
<entity id="9" offset="2020" length="18" source="SuppliedRule">
  <text>Investment Company</text>
  <type>company</type></entity>
<entity id="10" offset="509" length="11" source="SuppliedDictionary">
  <text>Pfizer Inc.</text>
  <type>company</type>
</entity>
<entity id="13" offset="23" length="4" source="SuppliedDictionary">
  <text>U.S.</text>
  <type>country</type>
  <normal>
    <value>United States of America</value>
  </normal>
</entity>
<entity id="17" offset="3527" length="6" source="SuppliedRule">
  <text>$14.91</text>
  <type>currency</type>
</entity>
<entity id="19" offset="1970" length="12" source="SuppliedRule">
  <text>$172 billion</text>
  <type>currency</type>
</entity>
<entity id="27" offset="1934" length="12" source="SuppliedRule">
  <text>January 2009</text>
  <type>date</type>
</entity>
<entity id="32" offset="1062" length="11" source="SuppliedRule">
  <text>0.1 percent</text>
  <type>percent</type>
</entity>
<entity id="46" offset="3405" length="23" source="SuppliedDictionary">
  <text>Chief Financial Officer</text>
  <type>person_jobtitle</type>
</entity>
<entity id="47" offset="157" length="9" source="SuppliedDictionary">
  <text>President</text>
  <type>person_jobtitle</type>
</entity>
<entity id="48" offset="751" length="7" source="SuppliedDictionary">
  <text>manager</text>
  <type>person_jobtitle</type>
</entity>
<entity id="49" offset="2599" length="15" source="SuppliedRule">
  <text>Robert W. Baird</text>
  <type>person_name</type>
</entity>
<entity id="56" offset="3482" length="7" source="SuppliedDictionary">
  <text>Florida</text>
  <type>state</type>
</entity>
<entity id="57" offset="842" length="8" source="SuppliedDictionary">
  <text>Michigan</text>
  <type>state</type>
</entity>
```

In all, entity extraction found 64 entities in this document, of which a small sample are shown here. Entity extraction finds various types of entities, such as cities, states, companies, and currencies. It also provides the location in the document where entities were found, as well what sources were used to determine the entities. In this business-related document, there were many examples of companies, currencies, and percentages. In addition, because this was a business news article focusing on the U.S., we see many U.S. cities and states in the entity extraction output.

This example shows the most basic use of entity extraction. From this starting point, we can start to augment the extraction policy to improve the precision and recall of the results.

Example 12–2 Extracting Only Certain Entity Types

You can limit what types of entities are extracted in the call to `CTX_ENTITY.EXTRACT` by specifying a comma-delimited list of entity types to extract:

```
declare
  mydoc clob;
  myresults clob;
begin
  select txt into mydoc from docs where id=1;
  ctx_entity.extract('p1', mydoc, null, myresults, 'city,company');
  insert into entities valueS(1, myresults);
  commit;
end;
/
```

This call to `CRX_ENTITY.EXTRACT` will only extract cities and companies. It can lead to a performance improvement, especially when using user-rules.

Example 12–3 Improving Entity Extraction by Adding User-Supplied Rules

In certain scenarios, the supplied entity types might not be descriptive enough. For example, in a discussion about the stock market, a person might want to identify which stocks are gaining or losing, just from unstructured text. Writing user rules with user-defined types provides a way to accomplish this.

Suppose that stocks that are gaining are described by the pattern "climbed *n* percent" or "jumped n percent" where `n` is a number. We can write a regular expression to describe this pattern as:

```
(climbed|jumped) \d+(\.\d+)? percent
```

This regular expression will match phrases of the form "climbed 5 percent" and "jumped 5.3 percent".

The procedure `CTX_ENTITY.ADD_EXTRACT_RULE` is used to associate a regular expression with an extraction policy. This procedure takes in a short XML string that contains the regular expression, which portion of the regular expression corresponds to the actual entity, and what type to assign the extracted text. User-defined types must be prefixed with the letter "x".

Suppose that we name our new entity type `xPositiveGain`. Then the extraction rule will look like the following:

```
<rule>
  <expression>((climbed|jumped) \d+(\.\d+)? percent)></expression>
  <type refid="1">xPositiveGain</type>
</rule>
```

Note that the attribute `refid` in the type tag. The refid corresponds to the backreference number, and indicates what text to extract. For example, the first backreference is the text enclosed by the first set of parentheses. This text will be extracted and classified as an `xPositiveGain` entity.

Next, we add the extraction rule to the policy, and then recompile the policy:

```
exec ctx_entity.add_extract_rule('p1', 1,
  '<rule><expression>((climbed|jumped) \d+(\.\d+)?
  percent)></expression><type refid="1">xPositiveGain</type></rule>');
exec ctx_entity.compile('p1');
```

Now we can rerun the extraction, and will obtain the following additional entities:

```
<entity id="65" offset="2067" length="19" source="UserRole" ruleid="1">
  <text>climbed 3.8 percent</text>
  <type>xPositiveGain</type>
</entity>
<entity id="66" offset="3505" length="18" source="UserRole" ruleid="1">
  <text>jumped 8.7 percent</text>
  <type>xPositiveGain</type>
</entity>
```

Example 12–4 Improving Entity Extraction by Adding a User-Supplied Dictionary

In certain scenarios, you might wish to add fixed terms to a user-supplied dictionary. For example, in a discussion about the stock market, you might want to extract the various indexes that are being referenced in a discussion. These can be inserted into a user-defined dictionary that can be used in the extraction.

First, we write a dictionary describing your additional terms. The user-defined dictionary is another XML document describing entities and their types:

```
<dictionary>
  <entities>
    <entity>
      <value>Dow Jones Industrail Average</value>
      <type>xIndex</type>
    </entity>
    <entity>
      <value>S&amp;P 500</value>
      <type>xIndex</type>
    </entity>
  <entities>
</dictionary>
```

You then load the dictionary using `CTXLOAD`, which is a command-line tool that you use to load an entity extraction dictionary:

```
ctxload -user scott/tiger -extract -name p1 -file ud.xml
```

Then, we set the compile policy and rerun the extraction:

```
exec ctx_entity.compile('p1');

declare
  mydoc clob;
  myresults clob;
begin
  select txt into mydoc from docs where id=1;
  ctx_entity.extract('p1', mydoc, null, myresults);
```

```
  insert into entities values(1, myresults);
  commit;
end;
/
```

We now are able to find the following additional entities:

```
<entity id="67" offset="1231" length="28" source="UserDictionary">
  <text>Dow Jones Industrial Average</text>
  <type>xIndex</type>
</entity>
<entity id="68" offset="1422" length="7" source="UserDictionary">
  <text>S&amp;P 500</text>
  <type>xIndex</type>
</entity>
<entity id="69" offset="1010" length="7" source="UserDictionary">
  <text>S&amp;P 500</text>
  <type>xIndex</type>
</entity>
```

13

Administering Oracle Text

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
- Text Manager in Oracle Enterprise Manager
- Servers and Indexing
- Database Feature Usage Tracking in Oracle Enterprise Manager
- Oracle Text on Oracle Real Application Clusters

Oracle Text Users and Roles

While any user can create an Oracle Text index and enter 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 during installation 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 `SYSDBA` privileges, and only `CTXSYS` could perform certain functions, such as modifying system-defined preferences or setting system parameters.

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 enter 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:** *Oracle Text Reference* for more information about this package

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 *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 *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).

Text Manager in Oracle Enterprise Manager

Oracle Enterprise Manager provides Text Manager for configuring, maintaining, and administering Oracle Text indexes. With Text Manager you can perform all of the basic configuration and administration tasks for Oracle Text indexes. You can monitor the overall health of Text indexes for a single Oracle database instance or for the Oracle Real Application Clusters environment. Text Manager provides summaries of critical information and enables you to drill down to the level of detail that you want, to resolve issues, and to understand any actions that may need to occur. You access Text Manager by clicking the Schema tab from the database home page in Oracle Enterprise Manager, and then selecting **Text Indexes** under the Text Manager group. On the Text Indexes page, select an index name and click **View** to see information and attributes for that index.

The Text Indexes page shows the jobs that are in progress, scheduled within the last seven days, or are experiencing problems. From this page you can go to the Job Scheduler to see a summary of all jobs for this database instance, and to manage selected jobs. The online help in Oracle Enterprise Manager provides details and procedures for using each Text Manager feature.

Note: You cannot create an Oracle Text index with Text Manager. Use the CREATE INDEX statement to create an Oracle Text index as described in Chapter 3, "Indexing with Oracle Text" under Creating Oracle Text Indexes on page 3-20.

Using Text Manager

From the main Text Manager page, you can perform the following actions on the selected index from the Actions list:

- Synchronize
- Optimize
- Rebuild
- Resume Failed Operation
- Show Logs
- Show Errors

You can also schedule jobs for the specified index.

To access Text Manager:

1. Log on to the database with a user account that is authorized to access Database Control. For example, this could be `SYS` or `SYSTEM`, with the password that you specified during database installation.

   Database Control displays the Database Home page.
2. Select the Schema tab from the Database Home page.
3. Click **Text Indexes** located under Text Manager.

   The Text Indexes page appears with a list of Text indexes for this database instance.

When you select a Text index from the Text Indexes page, options become available for that index for you to edit or perform actions. For example, to configure attributes for searching, click **Edit** for the selected index. From the Edit Text Index page, you can set attributes, including: Wild Card Maximum Term, Fuzzy Score, and Number of Fuzzy Expansions. You can change index and partition names, as well as specify settings for `URL_DATASTORE` in addition to other options.

Viewing General Information for a Text Index

You can use the View Text Index page to see general information about a specific index: index type, parallel degree, synchronization mode, wild card limit, fuzzy score, fuzzy numeric result, datastore, and so forth. Information about any partitions on the index is also available.

To view general information for a Text index:

- From the Text Indexes page, click the name of the index in the list of Text indexes.

  The View Text Index page opens with the General tab selected.

From here you can select actions to perform maintenance tasks.

Checking Text Index Health

You use the Text Indexes page in Text Manager to see the list of Text indexes and general health of the Text indexes for the database instance to help you understand any critical actions that may need to be taken in order to make sure that the entire application is performing properly. Information is displayed such as the status of the indexes and jobs submitted by users during the last seven days. Key information about the Text indexes is also displayed in a tabular form.

Use the Text Indexes page to see:

- The number of Text indexes that contain invalid partitions, and which are, therefore, invalid. The number of partitions that are invalid, if any, for all Text indexes is also shown.
- The number of indexes that are in an in-progress state, and the number of partitions, if any, that are in an in-progress state.
- The number of indexes where all partitions are valid and no activity is in progress.
- Sum total of the Text indexes found for this database instance.

Additionally, use the Text Indexes page to see the index type for each Text index, the owner, the number of documents that are not synchronized, total number of documents, and percentage of fragmentation.

You select a Text index from the list and then options become available for that index for you to edit or perform actions.

Servers and Indexing

You index documents and enter 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, or from Text Manager in Oracle Enterprise Manager.

> **See Also:** Chapter 3, "Indexing with Oracle Text" for more information about indexing and index synchronization

Database Feature Usage Tracking in Oracle Enterprise Manager

Database Feature Usage statistics in Oracle Enterprise Manager provide an approximation of how often various database features are used. Tracking this information is potentially useful for application development as well as for auditing. You access Database Feature Usage by clicking the Server tab in Oracle Enterprise Manager, and then selecting **Database Feature Usage** under the Database Configuration group.

The following information is gathered for Oracle Text:

- Package Usage Statistics
- Index Usage Statistics
- SQL Operator Usage Statistics

Package Usage Statistics

For package usage statistics, Database Feature Usage captures information about how often, if ever, and when the following packages have been used:

- `CTX_ADM`
- `CTX_CLS`
- `CTX_DDL`
- `CTX_DOC`
- `CTX_OUTPUT`
- `CTX_QUERY`

- CTX_REPORT
- CTX_THES
- CTX_ULEXER

Index Usage Statistics

For index usage statistics, Database Feature Usage captures the number of existing indexes in the database. The statistics are captured separately for each index type: CONTEXT, CTXCAT, and CTXRULE.

SQL Operator Usage Statistics

For SQL operator usage statistics, Database Feature Usage captures whether the user has ever used the CONTAINS, CATSEARCH, and MATCHES operators.

Note: The feature usage tracking statistics might not be 100% accurate.

Oracle Text on Oracle Real Application Clusters

Oracle Text queries can be parallelized across Oracle RAC nodes for maximum throughput and performance for OLAP applications. You can manage Oracle Text indexes on Oracle RAC nodes with Text Manager in Oracle Enterprise Manager as described in the previous section "Text Manager in Oracle Enterprise Manager" on page 13-5.

See Also: "Parallelizing Queries Across Oracle RAC Nodes" on page 7-10

14

Migrating Oracle Text Applications

This chapter covers issues relating to migrating your applications from previous releases of Oracle Text. This chapter contains the following sections:

- Migrating to Oracle Text 11g Release 2 (11.2)
- Migrating to Oracle Text 11g Release 1 (11.1)
- Migrating to Oracle Text 10g Release 2 (10.2)

Migrating to Oracle Text 11*g* Release 2 (11.2)

No changes to applications are required when migrating to Oracle Text 11*g* Release 2 (11.2).

Migrating to Oracle Text 11*g* Release 1 (11.1)

No changes to applications are required when migrating to Oracle Text 11*g* Release 1 (11.1).

Migrating to Oracle Text 10*g* Release 2 (10.2)

This section covers issues relating to migrating your applications to Oracle Text 10*g* Release 2 (10.2). This information is provided for those users who may be upgrading from the next-to-last release of Oracle Text and who want to know what migration steps were required to upgrade to the last release.

New Filter (INSO_FILTER versus AUTO_FILTER)

With Oracle Text 11*g* Release 1 and later, the `INSO_FILTER` filter type has been replaced by the `AUTO_FILTER` filter type. To maintain compatibility with existing Oracle Text applications, `INSO_FILTER` functionality is supported by `AUTO_FILTER`, and the `INSO_FILTER` filter type is being retained, although its behavior has been changed to match that of `AUTO_FILTER`.

For most users, the change in filter types has no significant effect, and most users do not need to change their applications or otherwise take any steps to migrate to the new filter type. The most significant differences produced by this change are as follows:

- `AUTO_FILTER` supports a different set of document types than `INSO_FILTER`. For a list of supported document types, see the Supported File Formats appendix of *Oracle Text Reference*.
- Display characteristics of HTML output from the new filter may differ from those output by the old filter.

- A different set of platforms support the AUTO_FILTER type than support INSO_FILTER. For a list of supported platforms, see the Supported File Formats appendix of *Oracle Text Reference*.

Along with the change from INSO_FILTER to AUTO_FILTER, several related changes are also part of the filter migration:

- The INSO_TIMEOUT and INSO_OUTPUT_FORMATTING attributes of MAIL_FILTER have been replaced by the AUTO_FILTER_TIMEOUT and AUTO_FILTER_OUTPUT_FORMATTING attributes, respectively.
- The INSOFILTER directive used in the mail configuration file of MAIL_FILTER has been replaced with the AUTO_FILTER directive.
- The PL/SQL constant TRACE_IDX_INSO_FILTER for tracing, used in procedures such as CTX_OUTPUT.ADD_TRACE, has been replaced by the TRACE_IDX_AUTO_FILTER filter.
- The system-defined preference CTXSYS.INSO_FILTER has been replaced by the CTXSYS.AUTO_FILTER preference.
- The default values of DEFAULT_FILTER_FILE and DEFAULT_FILTER_BINARY system parameters has been changed from CTXSYS.INSO_FILTER to CTXSYS.AUTO_FILTER for new installations and databases upgrading to this release, if these default values have not been modified. For databases upgrading to this release which have modified these default values, the modified default values will continue to be used.

For backward compatibility, INSO_FILTER, as well as most of its associated filter types, constants, and attributes have been retained in this release; however, users should use AUTO_FILTER in new applications and update their older applications whenever possible.

Migrating to the AUTO_FILTER Filter Type

Most applications will not have to be modified to reflect the change to the AUTO_FILTER type. The following migration steps will be performed automatically when you upgrade to this release:

- If an existing index uses the INSO_FILTER filter type, then it will be migrated to the AUTO_FILTER filter type.
- If an existing index uses the MAIL_FILTER filter type, then the INSO_TIMEOUT and INSO_OUTPUT_FORMATTING attributes will be migrated to the AUTO_FILTER_TIMEOUT and AUTO_FILTER_OUTPUT_FORMATTING attributes, respectively.
- If the default values of the system parameters DEFAULT_FILTER_FILE and DEFAULT_FILTER_BINARY have not already been modified, then they will be migrated from CTXSYS.INSO_FILTER to CTXSYS.AUTO_FILTER.

Although no actions are *required* to migrate to the new filter system, Oracle Text users upgrading their databases to this release should take the following steps to migrate away from using APIs deprecated by this release:

- If an index uses the MAIL_FILTER filter type with a nondefault mail configuration file, edit the file, replacing all occurrences of the INSOFILTER directive with the AUTO_FILTER directive.
- Replace the use of the PL/SQL constant TRACE_IDX_INSO_FILTER in your application with the TRACE_IDX_AUTO_FILTER constant.

- Replace the use of the system-defined preference `CTXSYS.INSO_FILTER` in your applications with `CTXSYS.AUTO_FILTER`.

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 by 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://www.oracle.com/technology/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

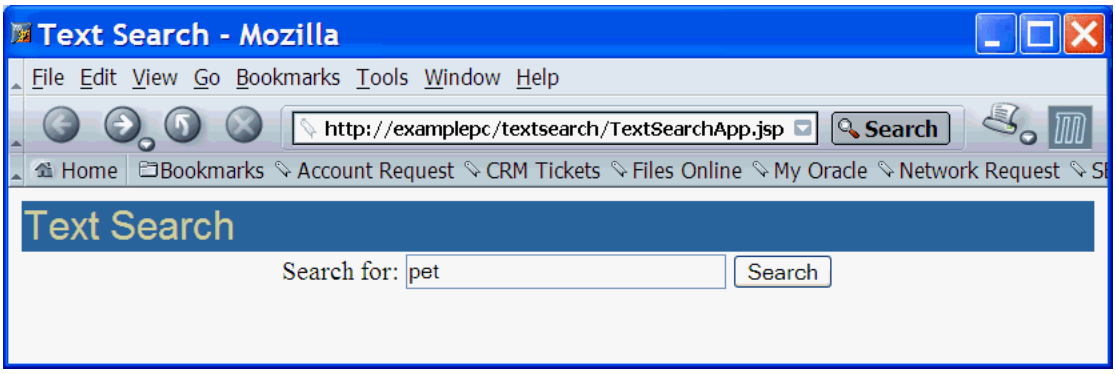
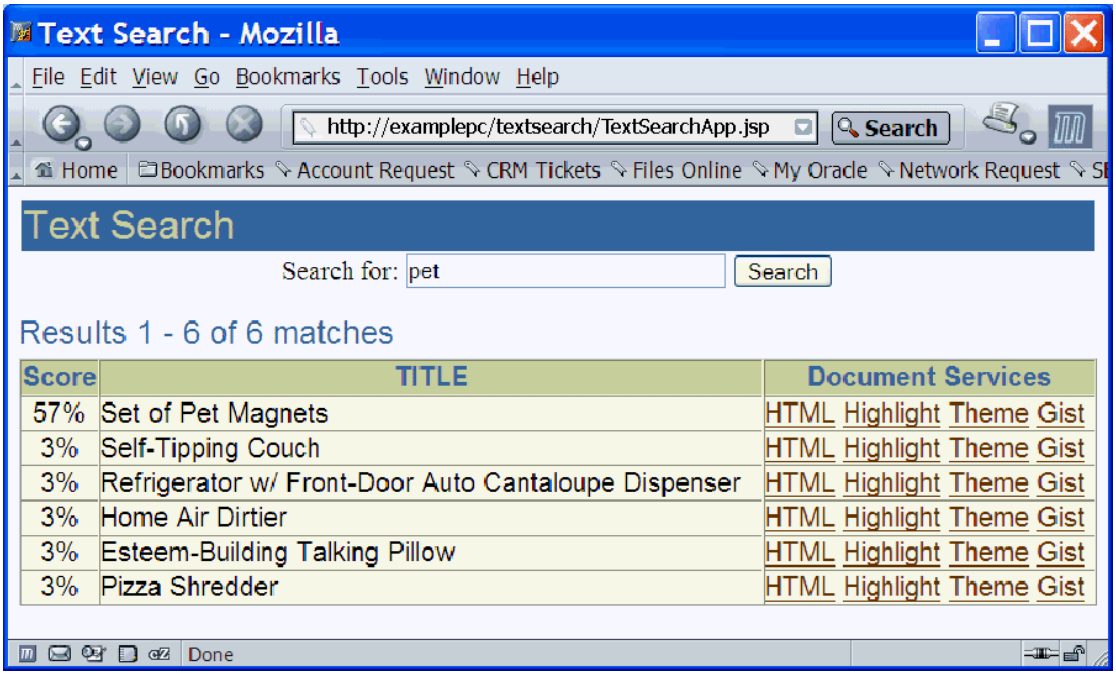


Figure A–2 shows the results of the text query.

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, "Sample Document for Highlighting, Gisting, and Theme Extraction" on page 5-7.)

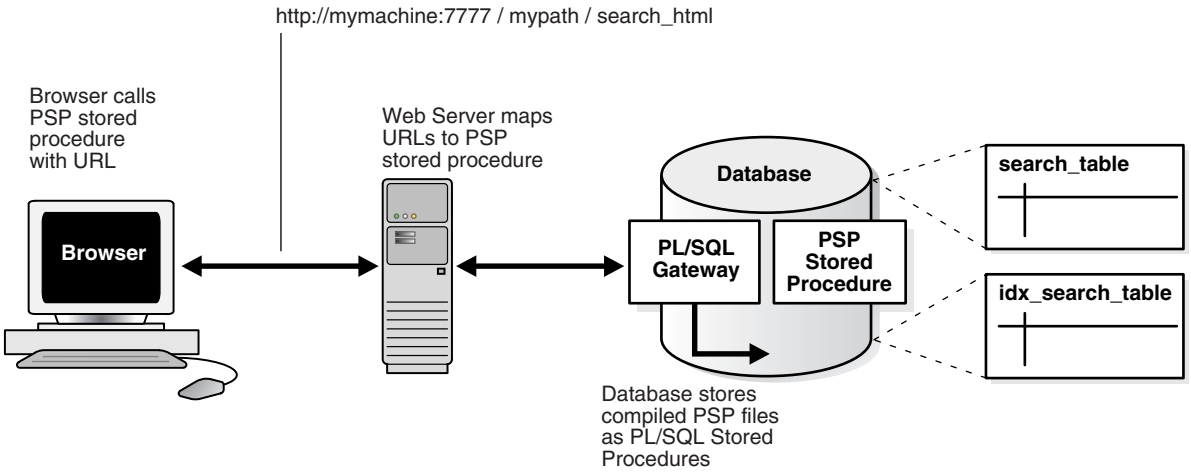
- The **Highlight** link displays the document with the search term highlighted. Figure 5–2, "Pet Highlighted in Pet Magnet Document" on page 5-8 shows an example of highlighting.

- The **Theme** link shows the top 50 themes associated with the document. Figure 5–3, "Query Application Displaying Document Themes" on page 5-9 shows an example of theme extraction.
- The **Gist** link displays a short summary of the document. Figure 5–4, "Query Application Presenting Document Gist" on page 5-10 shows an example of this gisting feature.

The PSP Web Application

This application is based on PL/SQL server pages. Figure A–3, "The PSP Web Application" 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 Oracle Database.

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 statement 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. Because 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 Advanced Application Developer's Guide* 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 Oracle Database and returns server output to the browser. See Figure A–3 on page A-3.

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 Enter 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://server.example.com: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-2.

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://www.oracle.com/technology/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 /*+ DOMAIN_INDEX_SORT */ 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-3). 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 Oracle Database.

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:@jsmith-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", "jsmith");
    info.put ("password","hello");
    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 /*+ DOMAIN_INDEX_SORT */ 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, either by writing your own code or by 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 using 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://www.oracle.com/technology/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 Oracle Database.

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

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:

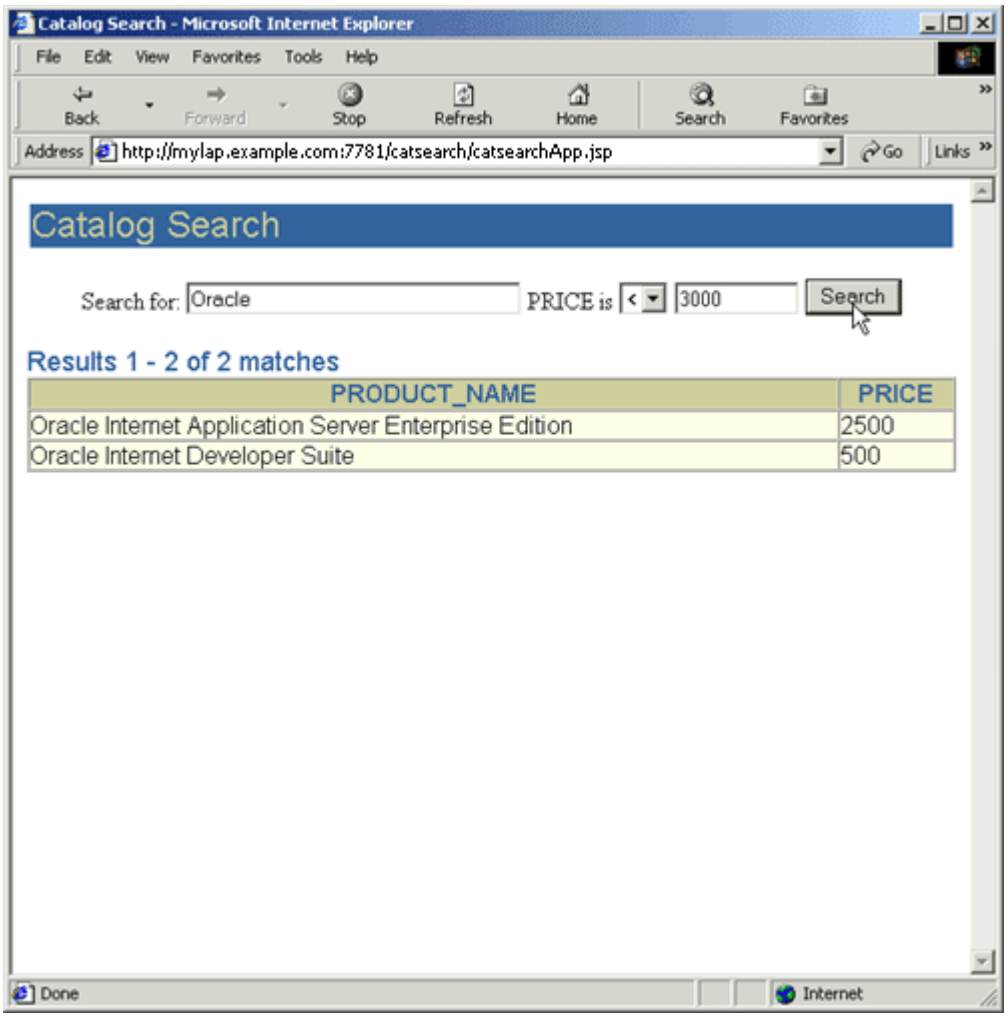
```
sqlplus>select id, title from book_catalog
        where catsearch(title,'Java','price > 10 order by price') > 0
```

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://www.oracle.com/technology/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>
<%
 }
%>
```

Glossary

alternate spelling

In Oracle Text, *alternate spelling* refers specifically to the use of spelling variations in German, Swedish, and Dutch; these variations may be indexed if the `BASIC_LEXER` attribute `ALTERNATE_SPELLING` has been specified.

attribute

An *attribute* is an optional parameter associated with a *preference*. For example, the `BASIC_LEXER` preference includes the `base_letter` attribute, which can have either the value of `YES` (perform base-letter conversions) or `NO` (do not perform such conversions). Attributes are set with the `CTX_DDL.SET_ATTRIBUTE` procedure or with the `ALTER INDEX` statement. See also: preference, base-letter conversion.

attribute section

A user-defined section, representing an attribute of an XML document, such as `AUTHOR` or `TITLE`. Attribute sections are added to section groups with `CTX_DDL.ADD_ATTR_SECTION` or with the `ALTER INDEX` statement. See also: AUTO_SECTION_GROUP, section, XML_SECTION_GROUP.

AUTO_SECTION_GROUP

A section group used to automatically crate a zone section for each start-and end-tag pair in an XML document; attribute sections are automatically created for XML tags that have attributes. See also: attribute section, section, section group, XML_SECTION_GROUP, zone section.

base-letter conversion

The conversion of a letter with alternate forms (such as accents, umlauts, or cedillas) to its basic form (for example, without an accent).

BASIC_SECTION_GROUP

A section group used to define sections where the start and end tags are of the form <*tag*> and </*tag*>. It does not support non-basic tags, such as comment tags or those with attributes or unbalanced parentheses. See also: HTML_SECTION_GROUP, section, section group.

case

Case refers to the capitalization of a word or letter, where upper-case letters are capitals (*M* instead of *m*, for example). Not all languages have case. Mixed-case indexing is supported for some languages, notably those of Western Europe.

classification

Also known as *document classification*. The conceptual separation of source documents into groups, or clusters, based on their content. For example, a group of documents might be separated into clusters concerning medicine, finance, and sports.

Oracle Text includes *rule-based classification*, in which a person writes the rules for classifying documents (in the form of queries), and Oracle Text performs the document classification according to the rules; *supervised classification*, in which Oracle Text creates classification rules based on a set of sample documents; and *clustering* (also known as *unsupervised classification*), in which the clusters and rules are both created by Oracle Text.

clustering

Also known as *unsupervised classification*. See: classification.

composite domain index

Also known as *CDI* type of index. An Oracle Text index that not only indexes and processes a specified text column, but also indexes and processes `FILTER BY` and `ORDER BY` structured columns that are specified during index creation. See also: domain index.

CONTEXT index

The basic type of Oracle Text index; an index on a text column. A `CONTEXT` index is useful when your source text consists of many large, coherent documents. Applications making use of `CONTEXT` indexes use the `CONTAINS` query operator to retrieve text.

CTXAPP role

A role for application developers that enables a user to create Oracle Text indexes and index preferences, and to use PL/SQL packages. This role should be granted to Oracle Text users.

CTXCAT index

A combined index on a text column and one or more other columns. Typically used to index small documents or text fragments, such as item names, prices and descriptions typically found in catalogs. The `CTXCAT` index typically has better mixed-query performance than the `CONTEXT` index.

Applications query this index with the `CATSEARCH` operator. This index is transactional, which means that it automatically updates itself with DML to the base table.

CTXRULE index

A `CTXRULE` index is used to build a document classification application. The `CTXRULE` index is an index created on a table of queries, where the queries serve as rules to define the classification criteria. This index is queried with the `MATCHES` operator.

CTXSYS user

The `CTXSYS` user is created at install time. The `CTXSYS` user can view all indexes; sync all indexes; run `ctxkbtc`, the knowledge base extension compiler; query all system-defined views; and perform all the tasks of a user with the `CTXAPP` role.

CTXXPATH index

An index used to speed up `existsNode()` queries on an `XMLType` column.

datastore

In Oracle Text, *datastore* refers to the method of storing text. The method is determined by specifying a storage preference of a particular type. For example, the `DIRECT_DATASTORE` type stores data directly into the text column, while the `URL_DATASTORE` specifies that data is stored externally in a location specified by a URL.

domain index

An Oracle Database domain index that indexes and processes a specified text column. See also: composite domain index.

endjoin

One or more non-alphanumeric characters that, when encountered as the last character in a token, explicitly identify the end of the token. The characters, as well as any *startjoin* characters that immediately follow it, are included in the Text index entry for the token. For example, if ++ is specified as an endjoin, then *C*++ will be recognized and indexed as a single token. See also: printjoin, skipjoin, startjoin.

entity extraction

Entity extraction is the process of locating phrases in an input document and classifying these phrases into a set of categories. This information can then be used to better understand the content of the document. Entities can be concrete objects, such as people or companies, or more abstract objects, such as percentages.

field section

A field section is similar to a zone section, with the main difference that the content between the start and end tags of a field section can be indexed separately from the rest of the document. This enables field section content to be "hidden" from a normal query. (The `INPATH` and `WITHIN` operators may be used to find the term in such a section.) Field sections are useful when there is a single occurrence of a section in a document, such as a filed in a news header. Field sections are added to section groups with the `CTX_DDL.ADD_FIELD_SECTION` procedure or with the `ALTER INDEX` statement. See also: INPATH operator, section, WITHIN operator, zone section.

filtering

One of the steps in the Oracle Text index-creation process. Depending on the filtering preferences associated with the creation of the index, one of three things happens during filtering: Formatted documents are filtered into marked-up text; text is converted from a non-database character set to a database character set; or no filtering takes place (HTML, XML, and plain-text documents are not filtered).

fuzzy matching

A *fuzzy*-matching query is one in which the query is expanded to include words that are spelled similarly to the specified term. This type of expansion is helpful for finding more accurate results when there are frequent misspellings in a document set. Fuzzy matching is invoked with the `FUZZY` query operator.

HASPATH operator

A `CONTAINS` query operator used to find XML documents that contain a section path exactly as specified in the query. See also: PATH_SECTION_GROUP.

highlighting

Generically, in Oracle Text, *highlighting* refers to generating a version of a document, or document fragments, with query terms displayed or called out in a special way.

Specifically, there are three forms of highlighting. First, CTX_DOC.MARKUP returns a document with the query term surrounded by plaintext or HTML tags. Second, CTX_DOC.HIGHLIGHT returns offsets for the query terms, allowing the user to mark up the document as desired. Third, CTX_DOC.SNIPPET produces a concordance, with the query term displayed in fragments of surrounding text. markup.

HTML_SECTION_GROUP

A section group type used for defining sections in HTML documents. See also: BASIC_SECTION_GROUP, section, section group.

INPATH operator

A CONTAINS query operator used to search within tags, or *paths*, of an XML document. It enables more generic path denomination than the WITHIN operator. See also: WITHIN operator.

Key Word in Context (KWIC)

In Oracle Text, a presentation of a query term with the text that surrounds it in the source document. This presentation may consist of a single instance of the query term, several instances, or every instance in the source document. The CTX_DOC.SNIPPET procedure produces such a presentation. Also known as Key Word in Context (KWIC).

knowledge base

Oracle Text includes a *knowledge base*, which is a hierarchical tree of concepts used for theme indexing, ABOUT queries, and deriving themes for document services. The knowledge base may be optionally installed. You can create your own knowledge base or extend the standard Oracle Text knowledge base.

lexer

The Oracle Text *lexer* breaks source text into tokens—usually words—in accordance with a specified language. To extract tokens, the lexer uses parameters as defined by a lexer *preference*. These parameters include the definitions for the characters that separate tokens, such as whitespace, and whether to convert text to all uppercase or not. When theme indexing is enabled, the lexer analyses text to create theme tokens.

When an application needs to index a table containing documents in more than one language, it can utilize the MULTI_LEXER (the multilingual lexer) and create *sub-lexers* to handle each language. Each sub-lexer is added to the main multi-lexer with the CTX_DDl.ADD_SUB_LEXER procedure.

markup

A form of highlighting. The CTX_DOC.MARKUP and CTX_DOC.POLICY_MARKUP procedures take a query term and a document, and return the document with the query terms marked up; that is, surrounded either by plaintext characters or HTML tags. You can use predefined markup tags or specify your own. In comparison, CTX_DOC.HIGHLIGHT and CTX_DOC.POLICY_HIGHLIGHT return offsets for query terms, so you can add your own highlighting tags. See also: highlighting.

MDATA

See: metadata.

MDATA section

An MDATA section contains user-defined index metadata. Use of this metadata can speed up mixed CONTAINS queries. See also: metadata, mixed query, section.

metadata

Metadata is information about a document that is not part of a document's regular content. For example, if an HTML document contains `<author>Mark Twain</author>`, *author* is considered the metadata type and *Mark Twain* is considered the value for `author`.

Sections containing metadata, known as `MDATA` sections, can be added to a document with the `CTX_DDL.ADD_MDATA_SECTION` procedure. Taking advantage of metadata can speed up mixed queries. Such queries can be made with the `MDATA` operator. See also: mixed query, section.

mixed query

A query that searches for two different types of information; for example, text content and document type. For example, a search for *Romeo and Juliet* in `<title>` metadata is a mixed query.

name search

Name searching (also called name matching) provides a solution to match proper names that might differ in spelling due to orthographic variation. It also enables you to search for somewhat inaccurate data, such as might occur when a record's first name and surname are not properly segmented.

NEWS_SECTION_GROUP

A section group type used for defining sections in newsgroup-formatted documents as defined by RFC 1036. See also: section, section group.

normalized word

The form of a word after it has been transformed for indexing, according to transformational rules in effect. Depending on the rules in effect, the normalized form of a word may be the same as the form found in the source document. The normalized form of a word may also include both the original and transformed versions. For example, if New German Spelling has been specified, the word *Potential* is normalized to both *Potenzial* and *Potential*.

NULL_SECTION_GROUP

The default section group type when no sections are defined or when only `SENTENCE` or `PARAGRAPH` sections are defined. See also: section, section group, special section.

PATH_SECTION_GROUP

A section group type used for indexing XML documents. It is similar to the `AUTO_SECTION_GROUP` type, except that it enables the use of the `HASPATH` and `INPATH` operators. See also: AUTO_SECTION_GROUP, HASPATH operator, INPATH operator, section, section group.

preference

A *preference* is an optional parameter that affects the way Oracle Text creates an index. For example, a lexer preference specifies the lexer to use when processing documents, such as the `JAPANESE_VGRAM_LEXER`. There are preferences for storage, filtering, lexers, classifiers, wordlist, section types, and more. A preference may or may not have *attributes* associated with it. Preferences are set with the `CTX_DDL.CREATE_PREFERENCE` procedure. See also: attribute.

printjoin

One or more non-alphanumeric character that, when they appear anywhere in a word (beginning, middle, or end), are processed as alphanumeric and included with the token in an Oracle Text index. This includes printjoins that occur consecutively.

For example, if the hyphen (-) and underscore (_) characters are defined as printjoins, terms such as *pseudo-intellectual* and *_file_* are stored in the Oracle Text index as *pseudo-intellectual* and *_file_*.

Printjoins differ from endjoins and startjoins in that position does not matter. For example, *$35* will be indexed as one token if *$* is defined as a startjoin or a printjoin, but as two tokens if it is an endjoin. See also: endjoin, printjoin, startjoin.

result set

A page of search results in applications can consist of many disparate elements — metadata of the first few documents, total hit counts, per-word hit counts, and so on. Generating these results in earlier versions of Oracle Text required several queries and calls. Each extra call takes time to reparse the query and look up index metadata. Additionally, some search operations, such as iterative query refinement or breakdown top ten, are difficult for SQL.

The result set interface enables you to produce the various kinds of data needed for a page of search results all at once, thus improving performance by sharing overhead. The result set interface can also return data views that are difficult to express in SQL, such as top *N* by category queries.

rule-based classification

See: classification.

SDATA section

Structured/Sort Data section. Unlike the `MDATA` section type, which only supports equality searches, `SDATA` sections are designed to also support range searches. By default, all `FILTER BY` and `ORDER BY` columns are mapped as `SDATA` sections. An `SDATA` section contains user-defined index metadata. Use of this type of section can speed up mixed `CONTAINS` queries. See also: mixed query, section.

section

A *section* is a subdivision of a document; for example, everything within an `<a>...</a>` section of an HTML page.

Dividing a document into sections and then searching within sections 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.

Section searching is performed with the `HASPATH`, `ISPATH`, or `WITHIN` operator. Sections searching is enabled by the used of the section group when indexing.

The various section types include attribute, field, HTML, MDATA, special, stop, XML, and zone sections.

section group

A section group identifies a type of document set and implicitly indicate the tag structure for indexing. For instance, to index HTML tagged documents, you use the `HTML_SECTION_GROUP`. section group type. Likewise, to index XML tagged documents, you can use the `XML_SECTION_GROUP` section group type. Section groups

are declared with the CTX_DDL.CREATE_SECTION_GROUP procedure or with the ALTER INDEX statement. See also: section.

skipjoin

A non-alphanumeric character that, when it appears within a word, identifies the word as a single token; however, the character is not stored with the token in the Text index. For example, if the hyphen character '-' is defined as a skipjoin, the word *pseudo-intellectual* is stored in the Text index as *pseudointellectual*. See also: endjoin, printjoin, startjoin.

startjoin

One or more non-alphanumeric characters that, when encountered as the first character in a token explicitly identify the start of the token. The characters, as well as any other startjoins characters that immediately follow it, are included in the Text index entry for the token. For example, if '*$*' is defined as a startjoin, then *$35* is indexed as a single token. In addition, the first startjoins character in a string of startjoins characters implicitly ends the previous token. See also: endjoin, printjoin, skipjoin.

stemming

The expansion of a query term to include all terms having the same root word. For example, stemming the verb *talk* yields *talking*, *talks*, and *talked*, as well as *talk* (but not *talkie*). Stemming is distinct from wildcard expansion, in which results are related only through spelling, not through morphology. See also: wildcard expansion.

special section

A document section that is not bounded by tags. Instead, sections are formed by plaintext document structures such as sentences and paragraphs. Special sections are added to a section group with the CTX_DDL.ADD_SPECIAL_SECTION procedure. See also: section, section group.

stop section

A section that, when added to an AUTO_SECTION_GROUP, causes the information for document sections of that type to be ignored during indexing; the section content may still be searched, however. Stop sections are added to section groups with the CTX_DDL.ADD_STOP_SECTION procedure. See also: AUTO_SECTION_GROUP, section, section group.

stopclass

A class of tokens, such as NUMBERs, that are to be skipped over during indexing. Stopclasses are specified by adding them to *stoplists* with CTX_DDL.ADD_STOPCLASS. See also: stoplist.

stoplist

A list of words, known as *stopwords*, themes (*stopthemes*), and data classes (*stopclasses*) that are not to be indexed. By default, the system indexes text using the system-supplied stoplist that corresponds to a given database language.

Oracle Text provides default stoplists for most common languages including English, French, German, Spanish, Chinese, Dutch, and Danish. These default stoplists contain only stopwords. Stoplists are created with CTX_DDL.CREATE_STOPLIST or with the ALTER INDEX statement. See also: stopclass, stoptheme, stopword.

stoptheme

A theme to be skipped over during indexing. Stopthemes are specified by adding them to *stoplists* with `CTX_DDL.ADD_STOPTHEMES`. See also: stoplist.

stopword

A word to be skipped over during indexing. Stopwords are specified by adding them to *stoplists* with `CTX_DDL.ADD_STOPWORD`. They can also be dynamically added to an index using the `ALTER INDEX` statement. See also: stoplist.

sub-lexer

See: lexer.

supervised classification

See: classification.

theme

A topic associated with a given document. A document may have many themes. A theme does not have to appear in a document; for example, a document containing the words *San Francisco* may have *California* as one of its themes.

Theme components are added to indexes with the `INDEX_THEMES` attribute of the `BASIC_LEXER` preference; they may be extracted from a document with `CTX_DOC.THEMES` and queried with the `ABOUT` operator.

unsupervised classification

Also known as clustering. See: classification.

wildcard expansion

The expansion of a query term to return words that fit a given pattern. For example, expansion of the query term *%rot%* would return both *trot* and *rotten*. Wildcard expansion is distinct from stemming. See also: stemming.

whitespace

Characters that are treated as blank spaces between tokens. The predefined default values for whitespace are 'space' and 'tab'. The `BASIC_LEXER` uses whitespace characters (in conjunction with punctuations and newline characters) to identify character strings that serve as sentence delimiters for sentence and paragraph searching.

WITHIN operator

A `CONTAINS` query operator used to search for query terms within a given XML document section. It is similar to the `INPATH` operator, but less generic. See also: INPATH operator.

wordlist

An Oracle Text preference that enables features such as fuzzy, stemming, and prefix indexing for better wildcard searching, as well as substring and prefix indexing. The wordlist preference improves performance for wildcard queries with `CONTAINS` and `CATSEARCH`. Create wordlists with the `CTX_DDL.ADD_WORDLIST` procedure or with the `ALTER INDEX` statement. See also: preference.

XML section

A section that defined by XML tags, enabling XML section searching. Indexing with XML sections permits automatic sectioning as well as declaring document-type-sensitive sections. XML section searching includes attribute searching

as well as path section searching with the INPATH, HASPATH, and WITHIN operators. See also: section.

XML_SECTION_GROUP

A section group used for identifying XML documents for indexing. See also: section, section group.

zone section

The basic type of document section; a body of text delimited by start and end tags in a document. Zone sections are well suited for defining sections in HTML and XML documents. Zone sections are added to section groups with the CTX_DDL.ADD_ZONE_SECTION procedure or with the ALTER INDEX statement. See also: field section, section, section group.

Index

A

ABOUT query, 4-16
  adding for your language, 10-8
  case-sensitivity, 4-10
  definition, 4-8
accents
  indexing characters with, 3-17
ACCUM operator, 4-17
ADD_STOPCLASS procedure, 3-25
ADD_STOPTHEME procedure, 3-25
ADD_STOPWORD procedure, 3-24, 3-25
ADD_SUB_LEXER procedure
  example, 3-23
administration tool, 13-5
ALTER INDEX statement
  rebuilding index, 3-35
  resuming failed index, 3-33
alternate spelling, 3-17
alternative grammar, 4-14
alternative grammar template, 4-14
alternative scoring, 4-13
alternative scoring template, 4-13
AND operator, 4-17
application
  sample, A-1, B-1
applications, updating, 14-1
attribute
  searching XML, 8-13
attribute sections, 8-11
AUTO_FILTER filter, 3-6, 3-12, 3-13, 7-18
AUTO_LEXER
  indexing columns and, 3-14
  language detection and, 3-14
AUTO_LEXER lexer type, 3-16
AUTO_LEXER with fuzzy matching and stemming, 3-18
AUTO_SECTION_GROUP object, 8-2
automatic sections, 8-13

B

background DML, 13-7
base-letter conversion, 3-17
BASIC_LEXER, 3-14
BASIC_SECTION_GROUP object, 8-2
BFILE column, 3-10
  indexing, 3-26
BINARY format column value, 3-13
BLOB column, 3-10
  indexing, 3-26
blocking operations
  tuning queries with, 7-10
bypassing rows, 3-13

C

cantaloupe dispenser, A-2
case-sensitive
  ABOUT query, 4-10
  indexing, 3-16
  queries, 4-9
  thesaurus, 10-2
catalog application, 2-5
  example, 2-5
CATSEARCH, 4-3
  creating index for, 3-29
  operators, 4-21
  SQL example, 4-3
CATSEARCH queries, 2-6
CHAR column, 3-10
Character Large Object (CLOB), 2-3
character set
  indexing, 3-13
  indexing mixed, 3-14
character set column, 3-11
charset column, 3-14
CHARSET_FILTER, 3-6, 3-13
Chinese indexing, 3-18
CHINESE_VGRAM_LEXER, 3-18
classification
  Decision Tree (supervised), 6-7
  rule-based, 6-3
  simple, see rule-based classification
  supervised, 6-7
  SVM (supervised), 6-10
  unsupervised
classification application
  example, 2-7
classifying phrases, 12-1
CLOB (Character Large Object) datatype, 2-3
CLOB column, 3-10

indexing, 3-26
clustering, see unsupervised classification
column types
supported for indexing, 3-10
composite words
indexing, 3-17
concordance, 5-3
CONTAINS
operators, 4-15
PL/SQL example, 4-2
query, 4-1
SQL example, 4-1
structured query, 4-2
CONTAINS query, 2-3
CONTEXT grammar, 4-15
CONTEXT index
about, 3-2
creating, 3-20, 3-25
HTML example, 2-3, 3-27, A-4
couch, self-tipping, A-2
counting hits, 4-20
CREATE_INDEX_SCRIPT, 13-3
CREATE_POLICY_SCRIPT, 13-3
CREATE_STOPLIST procedure, 3-24, 3-25
CTX_CLS.TRAIN procedure, 6-7
CTX_DDL.SYNC_INDEX procedure, 2-4
CTX_DOC.POLICY_SNIPPET procedure, 5-3
CTX_DOC.SNIPPET procedure, 5-3
CTX_INDEX_ERRORS view, 3-32, 13-2
CTX_OUTPUT.END_QUERY_LOG, 4-14
CTX_OUTPUT.START_QUERY_LOG, 4-14
CTX_PENDING view, 13-2
CTX_QUERY.RESULT_SET procedure, 11-1
CTX_REPORT, 3-38
CTX_REPORT package, 13-2
CTX_REPORT_QUERY_LOG_SUMMARY, 4-14
CTX_REPORT_TOKEN_TYPE, 13-5
CTX_REPORT.CREATE_INDEX_SCRIPT, 13-3
CTX_REPORT.CREATE_POLICY_SCRIPT, 13-3
CTX_REPORT.DESCRIBE_INDEX, 13-2
CTX_REPORT.DESCRIBE_POLICY, 13-2
CTX_REPORT.INDEX_SIZE, 13-4
CTX_REPORT.INDEX_STATS, 13-5
CTX_REPORT.QUERY_LOG_SUMMARY, 13-5
CTX_REPORT.TOKEN_INFO, 13-5
CTX_THES package
about, 10-1
CTX_USER_INDEX_ERRORS view, 3-32, 13-2
CTX_USER_PENDING view, 13-2
CTXAPP role, 2-1, 13-1
CTXCAT grammar, 4-21
CTXCAT index
about, 3-3
about performance, 7-16
automatic synchronization, 2-7
creating, 2-5
example, 3-28
ctxkbtc
example, 10-7
ctxload
load thesaurus example, 10-2, 10-5, 10-7
CTXRULE index, 6-6
about, 3-3
allowable queries, 6-6
creating, 2-8, 3-30
lexer types, 6-6
limitations, 6-6
parameters, 6-6
CTXSYS user, 13-1
CTXXPATH index, 1-7
about, 3-4

D

data storage
index default, 3-26
preference example, 3-22
datastore
about, 3-6, 3-21
DATE column, 3-26
Decision Tree supervised classification, 6-7
default thesaurus, 10-3
DEFAULT_INDEX_MEMORY, 7-17
defaults
index, 3-26
DEFINEMERGE operator, 4-21
DEFINESCORE operator, 4-21
DESCRIBE_INDEX, 13-2
DETAIL_DATASTORE, 3-11
about, 3-12
diacritical marks
characters with, 3-17
DIRECT_DATASTORE, 3-11
about, 3-12
example, 3-21
DML
view pending, 3-36
DML processing
background, 13-7
DML queue, 13-2
document
classification, 3-30, 6-1
document format
affect on index performance, 7-18
affect on performance, 7-12
document formats
filtering, 3-12
supported, 3-12
document invalidation, 3-38
document presentation
about, 5-6
document sections, 3-24
document services
about, 5-6
DOMAIN_INDEX_NO_SORT hint
better throughput example, 7-7
DOMAIN_INDEX_SORT hint
better response time example, 7-5
DROP INDEX statement, 3-32
DROP_STOPLIST procedure, 3-25

dropping an index, 3-32

E

entity extraction, 12-1
EQUIV operator, 4-17
errors
  DML, 13-2
  viewing, 3-32
explain plan, 4-10
exporting statistics, 7-1
extensible query optimizer, 7-1
extraction
  entity, 12-1

F

feedback
  query, 4-10
field section
  definition, 8-6
  nested, 8-6
  repeated, 8-7
  visible and invisible, 8-6
file paths
  storing, 3-11
FILE_DATASTORE, 3-6
  about, 3-11, 3-12
  example, 3-22
filter
  about, 3-6, 3-21
filtering
  custom, 3-13
  index default, 3-26
  to plain text and HTML, 5-6
filtering documents, 3-12
FIRST_ROWS hint, 4-19
  better throughput example, 7-7
FIRST_ROWS(n) hint, 7-4
format column, 3-11, 3-13
formats
  filtering, 3-12
  supported, 3-12
fragmentation of index, 3-38, 7-19
  viewing, 3-38
full themes
  obtaining, 5-4
functional lookup, 7-12
fuzzy matching, 3-18
  default, 3-26
fuzzy operator, 4-18

G

garbage collection, 3-38
German
  alternate spelling, 3-17
  composite words, 3-17
gist
  definition, 5-3
  example, 5-5
GIST procedure, 5-4
grammar
  alternative, 4-14
  CTXCAT, 4-21
grammar CONTEXT, 4-15
granting roles, 2-2, 13-2

H

HASPATH operator, 8-15
  examples, 8-17
HFEEDBACK procedure, 4-10
highlighting
  about, 5-6
  overview, 5-1
highlighting documents, 2-3
highlighting text, 5-1
highlighting themes, 5-1
hit count, 4-20
home air dirtier, A-2
HTML
  filtering to, 5-6
  indexing, 3-22, 8-2
  indexing example, 2-3, A-4
  searching META tags, 8-12
  zone section example, 3-24, 8-12
HTML_SECTION_GROUP object, 3-24, 8-2, 8-12
  with NULL_FILTER, 2-3, 3-22, A-4

I

IGNORE
  format column value, 3-13
IGNORE format column value, 3-13
importing statistics, 7-1
index
  about, 3-1
  creating, 3-20
  dropping, 3-32
  fragmentation, 3-38
  getting report on, 13-2
  incrementally creating, 3-26
  maintenance, 3-32
  online recreation, 3-33
  optimizing, 3-37, 3-39
  rebuilding, 3-35
  statistics on, 13-5
  structure, 3-5, 3-38
  synchronizing, 3-36, 13-7
  viewing information on, 13-2
index defaults
  general, 3-26
index engine
  about, 3-7
index errors
  viewing, 3-32
index fragmentation, 7-19
index memory, 7-17
index synchronization, 2-4
index types

choosing, 3-1
INDEX_SIZE, 13-4
INDEX_STATS, 13-5
INDEX_STATS procedure, 3-38
indexed lookup, 7-12
indexing
and views, 3-8
bypassing rows, 3-13
considerations, 3-9
overview of process, 3-5
parallel, 3-8, 7-18
resuming failed, 3-33
special characters, 3-15
indexing performance
FAQs, 7-16
parallel, 7-18
indexing time, 7-16
INPATH operator, 8-15
examples, 8-15
INSO_FILTER (deprecated), 14-1
INSO_OUTPUT_FORMATTING attribute (deprecated), 14-1
INSO_TIMEOUT attribute (deprecated), 14-1
INSOFILTER directive (deprecated), 14-1

J

Japanese indexing, 3-18
JAPANESE_LEXER, 3-18
Jdeveloper
Text wizard, 2-4, A-1, B-1

K

knowledge base
about, 10-7
augmenting, 10-5
linking new terms, 10-6
supported character set, 10-8
user-defined, 10-8
Korean indexing, 3-18
KOREAN_MORPH_LEXER, 3-18

L

language
default setting for indexing, 3-26
language detection
AUTO_LEXER, 3-16
language specific features, 3-16
languages
indexing, 3-14
language-specific feature
AUTO_LEXER, 3-16
language-specific knowledge base, 10-8
lexer
about, 3-7, 3-21
and CTXRULE, 6-6
list of themes
definition, 5-3
obtaining, 5-4
loading text
about, 3-9
LOB columns
improving query performance, 7-13
indexing, 3-26
local partitioned index, 7-15
improved response time, 7-5
location of text, 3-9
locking parameter for sync_index, 3-37
logical operators, 4-16

M

magnet, pet see pet magnet
maintaining the index, 3-32
marked-up document
obtaining, 5-2
MARKUP procedure, 2-3, 5-2
MATCHES
about, 4-4
PL/SQL example, 3-31, 4-6
SQL example, 4-4
MATCHES operator, 2-9, 6-6
materialized views, indexes on
MAX_INDEX_MEMORY, 7-17
maxtime parameter for sync_index, 3-37
MDATA operator, 8-7
MDATA section, 8-7
memory allocation
index synchronization, 7-20
indexing, 7-17
querying, 7-13
META tag
creating zone section for, 8-12
metadata
adding, 8-7
removing, 8-7
section, 8-7
migrating from previous releases, 14-1
mixed formats
filtering, 3-13
mixed query, 8-7, 8-10
MULTI_COLUMN_DATASTORE, 3-11
about, 3-12
example, 3-22
MULTI_LEXER, 3-15
example, 3-23
multi-language columns
indexing, 3-15
multi-language stoplist
about, 3-25
multiple CONTAINS
improving performance, 7-14
MVIEW see materialized views

N

name matching, 9-1
name search, 9-1
NCLOB column, 3-26

NEAR operator, 4-17
NEAR_ACCUM operator, 4-17
nested zone sections, 8-5
NESTED_DATASTORE, 3-11
  about, 3-12
NEWS_SECTION_GROUP object, 8-2
NOPOPULATE keyword
  and incremental rebuild,and replace
    parameter, 3-26
nopopulate with RECREATE_INDEX_
  ONLINE, 3-34
NOT operator, 4-17
NULL_FILTER, 3-6
  example, 2-3, 3-22, A-4
NULL_SECTION_GROUP object, 8-2
NUMBER column, 3-26

O

offset information
  highlight, 5-3
online
  recreating a CONTEXT indextype, 3-33
operator
  MDATA, 8-7
  SDATA, 8-10
operators
  CATSEARCH, 4-21
  CONTAINS, 4-15
  logical, 4-16
  thesaurus, 10-1
optimizing index, 3-37
  example, 3-39
  single token, 3-38
optimizing queries, 4-19, 7-1
  FAQs, 7-11
  response time, 7-4
  statistics, 7-1
  throughput, 7-7
  with blocking operations, 7-10
OR operator, 4-17
ora
  contains, 1-6
Oracle Enterprise Manager, 13-5
Oracle Enterprise Manager and Oracle Text, 13-5
Oracle Text pages in OEM, 13-5
Oracle XML DB, 1-5
out of line LOB storage
  improving performance, 7-13

P

parallel indexing, 3-8, 7-18
  partitioned table, 7-18
parallel queries, 7-9, 7-15
  across Real Application Clusters (RAC)
    nodes, 7-9
partitioned index, 7-15
  improved response time, 7-5
path section searching, 8-15
PATH_SECTION_GROUP
  example, 8-15
pending DML
  viewing, 3-36
pending updates, 13-2
performance tuning
  indexing, 7-16
  querying, 7-11
  updating index, 7-19
pet magnet, A-2
  gist, 5-10
  illustration, 5-7
  themes, 5-9
phrase query, 4-7
pizza shredder, A-2
plain text
  indexing with NULL_FILTER, 3-22
plain text filtering, 5-6
PL/SQL functions
  calling in contains, 4-19
POPULATE_PENDING, 3-27
preferences
  creating (examples), 3-21
  creating with admin tool, 13-5
  dropping, 3-35
previous releases, migrating from, 14-1
printjoins character, 3-15
PROCEDURE_FILTER, 3-13
PSP application, A-3, B-1

Q

query
  ABOUT, 4-16
  analysis, 4-14
  blocking operations, 7-10
  case-sensitive, 4-9
  CATSEARCH, 4-3
  CONTAINS, 4-1
  counting hits, 4-20
  CTXRULE, 6-6
  getting report on, 13-2
  log, 4-14
  MATCHES, 4-4
  mixed, 8-7, 8-10
  optimizing for throughput, 7-7
  overview, 4-1
  parallel, 7-9
  speeding up with MDATA, 8-7
  speeding up with SDATA, 8-10
  viewing information on, 13-2
  viewing log of, 13-5
query analysis, 4-14
query application
  example, 2-2
  sample, 1-2
query explain plan, 4-10
query expressions, 4-9
query features, 4-14
query feedback, 4-10

query language, 4-13
query log, 4-14, 13-5
query optimization, 4-19
  FAQs, 7-11
  response time, 7-4
Query Parallelized Across Oracle RAC Nodes, 7-10
query performance
  FAQs, 7-11
query relaxation, 4-12
query relaxation template, 4-12
query rewrite, 4-11
query rewrite template, 4-12
query template, 4-18, 4-21
  lang parameter and, 4-13
Query Templates, 4-11
QUERY_LOG_SUMMARY, 13-5
queue
  DML, 13-2

R

Real Application Clusters (RAC) and parallel queries, 7-9
rebuilding an index, 3-35
RECREATE_INDEX_ONLINE, 3-33
  and DML, 3-33
recreating a local partitioned index online, 3-34
recreating an index, 3-33
recreating an index online, 3-33
relaxing queries, 4-12
REMOVE_SQE procedure, 4-18
REMOVE_STOPCLASS procedure, 3-25
REMOVE_STOPTHEME procedure, 3-25
REMOVE_STOPWORD procedure, 3-25
response time
  improving, 7-4
  optimizing for, 4-19
result buffer size
  increasing, 7-10
result set interface, 11-1
result sets, 11-1
resuming failed index, 3-33
rewriting queries, 4-11
roles
  granting, 2-2, 13-2
  system-defined, 13-1
rule-based classification, 6-3

S

sample application, A-1, B-1
scoring
  alternative, 4-13
SDATA operator, 8-10
SDATA section, 8-10
searching
  XML, 1-5
section
  attribute, 8-11
  field, 8-6
  groups and types, 8-4
  HTML example, 3-24
  MDATA, 8-7
  nested, 8-5
  overlapping, 8-5
  repeated zone, 8-5
  SDATA, 8-10
  special, 8-11
  stop section, 8-7
  types and groups, 8-4
  zone, 8-4
section group
  about, 3-21
  and section types, 8-4
  creating with admin tool, 13-5
section searching, 4-17
  about, 4-11, 8-1
  enabling, 8-1
  HTML, 8-12
sectioner
  about, 3-6
sectioning
  automatic, 8-13
  path, 8-15
self-tipping couch, A-2
SGA memory allocation, 7-17
simple classification, see rule-based classification
single themes
  obtaining, 5-4
size of index, viewing, 13-4
skipjoins character, 3-15
SORT_AREA_SIZE, 7-10, 7-13, 7-17
special characters
  indexing, 3-15
special sections, 8-11
spelling
  alternate, 3-17
  searching different, 9-1
SQE operator, 4-18
statistics
  exporting and importing, 7-1
  optimizing with, 7-2
stem operator, 3-18, 4-18
stemming
  default, 3-26
  improving performance, 7-14
stop section, 8-7
stopclass, 3-25
stoplist, 3-24
  about, 3-21
  creating with admin tool, 13-5
  default, 3-26
  multi-language, 3-19, 3-25
  PL/SQL procedures, 3-25
stoptheme, 3-25
  about, 3-19
  definition, 4-8
stopword, 3-24, 3-25
  about, 3-19, 4-8
  case-sensitive, 4-10

storage
about, 3-21
STORE_SQE procedure, 4-18
stored query expression, defining, 4-18
stored query expressions, 4-18
storing text, 3-9
about, 3-10
structure of index, 3-38
structured data
adding, 8-10
removing, 8-10
section, 8-10
structured query
example, 3-28
supervised classification, 6-7
Decision Tree, 6-7
SVM supervised classification, 6-10
memory requirements, 6-11
swap and noswap with RECREATE_INDEX_ ONLINE, 3-34
SYN operator, 10-4
sync_index locking parameter, 3-37
sync_index maxtime parameter, 3-37
SYNC_INDEX procedure, 2-4
synchronize index, 2-4
synchronizing index, 3-36, 13-7
improving performance, 7-19
synonyms
defining, 10-4

T

talking pillow, A-2
template queries, 4-18, 4-21
templates, 4-11
query rewrite, 4-12
text column
supported types, 3-10
TEXT format column value, 3-13
text highlighting, 5-1
text storage, 3-9
theme functionality
adding, 10-8
theme highlighting, 5-1
theme summary
definition, 5-3
themes
indexing, 3-16
THEMES procedure, 5-4
thesaural queries
about, 4-11
thesaurus
about, 10-1
adding to knowledge base, 10-5
case-sensitive, 10-2
DEFAULT, 10-3
default, 10-3
defining terms, 10-4
hierarchical relations, 10-4
loading custom, 10-5
operators, 10-1
supplied, 10-3
using in application, 10-4
thesaurus operator, 4-18
throughput
improving query, 7-7
tildes
indexing characters with, 3-17
TOKEN_INFO, 13-5
TOKEN_TYPE, 13-5
tracing, 7-8
TRAIN procedure, 6-7
tuning queries
for response time, 7-4
for throughput, 7-7
increasing result buffer size, 7-10
with statistics, 7-1

U

umlauts
indexing characters with, 3-17
unsupervised classification, 6-12
updating index performance
FAQs, 7-19
updating your applications, 14-1
URL_DATASTORE
about, 3-12
example, 3-22
URLs
storing, 3-11
user
creating Oracle Text, 2-1
system-defined, 13-1
USER_DATASTORE, 3-8
about, 3-12
USER_FILTER, 3-13

V

VARCHAR2 column, 3-10
viewing information on indexes and queries, 13-2
viewing size of index, 13-4
views
and indexing, 3-8
materialized

W

wildcard operator, 4-18
improving performance, 7-14
WITHIN operator, 3-24
wizard
Oracle Text addin, 2-4, A-1, B-1
word query, 4-7
case-sensitivity, 4-10
wordlist
about, 3-21

X

XML DB, 1-5
XML documents
  attribute searching, 8-13
  doctype sensitive sections, 8-14
  indexing, 8-2
  section searching, 8-13
XML searching, 1-5
XML_SECTION_GROUP object, 8-2

Z

zone section
  definition, 8-4
  nested, 8-5
  overlapping, 8-5
  repeating, 8-5