Oracle® Database

Oracle Text Application Developer's Guide

23ai
F46836-09
April 2025

Oracle Database Oracle Text Application Developer's Guide, 23ai

F46836-09

Primary Author: Binika Kumar

Contributing Authors: Doug Williams, Prakash Jashnani

Contributors: Ajay Sunnyhith Chidurala, Aleksandra Czarlinska, Asha Makur, Bonnie Xia, Ce Wei, Denis Mukhin, Edwin Balthes, Gaurav Yadav, George Krupka, Mohammad Faisal, Nilay Panchal, Paul Lane, Rahul Kadwe, Rodrigo Fuentes Hernandez, Roger Ford, Sanoop Sethumadhavan, Saurabh Naresh Netravalkar, Simona Herdan, Sudhir Kumar, Yiming Qi

Contents

3 Indexing with Oracle Text

4 Creating Oracle Text Indexes

8 Presenting Documents in Oracle Text

9 Classifying Documents in Oracle Text

10 Tuning Oracle Text

11 Searching Document Sections in Oracle Text

C Custom Index Preference Examples

Preface

Oracle Text Application Developer's Guide provides information for building applications with Oracle Text.

Audience

This document is intended for users who perform the following tasks:

- Develop Oracle Text applications
- Administer Oracle Text installations

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

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customer access to and use of Oracle support services will be pursuant to the terms and conditions specified in their Oracle order for the applicable services.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

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

Oracle Text enables you to build text query applications and document classification applications.

This chapter contains the following topics:

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

1.1 Introduction to Oracle Text

Oracle Text provides indexing, word and theme searching, and viewing capabilities for text in query applications and document classification applications.

To design an Oracle Text application, first determine the type of queries that you expect to run. When you know the types, you can choose the most suitable index for the task.

Oracle Text is used for the following categories of applications:

- Document Collection Applications
- Catalog Information Applications
- Document Classification Applications
- XML Search Applications

1.2 Document Collection Applications

A text query application enables users to search *document collections*, such as websites, digital libraries, or document warehouses.

This section contains the following topics.

- About Document Collection Applications
- Flowchart of Text Query Application

1.2.1 About Document Collection Applications

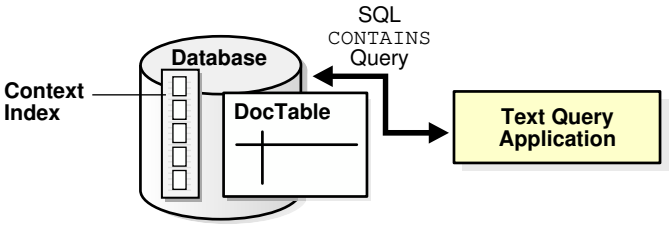
The collection is typically static and has no significant change in content after the initial indexing run. Documents can be any size and format, such as HTML, PDF, or Microsoft Word. These documents are stored in a document table. Searching is enabled by first indexing the document collection.

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

For this type of application, you should retrieve documents that are most relevant to a query. The documents must rank high in the result list.

The queries 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



1.2.2 Flowchart of Text Query Application

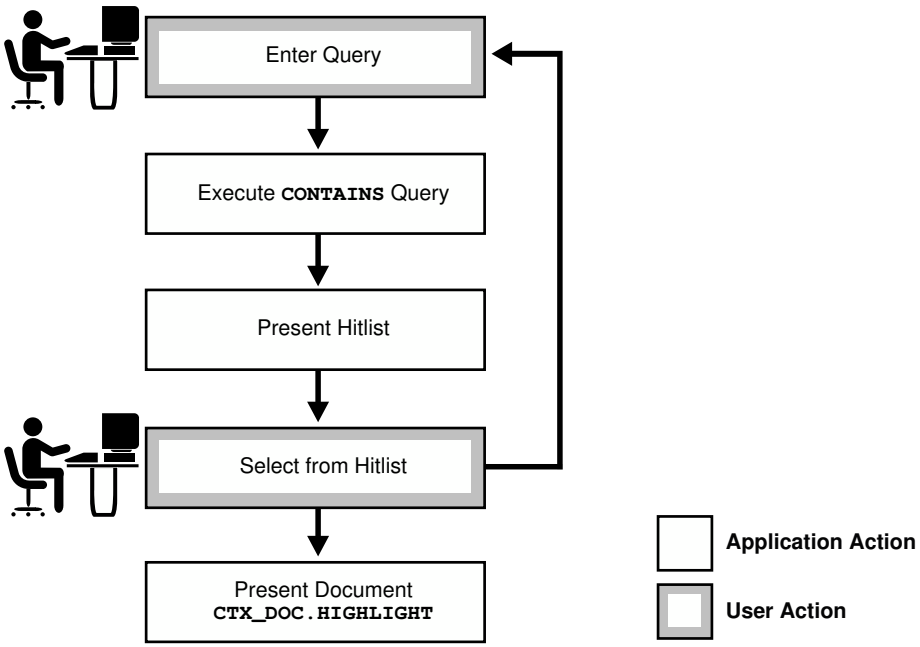
A typical text query application on a document collection lets the user enter a query. The application enters a `CONTAINS` query and returns a list, called a *hitlist*, of documents that satisfy the query. The results are usually ranked by relevance. The application enables the user to view one or more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

For example, an application might index URLs (HTML files) on the 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 user interaction with a simple text 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 the hitlist.
5. The application presents a document to the user for viewing.

Figure 1-2 Flowchart of a Text Query Application



1.3 Catalog Information Applications

Catalog information consists of inventory type information, such as for an online book store or auction site.

This section contains the following topics.

- About Catalog Information Applications
- Flowchart for Catalog Query Application

1.3.1 About Catalog Information Applications

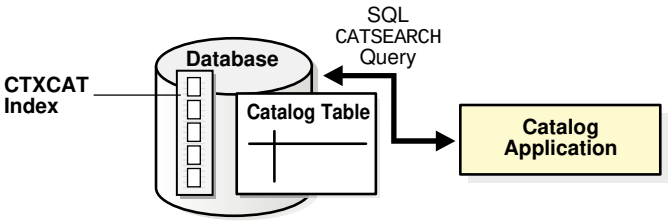
The stored catalog information consists of text information, such as book titles, and related structured information, such as price. The information is usually updated regularly to keep the online catalog up-to-date with the inventory.

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 relationship 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



> **Note:**
>
> The Oracle Text indextype `CTXCAT` is deprecated with Oracle Database 23ai. The indextype itself, and it's operator `CTXCAT`, can be removed in a future release. Both `CTXCAT` and the use of `CTXCAT` grammar as an alternative grammar for `CONTEXT` queries is deprecated. Instead, Oracle recommends that you use the `CONTEXT` indextype, which can provide all the same functionality, except that it is not transactional. Near-transactional behavior in `CONTEXT` can be achieved by using `SYNC(ON COMMIT)` or, preferably, `SYNC(EVERY [time-period])` with a short time period.
>
> `CTXCAT` was introduced when indexes were typically a few megabytes in size. Modern, large indexes, can be difficult to manage with `CTXCAT`. The addition of index sets to `CTXCAT` can be achieved more effectively by the use of `FILTER BY` and `ORDER BY` columns, or `SDATA`, or both, in the `CONTEXT` indextype. `CTXCAT` is therefore rarely an appropriate choice. Oracle recommends that you choose the more efficient `CONTEXT` indextype.

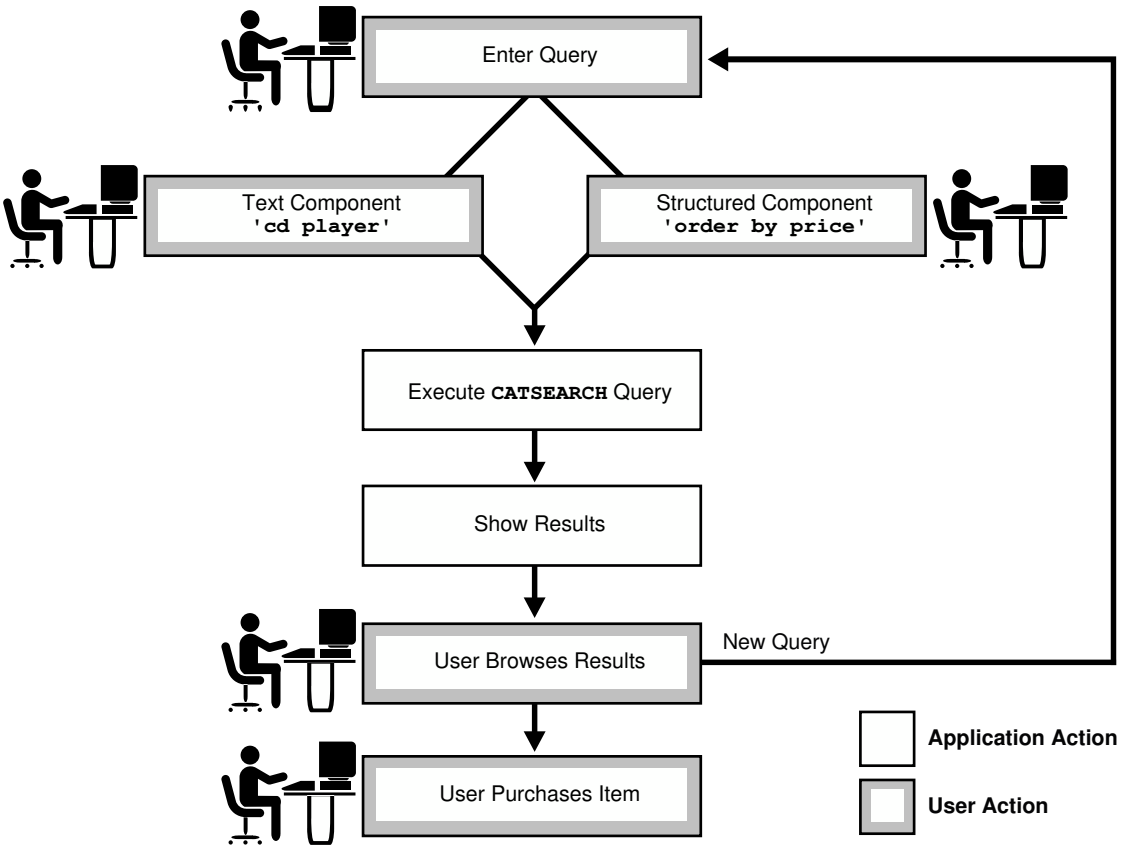
1.3.2 Flowchart for Catalog Query Application

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

Figure 1-4 Flowchart of a Catalog Query Application



1.4 Document Classification Applications

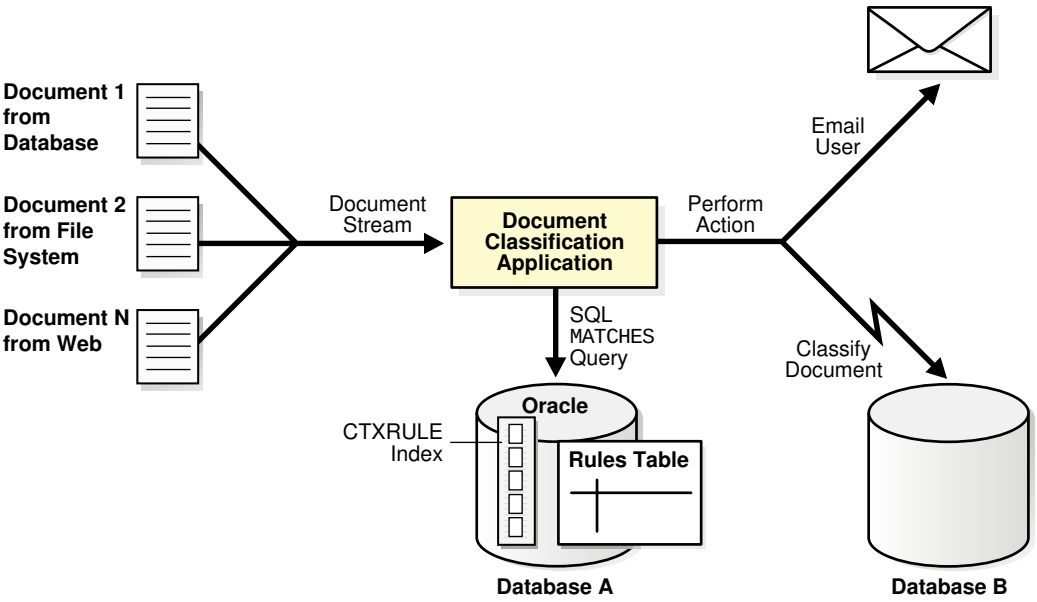
In a document classification application, an incoming stream or a set of documents is compared to a predefined set of rules. If a document matches one or more rules, then the application performs an action.

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

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

To create a document classification application, 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



1.5 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 the document part 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 by using the following approaches:

- The CONTAINS Operator with XML Search Applications
- Combining Oracle Text Features with Oracle XML DB (XML Search Index)

> **See Also:**
>
> Using XML Query Result Set Interface

1.5.1 The CONTAINS Operator with XML Search Applications

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:

- Token-based, whitespace-normalized searches
- Hitlists ranked by relevance
- Case-sensitive searching
- Section searching

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

> **WARNING:**
>
> Starting with Oracle Database 12*c*, Oracle XML Database (XML DB) is automatically installed when you install the new Oracle Database software or when you upgrade.

> **See Also:**
>
> "XML Section Searching with Oracle Text"

1.5.2 Combining Oracle Text Features with Oracle XML DB (XML Search Index)

When you want a full-text retrieval for applications, combine the features of Oracle Text and Oracle XML DB to create an XML Search Index. In this case, leverage the XML structure by entering queries such as "find all nodes that contain the word Pentium." Oracle Database 12*c* extends Oracle's support for the W3C XQuery specification by adding support for the XQuery full-text extension. This support lets you perform XML-aware, full-text searches on XML content that is stored in the database.

The following topics explain how to use Oracle XML DB with Oracle Text applications:

- Using the xml_enable Method for an XML Search Index
- Using the Text-on-XML Method
- Indexing JSON Data

> **See Also:**
>
> - "XML Section Searching with Oracle Text"
> - *Oracle Text Reference* for information about the `xml_enable` variable of `SET_SEC_GRP_ATTR` to enable XML awareness
> - *Oracle XML DB Developer's Guide* for more information about XML full-text indexing and XML Search Index

1.5.2.1 Using the xml_enable Method for an XML Search Index

An XML Search Index is an XML-enabled Oracle Text index (CTXSYS.CONTEXT). This index type supports information-retrieval searching and structured searching in one unified index. XML Search Index also stores a Binary Persistent Document Object Model (PDOM) internally within an Oracle Text table, so that XML operations can be functionally evaluated over the Binary PDOM. This XML Search Index is supported for XMLTYPE datastores. XMLEXISTS is seamlessly rewritten to a `CONTAINS` query in the presence of such an XML Search Index.

When you create an XML Search Index, a Binary PDOM of the XML document is materialized in an internal table of Oracle Text. Post evaluation from the Oracle Text index is redirected to go against the PDOM stored in this internal table.

> **See Also:**
>
> *Oracle Text Reference* for information on `xml_enable` variable of `SET_SEC_GRP_ATTR` to enable XML awareness for XML Search Index

The following example creates an Oracle XML Search Index:

```
exec
CTX_DDL.CREATE_SECTION_GROUP('secgroup','PATH_SECTION_GROUP');
exec
CTX_DDL.SET_SEC_GRP_ATTR('secgroup','xml_enable','t');
CREATE INDEX po_ctx_idx on T(X) indextype is ctxsys.context
parameters ('section group SECGROUP');
```

1.5.2.2 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` or any supported type, provided that 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. That is, specify 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;
```

Use the `INPATH` operator to specify more complex criteria with `XPATH` expressions:

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

1.5.2.3 Indexing JSON Data

JavaScript Object Notation (JSON) is a language-independent data format that is used for serializing structured data and exchanging this data over a network, typically between a server and web applications. JSON provides a text-based way of representing JavaScript object literals, arrays, and scalar data.

See Also:

- *Oracle Text Reference* for information about creating a search index on JSON
- *Oracle Database JSON Developer's Guide* for more information about JSON

2
Getting Started with Oracle Text

You can create an Oracle Text developer user account and build simple text query and catalog applications.

This chapter contains the following topics:

2.1 Overview of Getting Started with Oracle Text

This chapter provides basic information about how to configure Oracle Text, 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 appendixes.

> **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.

> **See Also:**
>
> " Classifying Documents in Oracle Text" to learn more about building document classification applications

2.2 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, perform the following steps as the system administrator user:

1. Create the user.

   The following SQL statement creates a user called `MYUSER` with a password of *`password`*:

   ```
   CREATE USER myuser IDENTIFIED BY password;
   ```

2. Grant roles to the user.

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

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

3. Grant EXECUTE privileges on the `CTX PL/SQL` package.

   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 these packages.

   To call any of these procedures from a stored procedure, your application requires execute privileges on the packages. For example, to grant `execute` privileges to `MYUSER` 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;
   ```

> **Note:**
>
> These permissions are granted to the `CTXAPP` role. However, because role permissions do not always work in PL/SQL procedures, it is safest to explicitly grant these permissions to the user who already has the `CTXAPP` role.

2.3 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`.

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 CONTEXT Query Application.

The examples in this section provide the basic SQL statements to load the text table, index the documents, and query the index.

- Creating the Text Table
- Using SQL*Loader to Load the Table

2.3.1 Creating the Text Table

Perform the following steps to create and load documents into a table.

1. Connect as the new user.

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

   ```
   CONNECT myuser;
   ```

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));
   ```

   > **Note:**
   >
   > Primary keys of the following type are supported: `NUMBER`, `VARCHAR2`, `DATE`, `CHAR`, `VARCHAR`, and `RAW`.

3. Load documents into the table.

   Use the SQL `INSERT` statement to load text into a table.

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

   ```
   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>');
   ```

2.3.2 Using SQL*Loader to Load the Table

You can use SQL*Loader to load a table in batches.

Perform the following steps to load your table in batches with SQL*Loader:

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. If you index PDF, Microsoft Word, or other formatted documents, then use the `CTXSYS.AUTO_FILTER` (the default) as your `FILTER` preference.

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

   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 eliminate unwanted markup, such as font information, from the index.

2. Query your table with `CONTAINS`.

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

   ```
   COLUMN text FORMAT a40;
   ```

Next, query the table with the `SELECT` statement with `CONTAINS`. This query 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>
```

3. Present the document.

   In a real-world application, you could present the selected document with query terms highlighted. Oracle Text enables you to mark up documents with the `CTX_DOC` package.

   You can demonstrate HTML document markup with an anonymous PL/SQL block in SQL*Plus. However, in a real-world application, you could 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 data type) to store the markup text and reads it back to the standard output. The `CLOB` is then deallocated 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.
```

4. Synchronize the index after data manipulation.

   When you create a `CONTEXT` index, you explicitly synchronize your index to update it with any inserts, updates, or deletions 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 2 Mb 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>
```

> **See Also:**
>
> "Building the PSP Web Application" for an example of how to use SQL*Loader to load a text table from a data file

2.4 Catalog Application Quick Tour

The examples in this section provide the basic SQL statements to create a catalog index for an auction site that sells electronic equipment, such as cameras and CD players.

New inventory is added every day, 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 use the `CATSEARCH` operator.

- Creating the Table
- Using SQL*Loader to Load the Table

  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 CATSEARCH Query Application .

> **Note:**
>
> The Oracle Text indextype `CTXCAT` is deprecated with Oracle Database 23ai. The indextype itself, and it's operator `CTXCAT`, can be removed in a future release. Both `CTXCAT` and the use of `CTXCAT` grammar as an alternative grammar for `CONTEXT` queries is deprecated. Instead, Oracle recommends that you use the `CONTEXT` indextype, which can provide all the same functionality, except that it is not transactional. Near-transactional behavior in `CONTEXT` can be achieved by using `SYNC(ON COMMIT)` or, preferably, `SYNC(EVERY [time-period])` with a short time period.
>
> `CTXCAT` was introduced when indexes were typically a few megabytes in size. Modern, large indexes, can be difficult to manage with `CTXCAT`. The addition of index sets to `CTXCAT` can be achieved more effectively by the use of `FILTER BY` and `ORDER BY` columns, or `SDATA`, or both, in the `CONTEXT` indextype. `CTXCAT` is therefore rarely an appropriate choice. Oracle recommends that you choose the more efficient `CONTEXT` indextype.

2.4.1 Creating the Table

Perform the following steps to create and load the table:

1. Connect as the appropriate user.

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

   ```
   CONNECT myuser;
   ```

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);
   ```

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');
   ```

2.4.2 Using SQL*Loader to Load the Table

You can use SQL*Loader to load a table in batches.

Perform the following steps to load your table in batches with SQL*Loader:

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. Later

on, when you use the CATSEARCH operator, specify the terms for the text column and the criteria for the structured clause.

2. Create the subindex to order by price.

   For Oracle Text to serve these queries efficiently, you need a subindex for the price column, because your queries are ordered by price.

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

```
EXEC CTX_DDL.CREATE_INDEX_SET('auction_iset');
EXEC CTX_DDL.ADD_INDEX('auction_iset','price'); /* subindex A*/
```

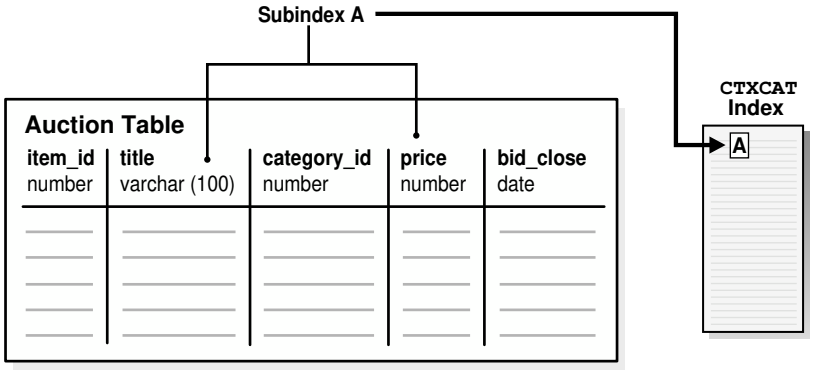
3. Create the CTXCAT index.

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

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

   The following figure shows how the CTXCAT index and its subindex relate to the columns.

Figure 2-1 Auction table schema and CTXCAT index



4. Query your table with CATSEARCH.

   After you create the CTXCAT index on the AUCTION table, query this index with the CATSEARCH operator.

   First, set the output format to make the output readable:

```
COLUMN title FORMAT a40;
```

   Next, 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
```

5. 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 the 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.

> **See Also:**
>
> "Building the PSP Web Application" for an example of how to use SQL*Loader to load a text table from a data file

2.5 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.

This section contains the following sections:

- About Classification of a Document
- Steps for Creating a Classification Application

2.5.1 About Classification of a Document

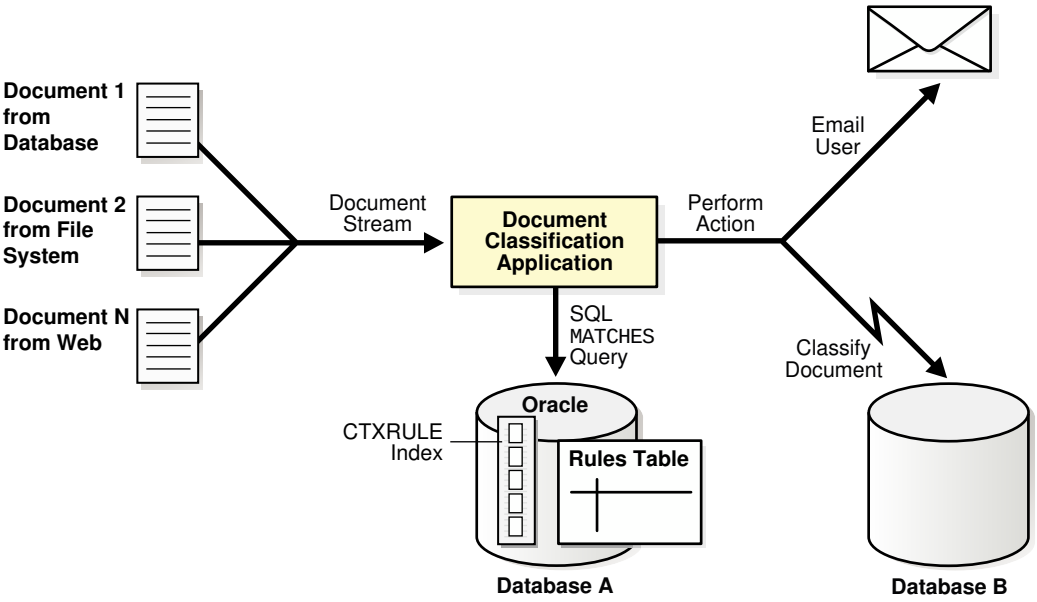
Documents are classified according to predefined rules. These rules select documents for a category. For instance, a query rule of *'presidential elections'* selects documents for a category about politics.

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

set of training documents that you provide. With *clustering,* Oracle Text does all the work for you, deriving both rules and categories.

To create a simple classification application for document content using Oracle Text, you create **rules.** Rules are essentially a table of queries that categorize document content. You index these rules in a `CTXRULE` index. To classify an incoming stream of text, use the `MATCHES` operator in the `WHERE` clause of a `SELECT` statement. See the following image for the general flow of a classification application.

Figure 2-2 Overview of a Document Classification Application



> **See Also:**
>
> "Overview of Document Classification"

2.5.2 Creating a Classification Application

The following example shows how to classify documents by using `myuser` with the `CTXAPP` role. You define simple categories, create a `CTXRULE` index, and use `MATCHES`.

1. Connect as the appropriate user.

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

   ```
   CONNECT myuser;
   ```

2. Create the rule table.

   In this example, you create a table called `queries`. Each row defines a category with an ID and a rule that 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');
```

3. Create your `CTXRULE` index.

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

4. Classify with `MATCHES`.

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

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

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

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

> **See Also:**
>
> Classifying Documents in Oracle Text for more extended classification examples

3
Indexing with Oracle Text

Oracle Text provides several types of indexes, which you create depending on the type of application that you develop.

This chapter contains the following topics:

3.1 About Oracle Text Indexes

The discussion of Oracle Text indexes includes the different types of indexes, their structure, the indexing process, and limitations.

The following topics provide information about Oracle Text indexes:

3.1.1 Types of Oracle Text Indexes

With Oracle Text, you create indexes by using the `CREATE INDEX` statement.

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 in, for example, 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 insert, update, and delete operations to the base table.	All `CREATE INDEX` preferences and parameters are supported, except for `INDEX SET`. Supported parameters: index partition clause format, charset, and language columns	`CONTAINS` The `CONTEXT` grammar supports a rich set of operations. Use the `CTXCAT` grammar 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 index structured column values for more efficient processing of mixed queries.
`SEARCH INDEX`	Use this index to build a text retrieval application when your text consists of large, coherent documents in, for example, 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 insert, update, and delete operations to the base table.	All `CREATE INDEX` preferences and parameters are supported, except for `INDEX SET`. Supported parameters: index partition clause format, charset, and language columns	`CONTAINS` The `SEARCH INDEX` grammar supports a rich set of operations. Use the `CONTEXT` and `CTXCAT` grammar with query templating.	Supports all documents services and query services. Supports indexing of partitioned text tables. Supports sharded databases and system managed partitioning for index storage tables.

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

Index Type	Description	Supported Preferences and Parameters	Query Operator	Notes
CTXCAT	Use this index for better mixed query performance of small documents and text fragments. To improve mixed query performance, include other columns in the base table, such as item names, prices, and descriptions. This index type is transactional. It automatically updates itself after inserts, updates, or deletes to the base table. CTX_DDL.SYNC_INDEX is not necessary. **Note**: The Oracle Text indextype CTXCAT is deprecated with Oracle Database 23ai. The indextype itself, and it's operator CTXCAT, can be removed in a future release. CTXCAT was introduced when indexes were typically a few megabytes in size. Modern, large indexes, can be difficult to manage with CTXCAT. The addition of index sets to CTXCAT can be achieved more effectively by the use of FILTER BY and ORDER BY columns, or SDATA, or both, in the CONTEXT indextype. CTXCAT is therefore rarely an appropriate choice. Oracle recommends that you choose the more efficient CONTEXT indextype.	INDEX SET LEXER STOPLIST STORAGE WORDLIST (The prefix_index attribute is supported only for Japanese data.) Not supported: Format, charset, and language columns Table and index partitioning	CATSEARCH The CTXCAT grammar supports logical operations, phrase queries, and wildcarding. Use the CONTEXT grammar 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 index partitioning, documents services (highlighting, markup, themes, and gists) or query services (explain, query feedback, and browse words.)

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

Index Type	Description	Supported Preferences and Parameters	Query Operator	Notes
CTXRULE	Use this index to build a document classification or routing application. Create this index on a table of queries, where the queries define the classification or routing criteria..	See "CTXRULE Parameters and Limitations".	MATCHES	Use the MATCHES operator to classify single documents (plain text, HTML, or XML). MATCHES turns a document into a set of queries and finds the matching rows in the index. To build a document classification application by using *simple* or *rule-based* classification, create an index of type CTXRULE. This index classifies plain text, HTML, or XML documents by using the MATCHES operator. Store your defining query set in the text table that you index.

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 the location of the stored text. Text is usually a collection of documents, but it can also be small text fragments.

> **Note:**
>
> If you are building a new application that uses XML data, Oracle recommends that you use XMLIndex, not CTXRULE.

Create an Oracle Text index as a type of extensible index to Oracle Database by 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 benefit of creating an Oracle Text index is fast response time for text queries with the CONTAINS, CATSEARCH, and MATCHES 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, do not use it with Oracle Text. However, you can create an Oracle Text index on a column in a table that is stored in a Transparent Data Encryption-enabled tablespace.

> **See Also:**
>
> - "Creating Oracle Text Indexes"
> - *Oracle XML DB Developer's Guide* for information about `XMLIndex` and indexing `XMLType` data

3.1.2 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 the 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 Indexing

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 also enable and disable theme indexing.

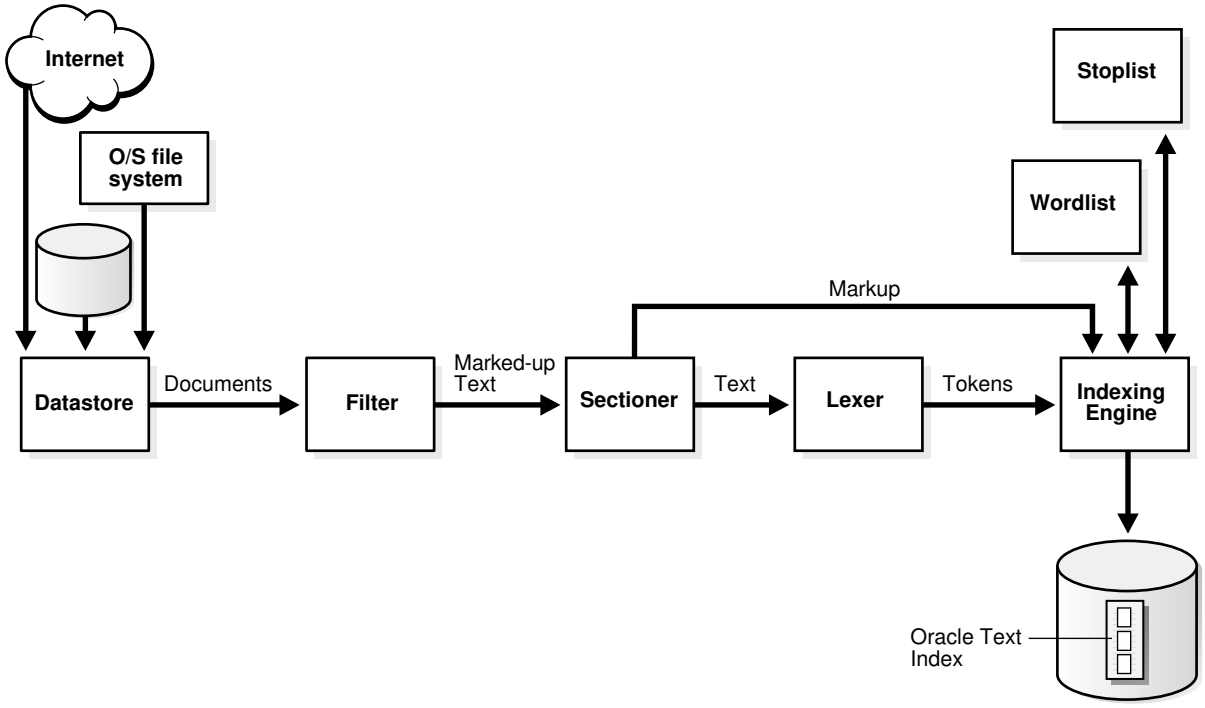
> **See Also:**
>
> "Creating Preferences " to learn more about indexing theme information

3.1.3 Oracle Text Indexing Process

This section describes the Oracle Text indexing process. Initiate the indexing process by using the `CREATE INDEX` statement to create an Oracle Text index of tokens, organized according to your parameters and preferences.

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 that you can specify in the parameter string of `CREATE INDEX` or `ALTER INDEX`.

Figure 3-1 Oracle Text Indexing Process



Oracle Text processes the data stream with the following objects and engine:

- Datastore Object
- Filter Object
- Sectioner Object
- Lexer Object
- Indexing Engine

3.1.3.1 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 defined your datastore as `DIRECTORY_DATASTORE`, then the stream starts by reading the files from an Oracle directory object. You can also store your documents on the internet or in Oracle Database. Wherever your files reside physically, a text table in Oracle Database must always point to the files.

3.1.3.2 Filter Object

The stream then passes through the filter. Your `FILTER` preference determines what happens. The stream can be acted upon in one of the following ways:

- No filtering takes place 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 when you specify the `AUTO_FILTER` preference type or when the value of the format column is `BINARY`.

3.1.3.3 Sectioner Object

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

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

3.1.3.4 Lexer Object

You create a lexer preference by 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 that are defined in your lexer preference. These parameters include the definitions for the characters that separate tokens, such as whitespace. Parameters also include whether to convert the text to all uppercase or to leave it in mixed case.

When you enable theme indexing, the lexer analyzes your text to create theme tokens for indexing.

3.1.3.5 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 that you specify to exclude stopwords or stopthemes from the index. Oracle Text also uses the parameters that are defined in your `WORDLIST` preference. Those parameters tell the system how to create a prefix index or substring index, if enabled.

3.1.4 About Updates to Indexed Columns

You can keep documents available for search operations until the index is synchronized, without immediately performing index synchronization.

In releases prior to Oracle Database 12*c* Release 2 (12.2), when there is an update to the column on which an Oracle Text index is based, the document is unavailable for search operations until the index is synchronized. User queries cannot perform a search of this document. Starting with Oracle Database 12*c* Release 2 (12.2), you can specify that documents must be searchable after updates, without immediately performing index synchronization. Before the index is synchronized, queries use the old index entries to fetch the contents of the old document. After index synchronization, user queries fetch the contents of the updated document.

The `ASYNCHRONOUS_UPDATE` option for indexes enables you to retain the old contents of a document after an update and then use this index to answer user queries.

> **Note:**
>
> The `ASYNCHRONOUS_UPDATE` setting of the `CONTEXT` indextype is deprecated in Oracle Database 23ai, and can be ignored or removed in a future release.
>
> Oracle can ignore or remove this attribute in a future release. Oracle recommends that you allow this value to be set to its default value, `SYNCHRONOUS_UPDATE`. To avoid unexpected loss of results during updates, use `SYNC (ON COMMIT)` or `SYNC(EVERY [time-period])` with a short time period.
>
> The `ASYNCHRONOUS_UPDATE` setting was introduced as a workaround for the fact that updates are implemented as "delete followed by insert," and that deletes are immediate (on commit), while inserts are only performed during an index sync. However, this setting is incompatible with several other index options. Oracle recommends that you discontinue its use.

Related Topics

- CREATE INDEX
- ALTER_INDEX

3.1.5 Partitioned Tables and Indexes

When you create a partitioned `CONTEXT` index on a partitioned text table, you must partition the table by range. Hash, composite, and list partitions are not supported.

You can 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, insert, update, delete operations, 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 that are 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

To query a partitioned table, 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.

3.1.6 Online Indexes

When it is not practical to lock your base table for indexing because of ongoing updates, you can create your index online with the `ONLINE` parameter of `CREATE INDEX` statement. This way an application with frequent inserts, updates, or deletes does not have to 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

3.1.7 Parallel Indexing

Oracle Text supports parallel indexing with the `CREATE INDEX` statement.

When you enter a parallel indexing statement on a nonpartitioned 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 intensive I/O operation, parallel indexing is most effective in decreasing your indexing time when you have distributed disk access and multiple CPUs. Parallel indexing can affect the performance of an initial index only with the `CREATE INDEX` statement. It does not affect insert, update, and delete operations with `ALTER INDEX`, and has minimal effect 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"
> – "Frequently Asked Questions About Indexing Performance"

3.1.8 Indexing and Views

If you want to index documents that have contents in different tables, then create a ubiquitous search index.

You can use the `DBMS_SEARCH PL/SQL` package to create a ubiquitous search index on multiple tables and views within a schema. You can create this index only on the views that have a primary key and a foreign key constraint relationship with the component table.

Alternatively, you can create a data storage preference by using the `USER_DATASTORE` object. With this object, you can define a procedure that synthesizes documents from different tables at index time.

Oracle Text supports the creation of `CONTEXT`, `CTXCAT`, and `CTXRULE` indexes on materialized views (`MVIEW`).

Related Topics

- *Oracle Text Reference*
- Performing Ubiquitous Search with DBMS_SEARCH APIs
  Starting with Oracle Database 23ai, you can use the `DBMS_SEARCH PL/SQL` package for indexing of multiple schema objects in a single index, enabling you to search across the entire database.

3.2 Considerations for Oracle Text Indexing

Use the `CREATE INDEX` statement to create an Oracle Text index. When you create an index but do not specify a parameter string, an index is created with default parameters. You can create a `CONTEXT` index, a `CTXCAT` index, or a `CTXRULE` index.

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

This section contains the following topics:

- Location of Text
- Supported Column Types
- Storing Text in the Text Table
- Storing File Path Names
- Storing URLs
- Storing Associated Document Information
- Format and Character Set Columns
- Supported Document Formats
- Summary of DATASTORE Types
- Document Formats and Filtering
- Bypass Rows
- Document Character Set

3.2.1 Location of Text

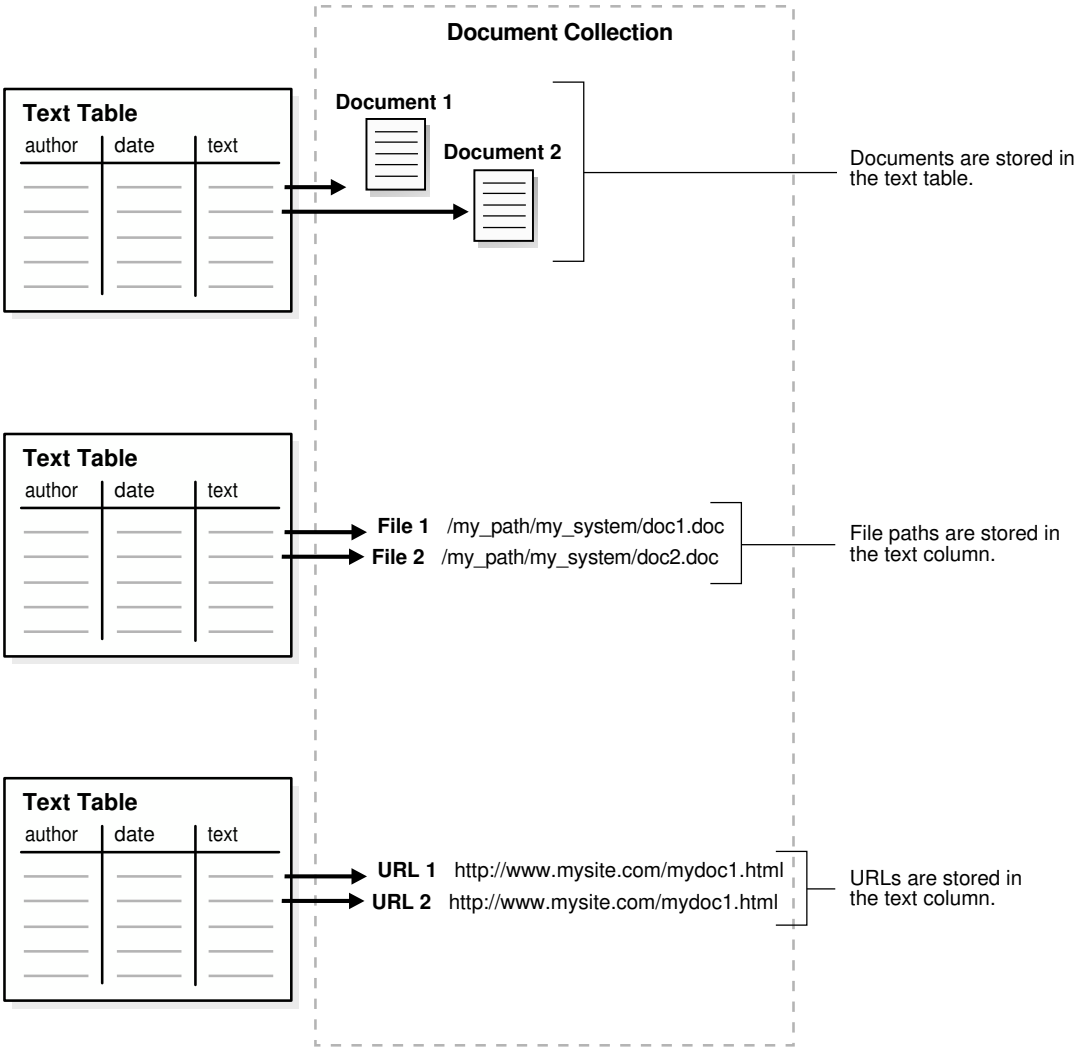
The basic prerequisite for a text query application is a text table that is populated with your document collection. The text table is required for indexing.

When you create a `CONTEXT` index, populate rows in your text table with one of the following elements. `CTXCAT` and `CTXRULE` indexes support only the first method.

- Text information (Documents or text fragments. By default, the indexing operation expects your document text to be directly loaded in your text table.)
- Path names of documents in your file system
- URLs of web documents

Figure 3-2 illustrates these different methods.

Figure 3-2 Different Ways of Storing Text



3.2.2 Supported Column Types

With Oracle Text, you can create a CONTEXT index with columns of type VARCHAR2, CLOB (limited to 4294967295 bytes), BLOB, CHAR, BFILE, XMLType, and URIType.

> **Note:**
>
> You cannot index the NCLOB, DATE, and NUMBER column types.

3.2.3 Storing Text in the Text Table

For CONTEXT data storage, use these datastore types to store documents in your text table.

- DIRECT_DATASTORE: In one column
- MULTI_COLUMN_DATASTORE: In multiple columns (Oracle Text concatenates the columns into a virtual document, one document for each row.)
- DETAIL_DATASTORE: Primary-detail relationships (Store one document across a number of rows.)
- NESTED_DATASTORE: In a nested table

Oracle Text supports the indexing of the XMLType data type, which you use to store XML documents.

For CTXCAT data storage, you can store short text fragments, such as names, descriptions, and addresses, over a number of columns. A CTXCAT index improves performance for mixed queries.

3.2.4 Storing File Path Names

In your text table, store path names to files stored in your file system. During indexing, use the DIRECTORY_DATASTORE preference type. This method of data storage is supported only for CONTEXT indexes.

> **Note:**
>
> Starting with Oracle Database 19c, the Oracle Text type FILE_DATASTORE is deprecated. Use DIRECTORY_DATASTORE instead.
>
> Oracle recommends that you replace FILE_DATASTORE text indexes with the DIRECTORY_DATASTORE index type, which is available starting with Oracle Database 19c. DIRECTORY_DATASTORE provides greater security because it enables file access to be based on directory objects.

3.2.5 Storing URLs

Store URL names to index websites. During indexing, use the NETWORK_DATASTORE preference type. This method of data storage is supported only for CONTEXT indexes.

> **Note:**
>
> Starting with Oracle Database 19c, the Oracle Text type `URL_DATASTORE` is deprecated. Use `NETWORK_DATASTORE` instead.
>
> The `URL_DATASTORE` type is used for text stored in files on the internet (accessed through HTTP or FTP), and for text stored in local file system files (accessed through the file protocol). It is replaced with `NETWORK_DATASTORE`, which uses ACLs to allow access to specific servers. This change aligns Oracle Text more closely with the standard operating and security model for accessing URLs from the database.

3.2.6 Storing Associated Document Information

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

3.2.7 Format and Character Set Columns

If your documents consist of mixed formats or mixed character sets, 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`. `IGNORE` is useful for bypassing rows containing data that is incompatible with Oracle Text indexing, such as images.
- A character set column to record the document character set for each row.

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

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

3.2.8 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 your text column has mixed formats, you can include a format column to help filtering during indexing, and you can specify whether a document is binary (formatted) or text (nonformatted, such as HTML). If you mix HTML and XML documents in one index, you might not be able to configure your index to your needs; you cannot prevent style sheet information from being added to the index.

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

3.2.9 Summary of DATASTORE Types

When you use `CREATE INDEX`, specify the location that uses the datastore preference. Use an appropriate datastore according to your application.

These are the different ways that you can store your text with datastore preference types.

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 primary table.
`FILE_DATASTORE`	Data is stored externally in operating system files. File names are stored in the text column, one for each row. **Note:** Starting with Oracle Database 19c, the Oracle Text type `FILE_DATASTORE` is deprecated. Use `DIRECTORY_DATASTORE` instead.
`DIRECTORY_DATASTORE`	Data is stored externally in Oracle directory objects. File names 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. **Note:** Starting with Oracle Database 19c, the Oracle Text type `URL_DATASTORE` is deprecated. Use `NETWORK_DATASTORE` instead.
`NETWORK_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 increases for indexing URLs, because the system must retrieve the document from the network.

> **Note:**
>
> To troubleshoot issues with triggers and `MULTI_COLUMN_DATASTORE` or `USER_DATASTORE`, refer to My Oracle Support document 1613741.1.

Related Topics

- MOS Document 1613741.1
- Datastore Examples
  You can use datastore preferences to specify how your text is stored. These are the examples for setting some of the datastore preference types.
- *Oracle Text Reference*

3.2.10 Document Formats and Filtering

To index formatted documents, such as Microsoft Word and PDF, you must filter them to text. The `FILTER` preference type determines the type of filtering that the system uses. 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. It can also index columns that contain mixed-format documents.

- No Filtering for HTML
- Filtering Mixed-Format Columns
- Custom Filtering

> **See Also:**
>
> *Oracle Text Reference* for information about `AUTO_FILTER` supported document and graphics formats

3.2.10.1 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"

3.2.10.2 Mixed-Format Columns Filtering

For 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, tag each row TEXT or BINARY. Rows that are tagged TEXT are not filtered.

For example, 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.

When you do not want a document to be indexed, you can use a third format column type, IGNORE. This column type is useful, for example, when a mixed-format table includes plain-text documents in Japanese and English, but you only want to process the English documents. This column type is also useful when a mixed-format table includes plain-text documents and images. Because IGNORE is implemented at the datastore level, you can use it with all filters.

3.2.10.3 Custom Filtering

You can create a 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"

3.2.11 Bypass Rows

In your text table, you can bypass rows that you do not want to index, such as rows that contain image data. To bypass rows, you create a format column, set it to IGNORE, and name the format column in the parameter clause of the CREATE INDEX statement.

3.2.12 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 filter type to convert your text for indexing.

Character Set Detection

When you set the CHARSET column 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.

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 that you create a CHARSET column, populate this column with the name of the document character set for each row, and name the column in the parameter clause of the CREATE INDEX statement.

3.3 Document Language

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

Depending on the language of your documents, use one of the following lexer types:

- `AUTO_LEXER`: To automatically detect the language being indexed by examining the content, and apply suitable options (including stemming) for that language. Works best where each document contains a single-language, and has at least a couple of paragraphs of text to aid identification.
- `BASIC_LEXER`: 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.
- `MULTI_LEXER`: To index tables containing documents of different languages such as English, German, and Japanese.
- `CHINESE_VGRAM`: To extract tokens from Chinese text.
- `CHINESE_LEXER`: To extract tokens from Chinese text. This lexer offers the following benefits over the `CHINESE_VGRAM` lexer:
  - Generates a smaller index
  - Better query response time
  - Generates real world tokens resulting in better query precision
  - Supports stop words
- `JAPANESE_VGRAM`: To extract tokens from Japanese text.
- `JAPANESE_LEXER`: To extract tokens from Japanese text. This lexer offers the following advantages over the `JAPANESE_VGRAM` lexer:
  - Generates smaller index
  - Better query response time
  - Generates real world tokens resulting in better precision
- `KOREAN_MORPH_LEXER`: To extract tokens from Korean text.
- `USER_LEXER`: To create your own lexer for indexing a particular language.
- `WORLD_LEXER`: To index tables containing documents of different languages and to autodetect the languages in the document.

With the `BASIC_LEXER` preference, Oracle Text provides a lexing solution for most languages. For the Japanese, Chinese, and Korean languages, you can create your own lexing solution in the user-defined lexer interface.

- **Language Features Outside BASIC_LEXER**: The user-defined lexer interface enables you to create a PL/SQL or Java procedure to process your documents during indexing and querying. With the user-defined lexer, you can also create your own theme lexing solution or linguistic processing engine.
- **Multilanguage Columns**: Oracle Text can index text columns that contain documents in different languages, such as a column that contains documents written in English, German, and Japanese. To index a multilanguage column, you add a language column to your text

table and use the `MULTI_LEXER` preference type. You can also incorporate a multilanguage stoplist when you index multilanguage columns.

Related Topics

- *Oracle Text Reference*
- MULTI_LEXER Example: Indexing a Multilanguage Table

3.4 Special Characters

When you use the `BASIC_LEXER` preference type, you can specify how nonalphanumeric 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 the hyphen (-) when it indexes a word such as *vice-president.*

These characters fall into `BASIC_LEXER` categories according to the behavior that 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:** Define a nonalphanumeric 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 underscores, define them as printjoins. This means that a word such as *vice-president* is indexed as *vice-president.* A query on *vicepresident* does not find *vice-president.*
- **Skipjoin Characters:** Define a nonalphanumeric character as `skipjoin` when you do not want this character to be indexed with the token that contains it. For example, with the hyphen (-) defined as a skipjoin, *vice-president* is indexed as *vicepresident.* A query on *vice-president* finds documents containing *vice-president* and *vicepresident*.
- **Other Characters:** You can specify other characters 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 modifiable defaults.

> **See Also:**
>
> - "BASIC_LEXER Example: Setting Printjoin Characters"
> - *Oracle Text Reference* to learn more about the `BASIC_LEXER` type

3.5 Case-Sensitive Indexing and Querying

By default, all text tokens are converted to uppercase and then indexed. This conversion results in case-insensitive queries. For example, queries on *cat, CAT,* and *Cat* 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 the 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`

3.6 Improved Document Services Performance with a Forward Index

When it searches for a word in a document, Oracle Text uses an inverted index and then displays the results by calculating the snippet from that document. For calculating the snippet, each document returned as part of the search result is reindexed. The search operation slows down considerably when a document's size is very large.

The forward index overcomes the performance problem of very large documents. It uses a `$O` mapping table that refers to the token offsets in the `$I` inverted index table. Each token offset is translated into the character offset in the original document, and the text surrounding the character offset is then used to generate the text snippet.

Because the forward index does not use in-memory indexing of the documents while calculating the snippet, it provides considerable improved performance over the inverted index while searching for a word in very large documents.

The forward index improves the performance of the following procedures in the Oracle Text `CTX_DOC` package:

- `CTX_DOC.SNIPPET`
- `CTX_DOC.HIGHLIGHT`
- `CTX_DOC.MARKUP`

> **See Also:**
>
> *Oracle Text Reference* for information about the `forward_index` parameter clause of the `BASIC_STORAGE` indexing type

3.6.1 Enabling Forward Index

The following example enables the forward index feature by setting the `forward_index` attribute value of the `BASIC_STORAGE` storage type to `TRUE`:

```
exec ctx_ddl.create_preference('mystore', 'BASIC_STORAGE');
exec ctx_ddl.set_attribute('mystore','forward_index','TRUE');
```

3.6.2 Forward Index with Snippets

In some cases, when you use the `forward_index` option, generated snippets may be slightly different from the snippets that are generated when you do not use the `forward_index` option.

The differences are generally minimal, do not affect snippet quality, and are typically "few extra white spaces" and "newline."

3.6.3 Forward Index with Save Copy

Using Forward Index with Save Copy

To use the forward index effectively, you should store copies of the documents in the `$D` table, either in plain-text format or filtered format, depending upon the `CTX_DOC` package procedure that you use. For example, store the document in plain-text when you use the `SNIPPET` procedure and store it in the filtered format when you use the `MARKUP` or `HIGHLIGHT` procedure.

You should use the Save Copy feature of Oracle Text to store the copies of the documents in the `$D` table. Implement the feature by using the `save_copy` attribute or the `save_copy` column parameter.

- `save_copy` basic storage attribute:

  The following example sets the `save_copy` attribute value of the `BASIC_STORAGE` storage type to `PLAINTEXT`. This example enables Oracle Text to save a copy of the text document in the `$D` table while it searches for a word in that document.

  ```
  exec ctx_ddl.create_preference('mystore', 'BASIC_STORAGE');
  exec ctx_ddl.set_attribute('mystore','save_copy','PLAINTEXT');
  ```

- `save_copy column` index parameter:

  The following example uses the `save_copy column` index parameter to save a copy of a text document into the `$D` table. The `create index` statement creates the `$D` table and copies document 1 ("hello world") into the `$D` table.

  ```
  create table docs(
    id       number,
    txt      varchar2(64),
    save     varchar2(10)
  );

  insert into docs values(1, 'hello world', 'PLAINTEXT');

  create index idx on docs(txt) indextype is ctxsys.context
      parameters('save_copy column save');
  ```

For the `save_copy` attribute or column parameter, you can specify one of the following values:

- `PLAINTEXT` saves the copy of the document in a plain-text format in the `$D` index table. The plain-text format is defined as the output format of the sectioner. Specify this value when you use the `SNIPPET` procedure.
- `FILTERED` saves a copy of a document in a filtered format in the `$D` index table. The filtered format is defined as the output format of the filter. Specify this value when you use the `MARKUP` or `HIGHLIGHT` procedure.
- `NONE` does not save the copy of the document in the `$D` index table. Specify this value when you do not use the `SNIPPET`, `MARKUP`, or `HIGHLIGHT` procedure and when the indexed column is either `VARCHAR2` or `CLOB`.

3.6.4 Forward Index Without Save Copy

In the following scenarios, you can take advantage of the performance enhancement of forward index without saving copies of all documents in the `$D` table (that is, without using the Save Copy feature):

- The document set contains HTML and plain text: Store all documents in the base table by using the `DIRECT_DATASTORE` or the `MULTI_COLUMN_DATASTORE` datastore type.
- The document set contains HTML, plain text, and binary: Store all documents in the base table by using the `DIRECT_DATASTORE` datastore type. Store only the binary documents in the `$D` table in the filtered format.

3.6.5 Save Copy Without Forward Index

Even if you do not enable the forward index feature, the Save Copy feature improves the performance of the following procedures of the `CTX_DOC` package:

- `CTX_DOC.FILTER`
- `CTX_DOC.GIST`
- `CTX_DOC.THEMES`
- `CTX_DOC.TOKENS`

3.7 Language-Specific Features

You can enable the following language-specific features:

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

3.7.1 Theme Indexing

By default, themes are indexed in English and French, for which you can index document *theme* information. A document theme is a concept that is sufficiently developed in the document.

Search document themes with the `ABOUT` operator and retrieve document themes programatically with the `CTX_DOC` PL/SQL package.

Enable and disable theme indexing with the `index_themes` attribute of the `BASIC_LEXER` preference type.

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

> **See Also:**
>
> - *Oracle Text Reference* to learn more about the `BASIC_LEXER`
> - "ABOUT Queries and Themes"

3.7.2 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 do not have to contain diacritical marks to score matches.

For example, in a Spanish base-letter index, a query of *energía* matches *energía* and *energia.* However, if you disable base-letter indexing, a query of *energía* only matches *energía.*

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`

3.7.3 Alternate Spelling

Languages such as German, Danish, and Swedish contain words that have more than one accepted spelling. For example, in German, you can substitute *ae* 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 you can query these words with either spelling.

Enable and disable alternate spelling for your language with the `alternate_spelling` attribute in the `BASIC_LEXER` preference type.

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

3.7.4 Composite Words

You can create composite indexes for all the languages that are supported for `AUTO_LEXER` and `BASIC_LEXER`.

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

You can enable and disable composite indexes with the `composite` attribute of the `AUTO_LEXER` and `BASIC_LEXER` preferences. The default value for `composite` is `YES` (composite word indexing enabled).

When composite word indexing is disabled, words that are usually one entry in a dictionary are not split into composite stems. Words that are not dictionary entries are split into composite stems.

Related Topics

- AUTO_LEXER Language Support
- AUTO_LEXER Language-Independent Attributes
- BASIC_LEXER Language Support
- BASIC_LEXER Attributes

3.7.5 Korean, Japanese, and Chinese Indexing

This is a list of specific lexers that you can use to index Korean, Japanese, and Chinese languages.

Table 3-3 Lexers for Asian Languages

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

These lexers have their own sets of attributes to control indexing.

Related Topics

- *Oracle Text Reference*

3.8 About Entity Extraction and CTX_ENTITY

Entity extraction is the identification and extraction of named entities within text. Entities are mainly nouns and noun phrases, such as names, places, times, coded strings (such as phone numbers and zip codes), percentages, and monetary amounts. The `CTX_ENTITY` package implements entity extraction by means of a built-in dictionary and a set of rules for English text. You can extend the capabilities for English and other languages with user-provided add-on dictionaries and rule sets.

> **See Also:**
>
> - `CTX_ENTITY` Package in *Oracle Text Reference*
> - Entity Extraction User Dictionary Loader (ctxload) in *Oracle Text Reference*

This section contains the following examples:

- Basic Example of Using Entity Extraction

- Example of Creating a New Entity Type Using a User-defined Rule

3.8.1 Basic Example of Using Entity Extraction

The example in this section provides a very basic example of entity extraction. The example assumes that a CLOB contains the following text:

```
New York, United States of America
The Dow Jones Industrial Average climbed by 5% yesterday on news of a new software
release from database giant Oracle Corporation.
```

The example uses `CTX_ENTITY.EXTRACT` to find the entities in CLOB value. (For now, do not worry about how the text got into the CLOB or how we provide the output CLOB.)

Entity extraction requires a new type of policy, an "extract policy," which enables you to specify options. For now, create a default policy:

```
ctx_entity.create_extract_policy( 'mypolicy' );
```

Now you can call `extract` to do the work. It needs four arguments: the policy name, the document to process, the language, and the output CLOB (which you should have initialized, for example, by calling `dbms_lob.createtemporary`).

```
ctx_entity.extract( 'mypolicy', mydoc, 'ENGLISH', outclob )
```

In the previous example, `outclob` contains the XML that identifies extracted entities. When you display the contents (preferably by selecting it as XMLTYPE so that it is formatted nicely), here is what you see:

```
<entities>
  <entity id="0" offset="0" length="8" source="SuppliedDictionary">
    <text>New York</text>
    <type>city</type>
  </entity>
  <entity id="1" offset="150" length="18" source="SuppliedRule">
    <text>Oracle Corporation</text>
    <type>company</type>
  </entity>
  <entity id="2" offset="10" length="24" source="SuppliedDictionary">
    <text>United States of America</text>
    <type>country</type>
  </entity>
  <entity id="3" offset="83" length="2" source="SuppliedRule">
    <text>5%</text>
    <type>percent</type>
  </entity>
  <entity id="4" offset="113" length="8" source="SuppliedDictionary">
    <text>software</text>
    <type>product</type>
  </entity>
  <entity id="5" offset="0" length="8" source="SuppliedDictionary">
    <text>New York</text>
    <type>state</type>
  </entity>
</entities>
```

This display is fine if you process it with an XML-aware program. However, if you want it in a more "SQL friendly" view, use Oracle XML Database (XML DB) functions to convert it as follows:

```
select xtab.offset, xtab.text, xtab.type, xtab.source
from xmltable( '/entities/entity'
PASSING xmltype(outclob)
  COLUMNS
    offset number       PATH '@offset',
    lngth number        PATH '@length',
    text   varchar2(50) PATH 'text/text()',
    type   varchar2(50) PATH 'type/text()',
    source varchar2(50) PATH '@source'
) as xtab order by offset;
```

Here is the output:

```
    OFFSET TEXT                      TYPE               SOURCE
---------- ------------------------- ------------------ --------------------
         0 New York                  city               SuppliedDictionary
         0 New York                  state              SuppliedDictionary
        10 United States of America  country            SuppliedDictionary
        83 5%                        percent            SuppliedRule
       113 software                  product            SuppliedDictionary
       150 Oracle Corporation        company            SuppliedRule
```

If you do not want to fetch all entity types, you can select the types by adding a fourth argument to the "extract" procedure, with a comma-separated list of entity types. For example:

```
ctx_entity.extract( 'mypolicy', mydoc, 'ENGLISH', outclob, 'city, country' )

That would give us the XML

<entities>
  <entity id="0" offset="0" length="8" source="SuppliedDictionary">
    <text>New York</text>
    <type>city</type>
  </entity>
  <entity id="2" offset="10" length="24" source="SuppliedDictionary">
    <text>United States of America</text>
    <type>country</type>
  </entity>
</entities>
```

3.8.2 Example of Creating a New Entity Type by Using a User-Defined Rule

The example in this section shows how to create a new entity type with a user-defined rule. You define rules with a regular-expression-based syntax and add the rules to an extraction policy. The policy is applied whenever it is used.

The following rule identifies increases in a stock index by matching any of the following expressions:

```
  climbed by 5%
  increased by over 30 percent
  jumped 5.5%
```

Therefore, you must create a new type of entity as well as a regular expression that matches any of the expressions:

```
exec ctx_entity.add_extract_rule( 'mypolicy', 1,
    '<rule>'                                                          ||
      '<expression>'                                                  ||
         '((climbed|gained|jumped|increasing|increased|rallied)'      ||
         '( (by|over|nearly|more than))* \d+(\.\d+)?( percent|%))'    ||
      '</expression>'                                                 ||
```

```
      '<type>Positive Gain</type>'                                    ||
    '</rule>');
```

In this case, you must compile the policy with `CTX_ENTITY.COMPILE`:

```
ctx_entity.compile('mypolicy');
```

Then you can use it as before:

```
ctx_entity.extract('mypolicy', mydoc, null, myresults)
```

Here is the (abbreviated) output:

```
<entities>
  ...
  <entity id="6" offset="72" length="18" source="UserRule">
    <text>climbed by over 5%</text>
    <type>Positive Gain</type>
  </entity>
</entities>
```

Finally, you add another user-defined entity, but this time it uses a dictionary. You want to recognize "Dow Jones Industrial Average" as an entity of type `Index`. You also add "S&P 500". To do that, create an XML file containing the following:

```
<dictionary>
  <entities>
    <entity>
      <value>dow jones industrial average</value>
      <type>Index</type>
    </entity>
    <entity>
      <value>S&amp;P 500</value>
      <type>Index</type>
    </entity>
  </entities>
</dictionary>
```

Case is not significant in this file, but notice how the "&" in "S&P" must be specified as the XML entity `&`. Otherwise, the XML is not valid.

This XML file is loaded into the system with the `CTXLOAD` utility. If the file were called `dict.load`, you would use the following command:

```
ctxload -user username/password -extract -name mypolicy -file dict.load
```

You must compile the policy with `CTX_ENTITY.COMPILE`.

3.9 About Fuzzy Matching and Stemming

Use the `BASIC_WORDLIST` preference to enable query options, such as stemming and fuzzy matching for your language.

Overview

Fuzzy matching allows you to match words that have a similar spelling as the specified term. Oracle Text provides entity extraction for multiple languages.

Stemming enables indexing by the stem (same linguistic root as the specified *$term*). For example, you can index words like `speak`, `speaks`, `spoke`, and `spoken` by the term `speak`. The term `speak` is interpreted as the stem of those words.

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

Fuzzy Matching Attributes

Fuzzy matching (`fuzzy_match`) is enabled with default parameters for its fuzzy score and maximum number of expanded terms. Fuzzy score (`fuzzy_score`) is a measure of how closely the expanded word matches the query word. Fuzzy number results (`fuzzy_numresults`) specify the maximum number of fuzzy expansions. At index time, you can change these default parameters.

Stemming Attributes

- Language Attribute Values for `AUTO_LEXER`:

  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`. Use the stemmer that corresponds to the document language and always configure the stemmer to maximize document recall.

  For compound words in languages (for example, in German, Finnish, Swedish, or Dutch), if you set `composite` to `YES` (default value), then compound word stemming is automatically performed in documents. Compounds are always separated into their component stems.

- Language Attribute Values for `BASIC_LEXER`:

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

  Starting with Oracle Database 23ai, the old stemmer has been removed, making the `_NEW` suffix redundant. For example, `ENGLISH_NEW` is equivalent to `ENGLISH`.

  For compound words in languages (for example, in German, Finnish, Swedish, or Dutch), if you set `composite` to `YES` (default value), then compound word stemming is automatically performed in documents. Compounds are always separated into their component stems.

Related Topics

- BASIC_WORDLIST
- AUTO_LEXER Language Support
- AUTO_LEXER Language-Independent Attributes
- BASIC_LEXER Attributes

3.10 Better Wildcard Query Performance

Wildcard queries enable you to enter left-truncated, right-truncated, and double-truncated queries, such as *%ing, cos%,* or *%benz%.* With normal indexing, these queries can sometimes expand into large word lists and degrade 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"
> - *Oracle Text Reference* for more information on how to keep wildcard query performance within an acceptable limit

3.11 Document Section Searches

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

Before indexing, you must define sections and specify them 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:**
>
> Searching Document Sections in Oracle Text

3.12 Stopwords and Stopthemes

A *stopword* is a word that you do not want indexed. Stopwords are typically low-information words in a given language, such as *this* and *that* in English.

By default, Oracle Text provides a stoplist for indexing a given language. Modify this list or create your own with the `CTX_DDL` package. Specify the stoplist in the parameter string of the `CREATE INDEX` statement.

A *stoptheme* is a word that is prevented from being theme-indexed or that is prevented from contributing to a theme. Add stopthemes with the `CTX_DDL` package.

- Language detection and stoplists: 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).
- Multilanguage stoplists: You create multilanguage stoplists to hold language-specific stopwords. This stoplist is useful when you use `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, just as it determines the active lexer with the multi-lexer.

3.13 Index Performance

Factors that influence indexing performance include memory allocation, document format, degree of parallelism, and partitioned tables.

> **See Also:**
>
> "Frequently Asked Questions About Indexing Performance"

3.14 Query Performance and Storage of Large Object (LOB) Columns

If your table contains large object (LOB) structured columns that are frequently accessed in queries but rarely updated, you can improve query performance by storing these columns out-of-line. However, you cannot map attributes to remote LOB columns.

> **See Also:**
>
> "Does out-of-line LOB storage of wide base table columns improve performance?"

3.15 Mixed Query Performance

If your `CONTAINS()` query also has structured predicates on the nontext columns, then consider indexing those column values. To do so, specify those columns in the `FILTER BY` clause of the `CREATE INDEX` statement. Oracle Text can then determine whether to have the structured predicates processed by the Oracle Text index for better performance.

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

> **See Also:**
>
> "CONTEXT Index Example: Query Processing with FILTER BY and ORDER BY"

3.16 In-Memory Full Text Search and JSON Full Text Search

The queries using `CONTAINS()` and `JSON_TEXTCONTAINS()` can be evaluated in SQL predicates when the underlying columns that store the full text documents or JSON documents are enabled for In-Memory full text search.

Normally, to use full-text (keyword) searching against textual columns, you must create an Oracle Text index on that column. For JSON data, you create a JSON search index. Starting with Oracle Database Release 21c, instead of creating an index, you can load the column into memory, using an In-Memory columnar format. This does not require an index, but allows for fast scanning of the text using In-Memory techniques. This is particularly valuable when running queries which combine text searches and structured searches on other In-Memory columns.

You must declare the columns that must be loaded into memory during table creation time, using the `INMEMORY TEXT` clause. These columns can be searched using the same `CONTAINS()` and `JSON_TEXTCONTAINS()` functions that are used with Oracle Text or JSON search indexes, but there are limitations on the types of query operators that can be used. Hence, In-Memory is not a replacement for Oracle Text or JSON search indexes, but an alternative that can be used when required, and when the limitations are not considered to be a problem.

It is possible to have a column which has an Oracle Text index on it and also uses `INMEMORY TEXT` clause. In this situation, the optimizer chooses the Oracle Text index to execute the query. If there is an Oracle Text index on the column, the query always uses the Oracle Text index. If there is no Oracle Text index, then the optimizer checks if the table is marked as In-Memory. If the table is marked as In-Memory, the In-Memory evaluation is used for the query. If there is no Oracle Text index and the table is not marked as In-Memory, then the "`DRG-10599: column is not indexed`" error is returned.

Unlike `CONTEXT` indexes, you can use the `INMEMORY TEXT` clause with indirect datastore types (`NETWORK_DATASTORE` and `DIRECTORY_DATASTORE`) on `LOB` (large object) or `LONG` columns.

For detailed information on how to specify an in-memory Text column, see *Oracle Database In-Memory Guide*.

Supported Data Types

The In-Memory full text search supports the following data types:

- `CHAR`
- `VARCHAR2`
- `CLOB`
- `BLOB`
- `JSON`

Both JSON and text columns support a custom indexing policy created with the `CTX_DDL.CREATE_POLICY` procedure. If the column data type is `JSON`, then the In-Memory full text version of this column enables path-aware search using `JSON_TEXTCONTAINS()` when the column uses either of the following:

- A default policy
- A custom policy with a `PATH_SECTION_GROUP` having `JSON_ENABLED` attribute set to `TRUE`

Usage Notes

You specify an In-Memory full text search column with the `INMEMORY TEXT` clause. Both `CREATE TABLE` and `ALTER TABLE` statements support the `INMEMORY TEXT` clause. You can use the `PRIORITY` subclause to control the order of object population. The default priority is `NONE`. The `MEMCOMPRESS` subclause is not valid with `INMEMORY TEXT`.

Specify either the `CREATE TABLE` or `ALTER TABLE` statement with the `INMEMORY TEXT` clause, using either of the following forms:

- `INMEMORY TEXT (col1, col2, …)`
- `INMEMORY TEXT (col1 USING policy1, col2 USING policy2, …)`

Oracle recommends that you run in-memory repopulate operations after a batch of DML operations or before running any queries. You can use the `DBMS_INMEMORY.REPOPULATE` procedure that forces immediate repopulation of an object. See *Oracle Database PL/SQL Packages and Types Reference*.

You must set the following database initialization parameters when using the `INMEMORY TEXT` clause:

- `MAX_STRING_SIZE`: This parameter controls the maximum size of the `VARCHAR2`, `NVARCHAR2`, and `RAW` data types in SQL. You must set `MAX_STRING_SIZE` to `EXTENDED`. This setting raises the byte limit to `32767`, which requires shutting down or upgrading your database.
- `INMEMORY_EXPRESSIONS_USAGE`: This parameter controls the type of IM expression that the database populates. Set `INMEMORY_EXPRESSIONS_USAGE` to a value other than `DISABLE`:
  - `ENABLE` (default) to enable both static and dynamic IM expressions
  - `STATIC_ONLY` to enable only static IM expressions
- `INMEMORY_VIRTUAL_COLUMNS`: This parameter controls which user-defined virtual columns are stored as IM virtual columns. Set `INMEMORY_VIRTUAL_COLUMNS` to `ENABLE`, which is the default setting.

Limitations

Data Types	`BFILE`, `XMLType`, and `URIType` data types are not supported in In-Memory full text search columns. The `DIRECT_DATASTORE`, `DIRECTORY_DATASTORE`, and `NETWORK_DATASTORE` datastore types are supported. However, you cannot use `DIRECTORY_DATASTORE` and `NETWORK_DATASTORE` with a context index on the `CHAR` data type column.
Oracle Text Query Operators	For querying a text column, only the following Oracle Text query operators are supported: • `AND` • `OR` • `NOT` • `NEAR` For querying a JSON column, the following Oracle Text query operators are also supported: • `HASPATH` • `INPATH`

Policies	• In the `CTX_DDL.CREATE_POLICY` procedure, the `filter` parameter is not supported. All of the `BASIC_WORDLIST` attributes (such as `wildcard_index`, `stemmer`, `fuzzy_match`, or `substring_index`) are not supported. The `section_group` parameter must be set to either `NULL_SECTION_GROUP` (default) or `JSON_SECTION_GROUP` with `JSON_ENABLE` set to `TRUE` (for JSON enabled context indexes). The `lexer` parameter is supported only with the `BASIC_LEXER` lexer type. • JSON enabled indexing policies are supported only for JSON columns. • You can only use your own custom indexing policy for In-Memory full text search and JSON In-Memory full text search. Also, you can not use a JSON enabled indexing policy for text columns with `IS JSON` check constraint.
Disable or Enable In-Memory Full Text Search	You cannot disable and re-enable In-Memory full text search by using a single `ALTER TABLE` statement. You must first disable the In-Memory full text search before re-enabling it.

Examples

Example 3-1 Using In-Memory Full Text Search

The following example shows you how to query from an In-Memory full text search enabled column using the `CONTAINS` operator. It also shows you how to create a custom policy for text search and apply it on a column.

Create a table named `text_docs` that is loaded in memory and populate it with an In-Memory full text search column named `doc`:

```
CREATE TABLE text_docs(id NUMBER,  docCreationTime DATE, doc CLOB) INMEMORY
INMEMORY TEXT(doc);
```

Query using the `CONTAINS` operator with your condition:

```
SELECT id FROM text_docs WHERE docCreationTime >  to_date('2014-01-01', 'YYYY-
MM-DD')
AND CONTAINS(doc, 'in memory text processing');
```

You can also create a custom policy for text search, and then apply it to the `doc` column:

```
EXEC CTX_DDL.CREATE_POLICY('first_policy');
ALTER TABLE text_docs INMEMORY TEXT (doc USING 'first_policy');
```

You can replace an existing custom policy by disabling the In-Memory full text search using the `NO INMEMORY TEXT` clause and then enabling In-Memory full text search using the `INMEMORY TEXT` clause:

```
EXEC CTX_DDL.CREATE_POLICY('second_policy');
ALTER TABLE text_docs NO INMEMORY TEXT(doc);
ALTER TABLE text_docs INMEMORY TEXT (doc USING 'second_policy');
```

Example 3-2 Using JSON In-Memory Full Text Search

The following example shows you how to query from an In-Memory full text search enabled column using the `JSON_TEXTCONTAINS` operator.

Create a table named `json_docs` that is loaded in memory and populate it with an In-Memory full text search column named `doc`:

```
CREATE TABLE json_docs(id NUMBER,  docCreationTime DATE, doc JSON) INMEMORY
INMEMORY TEXT(doc);
```

Query using the `JSON_TEXTCONTAINS` operator with your condition:

```
SELECT id FROM json_docs WHERE docCreationTime >  to_date('2014-01-01', 'YYYY-
MM-DD')
AND JSON_TEXTCONTAINS(doc, '$.abstract', 'in memory text processing');
```

Example 3-3 Prioritizing In-Memory Population in Full Text Search

The following example shows you how to set the priority level for data population using the `PRIORITY` subclause.

Create a table named `prioritized_docs` that is loaded in memory and use the `PRIORITY` subclause to set the priority level:

```
CREATE TABLE prioritized_docs(id NUMBER,  docCreationTime DATE, doc CLOB,
json_doc CHECK(json_doc IS json))
INMEMORY PRIORITY CRITICAL INMEMORY TEXT(doc, json_doc);
```

Related Topics

- *Oracle Text Reference*
- *Oracle Database In-Memory Guide*
- *Oracle Database JSON Developer's Guide*

4
Creating Oracle Text Indexes

Learn how to create Oracle Text indexes.

This chapter contains the following topics:

4.1 Summary of the Procedure for Creating an Oracle Text Index

With Oracle Text, you can create indexes of type `CONTEXT`, `SEARCH INDEX`, `CTXCAT`, and `CTXRULE`.

By default, the system expects your documents to be stored in a text column. After you satisfy this requirement, you can create an Oracle Text index by using the `CREATE INDEX` SQL statement as an extensible index of type `CONTEXT`, without explicitly specifying preferences. The system automatically detects your language, the data type of the text column, and the format of the documents. Next, the system sets indexing preferences.

You can create a search index using the `CREATE SEARCH INDEX` SQL statement for indexing and querying structured, unstructured, or semi-structured data, such as textual, JSON, and XML documents. The `SEARCH INDEX` is an index type that supports the `CONTEXT` index functionality along with sharded databases and system-managed partitioning for index storage.

To create an Oracle Text index:

1. (Optional) Determine your custom indexing preferences, section groups, or stoplists if you do not use the 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?
Stoplist	What words or themes are not to be indexed?

Class	Description
Section Group	How are document sections defined?

2. (Optional) Create custom preferences, section groups, or stoplists.
3. Create the Oracle Text index with the `CREATE INDEX` SQL statement. Name your index and, if necessary, specify preferences.

Related Topics

- Considerations for Oracle Text Indexing
- 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.
- Creating Preferences
- CREATE_INDEX
- CREATE SEARCH INDEX

4.2 Creating Preferences

If you want, you can create 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 and then set the attributes.

> **See Also:**
>
> "Custom Index Preference Examples"

4.3 Section Searching Example: Creating HTML Sections

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

This example 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:**
>
> Searching Document Sections in Oracle Text

4.4 Using Stopwords and Stoplists

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

The system supplies 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 `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
- Stopthemes and Stopclasses
- PL/SQL Procedures for Managing Stoplists

4.4.1 Multilanguage Stoplists

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

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

4.4.2 Stopthemes and Stopclasses

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

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

You create a single stoplist, to which you add the stopwords, stopthemes, and stopclasses, and specify the stoplist in the `paramstring` for `CREATE INDEX`.

4.4.3 PL/SQL Procedures for Managing Stoplists

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 procedures

4.5 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
- Default CONTEXT Index Example
- Incrementally Creating a CONTEXT Index
- Custom CONTEXT Index Example: Indexing HTML Documents
- CONTEXT Index Example: Query Processing with FILTER BY and ORDER BY

4.5.1 CONTEXT Index and DML

A CONTEXT index is not transactional. When you delete a record, the index is changed immediately. That is, your session no longer finds the record from the moment you make the change, and other users cannot find the record after you commit. For inserts and updates, the new information is not 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"

4.5.2 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 the CREATE INDEX statement without explicitly specifying parameters, the system completes 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.

- Assumes that the language of the text to index is the language specified in your database setup.
- Uses the default stoplist for the language specified 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 customizing your preferences and specifying those preferences in the parameter string of `CREATE INDEX`.

> **See Also:**
>
> *Oracle Text Reference* to learn more about configuring your environment to use the `AUTO_FILTER` filter

4.5.3 Incrementally Creating a CONTEXT Index

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

You can incrementally create Oracle Text indexes, which means that the index structure is immediately created but the data is not populated during the index creation or rebuild process. You populate the index later at a suitable time. This procedure is useful for creating indexes in large installations that cannot afford to have the indexing process running continuously. It provides finer control over the creation of indexes, allowing you to avoid building indexes in a single operation.

Incremental index creation involves the following steps:

1. **Create an empty index**:

   If you specify the `NOPOPULATE` keyword at the time of index creation or rebuild, it only creates metadata for the index tables but does not populate them.

   - Global index:

     For a global index, use `CREATE INDEX` to support the `NOPOPULATE` keyword in the `REPLACE` parameter of the `REBUILD` clause.

   - Local index partition:

     For a local index partition, modify the `ALTER INDEX ... REBUILD partition ... parameters ('REPLACE ...')` parameter string to support the `NOPOPULATE` keyword.

For 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.

2. **Place all ROWIDs into the pending queue:**

   Use the `CTX_DDL.POPULATE_PENDING` procedure to populate the pending queues with every ROWID in the base table or table partition.

3. **Populate the index**:

   Use the `CTX_DDL.SYNC_INDEX` procedure to populate the index with the queued data.

   The `SYNC_INDEX` procedure includes the `maxtime` argument that indicates a suggested time limit in minutes for the operation. The indexing process runs in an estimate of the given `maxtime` instead of running to completion. You might need to run multiple `SYNC_INDEX` calls until the index is fully synced.

   You can choose to run both the `POPULATE_PENDING` and `SYNC_INDEX` calls separately so that the population of the pending queue and the population of the index happen at different times, thereby optimizing system performance.

Example 4-1 Incrementally Build an Empty Global Index

```
-- Create an empty index

CREATE INDEX ctx_ind ON ctx_tab(doc) INDEXTYPE IS CTXSYS.CONTEXT
   PARAMETERS ('NOPOPULATE');


declare
 n_pending number;

 function get_pending return number is
   n_pending number;

 begin
   n_pending := 0;
   begin
     execute immediate 'SELECT COUNT(*) FROM DR$CTX_IND$C' into n_pending;
   exception when others then
     if (sqlcode != -942) then
       raise;
     end if;
   end;

   if (n_pending = 0) then
     execute immediate 'SELECT COUNT(*) FROM CTX_USER_PENDING WHERE
PND_INDEX_NAME = :1'
         into n_pending using 'CTX_IND';
   end if;
   return n_pending;
 end get_pending;

begin
 -- Fill in the pending queue
 CTX_DDL.POPULATE_PENDING('CTX_IND');
 n_pending := get_pending;
 while (n_pending > 0) loop
```

```
  -- Populate the index through sync_index
   CTX_DDL.SYNC_INDEX('CTX_IND', maxtime => 1);
   n_pending := get_pending;
 end loop;
end;
/
```

Related Topics

- CREATE INDEX
- ALTER INDEX
- CTX_DDL.POPULATE_PENDING
- CTX_DDL.SYNC_INDEX

4.5.4 Custom CONTEXT Index Example: Indexing HTML Documents

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

You can also specify your `htmgroup` section group that uses `HTML_SECTION_GROUP` and `NETWORK_PREF` datastore that uses `NETWORK_DATASTORE`:

```
begin
 ctx_ddl.create_preference('NETWORK_PREF','NETWORK_DATASTORE');
 ctx_ddl.set_attribute('NETWORK_PREF','HTTP_PROXY','www-proxy.us.example.com');
 ctx_ddl.set_attribute('NETWORK_PREF','NO_PROXY','us.example.com');
 ctx_ddl.set_attribute('NETWORK_PREF','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:

```
CREATE INDEX myindex on docs(htmlfile) indextype is ctxsys.context
parameters(
'datastore NETWORK_PREF filter ctxsys.null_filter section group htmgroup'
);
```

> **Note:**
>
> Starting with Oracle Database 19c, the Oracle Text type `URL_DATASTORE` is deprecated. Use `NETWORK_DATASTORE` instead.

Related Topics

- Creating Preferences

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

To enable more efficient query processing and better response time for mixed queries, 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;
```

Because you specified the `FILTER BY category, publisher, pub_date` clause at query time, Oracle Text also considers pushing a relational predicate on any of these columns into the Oracle Text index row source.

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

4.6 Creating a CTXCAT Index

The `CTXCAT` index type is well-suited for indexing small text fragments and related information.

This index type provides better structured query performance than a `CONTEXT` index.

- CTXCAT Index and DML
- About CTXCAT Sub-Indexes and Their Costs
- Creating CTXCAT Sub-indexes
- Creating CTXCAT Index

> **Note:**
>
> The Oracle Text indextype `CTXCAT` is deprecated with Oracle Database 23ai. The indextype itself, and it's operator `CTXCAT`, can be removed in a future release. Both `CTXCAT` and the use of `CTXCAT` grammar as an alternative grammar for `CONTEXT` queries is deprecated. Instead, Oracle recommends that you use the `CONTEXT` indextype, which can provide all the same functionality, except that it is not transactional. Near-transactional behavior in `CONTEXT` can be achieved by using `SYNC(ON COMMIT)` or, preferably, `SYNC(EVERY [time-period])` with a short time period.
>
> `CTXCAT` was introduced when indexes were typically a few megabytes in size. Modern, large indexes, can be difficult to manage with `CTXCAT`. The addition of index sets to `CTXCAT` can be achieved more effectively by the use of `FILTER BY` and `ORDER BY` columns, or `SDATA`, or both, in the `CONTEXT` indextype. `CTXCAT` is therefore rarely an appropriate choice. Oracle recommends that you choose the more efficient `CONTEXT` indextype.

4.6.1 CTXCAT Index and DML Operations

A `CTXCAT` index is transactional. When you perform 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, do not result in automatic index synchronization as described in this section.

4.6.2 About CTXCAT Subindexes and Their Costs

A `CTXCAT` index contains subindexes that you define as part of your index set. You create a subindex on one or more columns to improve mixed query performance. However, 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 subindexes in the index set
- Number of columns in the base table that make up the subindexes

Many component indexes in your index set also degrade the performance of insert, update, and delete operations, because more indexes must be updated.

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

> **Note:**
>
> You can use `I_ROWID_INDEX_CLAUSE` of `BASIC_STORAGE` to speed up creation of a `CTXCAT` index. This clause is described in *Oracle Text Reference*.

4.6.3 Creating CTXCAT Subindexes

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

Figure 4-1 Auction Table Schema and CTXCAT Index

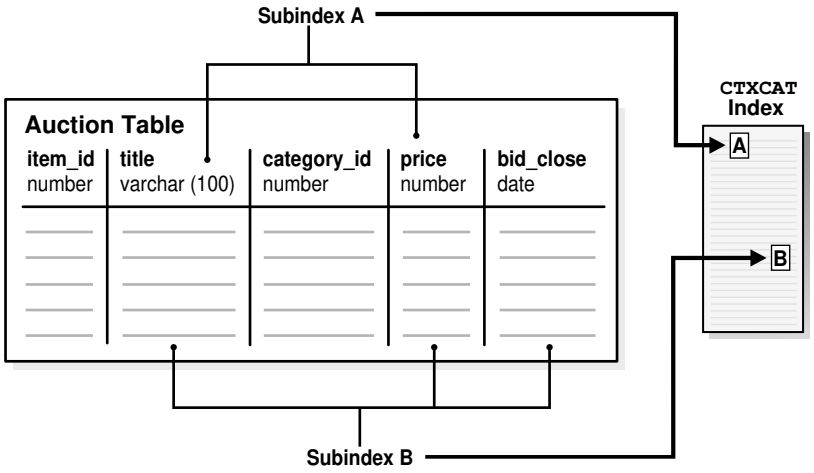


Figure 4-1 shows a table called `AUCTION` with the following schema:

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

To create your subindexes, create an index set to contain them:

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

Next, determine the structured queries that you are likely to enter. The `CATSEARCH` query operator takes a mandatory text clause and optional structured clause.

In the example, this means that 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	Subindex Definition to Serve Query	Category
'price < 200' 'price = 150' 'order by price'	'price'	A
'price = 100 order by bid_close' 'order by price, bid_close'	'price, bid_close'	B

Structured Query Clause Category A

The structured query clause contains an expression only for 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 by using subindex B. However, for efficiency, you can also create a subindex only on `price` (subindex A):

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

Structured Query Clause Category B

The structured query clause includes an equivalent 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 subindex 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 that you specify with `CTX_DDL.ADD_INDEX` affects the efficiency and viability of the index scan which 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'`.

4.6.4 Creating CTXCAT Index

This example combines the previous examples and creates the index set preference with the two subindexes:

```
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 4-1 shows how the subindexes A and B are created from the auction table. Each subindex is a b-tree index on the text column and the named structured columns. For example, subindex A is an index on the `title` column and the `bid_close` column.

You create the combined catalog index with the `CREATE INDEX` statement 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`

4.7 Creating a CTXRULE Index

To build a document classification application, use the CTXRULE index on a table or queries. The stream of incoming documents is classified by content, and the queries define your categories. You can use the MATCHES operator to classify single documents.

To create a CTXRULE index and a simple document classification application:

1. Create a table of queries.

   Create a myqueries table to hold the category name and query text, and then populate the table with the classifications and the queries that define each classification.

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

   For example, consider a classification for the *US Politics, Music,* and *Soccer* subjects:

   ```
   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)');
   ```

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

2. Create the CTXRULE index.

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

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

3. Classify a document.

   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);
   ```

   If you want, create a "before insert" trigger with MATCHES to route each document to a news_route table 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;
```

> **See Also:**
>
> - Classifying Documents in Oracle Text for more information on document classification and the `CTXRULE` index
> - *Oracle Text Reference* for more information on `CTX_CLS.TRAIN`

4.8 Creating a JSON Search Index

Oracle Text supports a simpler alternative syntax for creating a search index on JavaScript Object Notation (JSON). The JSON search index is created on the table column name.

When creating a JSON search index, you can specify **path subsetting** to identify the JSON fields to include or exclude from indexing. The excluded fields are not indexed, and the JSON search index is not used for those fields when querying.

Related Topics

- *Oracle Database JSON Developer's Guide*
- *Oracle Text Reference*

4.9 Creating an Oracle Text Search Index

You can create a `CONTEXT` index using a simplified `SEARCH INDEX` syntax.

The Oracle Text `SEARCH INDEX` is a new index type which supports `CONTEXT` index functionality but also supports sharded databases and system managed partitioning for index storage.

> **See Also:**
>
> *Oracle Text Reference* for more information about `CREATE SEARCH INDEX`

4.10 Creating a Hybrid Vector Index

You can create a hybrid vector index using the `CREATE HYBRID VECTOR INDEX` syntax.

A hybrid vector index inherits all the information retrieval capabilities of Oracle Text search indexes and leverages the semantic search capabilities of Oracle AI Vector Search vector indexes. These indexes allow you to index and query documents using a combination of full-text search and semantic vector search.

Related Topics

- *Oracle Text Reference*
- *Oracle Database AI Vector Search User's Guide*

5
Maintaining Oracle Text Indexes

Learn how to manage indexing errors or failures, re-create or rebuild indexes, drop custom index preferences, manage synchronization and optimization tasks, and automate index maintenance operations.

5.1 Viewing Index Errors

Sometimes an indexing operation might fail or it might not complete successfully. When the system encounters an error during row indexing, 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 index error views

5.2 Dropping an Index

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

Drop an index by 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, because of an indexing malfunction), you cannot drop the index. Instead use:

```
DROP INDEX newsindex FORCE;
```

> **See Also:**
>
> *Oracle Text Reference* to learn more about the `DROP INDEX` statement

5.3 Resuming a Failed Index

You can sometimes resume a failed index by using the `ALTER INDEX` statement. You typically resume a failed index after you have investigated and corrected the index failure. You cannot resume all index failures.

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

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

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

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

5.4 Re-creating an Index

This section describes the procedures for re-creating an index. During the re-creation process, you can query the index normally.

- Re-creating a Global Index
- Re-creating a Local Partitioned Index

5.4.1 Re-creating a Global Index

Oracle Text provides `RECREATE_INDEX_ONLINE` to re-create a `CONTEXT` index with new preferences, while preserving inserts, updates, and deletes on the base table. You can use `RECREATE_INDEX_ONLINE` in a single-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 that is roughly equal to the size of the existing index. Also, because the `RECREATE_INDEX_ONLINE` operation is performed online, you can perform inserts, updates, and

deletes on the base table during the operation. All insert, update, and delete operations that occur during the re-creation process are logged into an online pending queue.

- After the re-creation operation is complete, new information may not be immediately reflected. As with creating an index online, you should synchronize the index after the re-creation operation is complete to bring it fully up-to-date.
- Synchronizations issued against the index during the re-creation operation are processed against the existing data. Synchronizations are blocked when queries return errors.
- Optimize commands issued against the index during the re-creation operation return immediately without error and without processing.
- During `RECREATE_INDEX_ONLINE`, you can query the index normally most of the time. Queries return results based on the existing index and policy until after the final swap. Also, if you issue insert, update, and delete operations and synchronize them, then you will be able to see the new rows when you query the existing index.

> **Note:**
>
> Transactional queries are not supported with `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 nonbusiness hours and incrementally re-create the index. Use the `CREATE_SHADOW_INDEX` procedure with `POPULATE_PENDING` and maxtime.

Re-creating a Global Index with Scheduled Swap

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

> **See Also:**
>
> - *Oracle Text Reference* to learn more about the `RECREATE_INDEX_ONLINE` procedure
> - *Oracle Text Reference* for information and examples for `CREATE_SHADOW_INDEX`
> - *Oracle Text Reference* for information and examples for `CTX_DDL.EXCHANGE_SHADOW_INDEX`

5.4.2 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 swaps the index partition data and index partition metadata.

You can also use this procedure to update the metadata (for example, the storage preference) of each partition when you specify `NOPOPULATE` in the parameter string. This keyword is useful

for incremental building of a shadow index through time-limited synchronization. If you specify NOPOPULATE, then NOSWAP is silently enforced.

- When all partitions use NOSWAP, the storage requirement is approximately equal to the size of the existing index. During re-creation of the index partition, because 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. Because index partition data and metadata are swapped after re-creation, queries spanning multiple partitions do not return consistent results from partition to partition, but they will always be correct with respect to each index partition.
- If you specify SWAP, then insert, update, and delete operations and synchronization on the partition are blocked during the swap process.

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 the re-creation 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.

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, and each partition is 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 complete information about RECREATE_INDEX_ONLINE

5.5 Rebuilding an Index

You can rebuild a valid index by 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 the CREATE INDEX statement.

> **See Also:**
>
> "Re-creating an Index" for information about changing index settings

The following statement rebuilds the index and replaces the lexer preference with my_lexer:

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

5.6 Dropping a Preference

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

You drop index preferences with the `CTX_DDL.DROP_PREFERENCE` procedure.

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

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

The following code drops the `my_lexer` preference:

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

5.7 Managing DML Operations for a CONTEXT Index

DML operations refer to when documents are inserted, updated, or deleted from the base table.

This section describes how you can view, synchronize, and optimize the Oracle Text `CONTEXT` index for DML operations. This section contains the following topics:

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

5.7.1 Viewing Pending DML Operations

When you insert, update, or delete documents in the base table, their ROWIDs are held in a DML queue until you synchronize the index.

You can view the DML queue by querying index tables, as follows:

- When the index is created using the default `fast_dml` option and when the `COMPATIBLE` database parameter is set to a value lower than `20.0`, the `CTXSYS.DR$PENDING` table keeps track of pending DMLs. You can query pending insert and update operations with the `CTX_PENDING` and `CTX_USER_PENDING` views.
- When the index is created using the default `fast_dml` option and when the `COMPATIBLE` database parameter is set to `20.0` or higher, the `DR$`*`INDEX_NAME`*`$C` table stores information on ROWIDs that are waiting for synchronization into the index.

The indexes that are set to `MAINTENANCE AUTO` (automatic maintenance) or `SYNC EVERY` are automatically synchronized, however you can periodically examine the pending DML tables to determine whether a synchronization call is failing. For example, if your query results appear incorrect or outdated, then you can check if the documents have been synchronized and accordingly run a manual `SYNC_INDEX` call if required.

For example, to view pending DML operations on your indexes, enter the following command:

```
SELECT COUNT(*) FROM myschema.dr$myindex$c;
```

The output appears as follows:

```
COUNT(*)
----------
1
```

To retrieve ROWIDs of all unsynchronized changes from the `$C` table, enter the following command:

```
SELECT dml_rid FROM myschema.dr$myindex$c;
```

The output appears as follows:

```
DML_RID
------------------
AAAVP9AAMAAAAKGAAD
```

You can run `CTX_DDL.SYNC_INDEX` to synchronize your index, and then check if the `$C` table has been cleared:

```
EXEC CTX_DDL.SYNC_INDEX('myschema.myindex');


SELECT COUNT(*) FROM myschema.dr$myindex$c;


COUNT(*)
----------
0
```

5.7.2 Synchronizing the Index

When you synchronize the index, you process all pending updates and inserts to the base table. You can do this in PL/SQL with the `CTX_DDL.SYNC_INDEX` procedure. You can also control the duration and locking behavior for index synchronization with the `CTX_DDL.SYNC_INDEX` procedure.

Synchronizing the Index with SYNC_INDEX

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

```
begin

ctx_ddl.sync_index('myindex', '2M');

end;
```

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 procedure processes 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 may be less than, or greater than, what you specify.
- The ALTER INDEX... sync command has no changes because it 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 works when another synchronization 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 synchronizations (that is, sync on commit or sync every).
- When the locking mode is LOCK_WAIT, the mode waits forever and ignores the maxtime setting if it cannot get a lock.

The options are as follows:

Option	Description
CTX_DDL.LOCK_WAIT	If another SYNC_INDEX is running, wait until the running synchronization is complete, and then begin the new synchronization.
CTX_DDL.LOCK_NOWAIT	If another SYNC_INDEX is running, immediately return without error.
CTX_DDL.LOCK_NOWAIT_ERROR	If another SYNC_INDEX is running, immediately generate an error (DRG-51313: timeout while waiting for inserts, updates, or deletes or optimize lock).

> **Note:**
>
> Starting with Oracle Database 12*c* Release 2 (12.2.0.1), you automatically merge rows from STAGE_ITAB back to the $I table by using SYNC_INDEX. This merging of rows happens when the number of rows in STAGE_ITAB ($G) exceeds the STAGE_ITAB_MAX_ROWS parameter (10K by default). Therefore, you do not have to run merge optimization explicitly or schedule an auto optimize job.

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

5.7.3 Optimizing the Index

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 the following entry:

```
DOG DOC1 DOC3 DOC5
```

Frequent index synchronization ultimately causes fragmentation of your `CONTEXT` index. Index fragmentation can adversely affect query response time. Therefore, to reduce fragmentation and index size and to ensure optimal query performance, allow time to optimize your `CONTEXT` index.

To schedule an auto optimize job, you must explicitly set `STAGE_ITAB_MAX_ROWS` to `0` to disable the automatic merging that now happens with `SYNC_INDEX`.

To optimize an index, Oracle recommends that you use `CTX_DDL.OPTIMIZE_INDEX`. To understand index optimization, you must understand the structure of the index and what happens when it is synchronized. This section contains the following topics:

- Index Fragmentation
- Document Invalidation and Garbage Collection
- Single Token Optimization
- Viewing Index Fragmentation and Garbage Data

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

5.7.3.1 Index Fragmentation

When you add new documents to the base table, the index is synchronized by adding new rows. For example, if you add the DOC 7 document with the word *dog* and synchronize the index, you now have:

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

Subsequent inserts, updates, or deletes also create new rows, as follows:

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

Index fragmentation occurs when you add new documents and synchronize the index. In particular, background inserts, updates, or deletes, which synchronize the index frequently, generally produce more fragmentation than batch mode synchronization.

When you perform batch processing less frequently, you reduce fragmentation because you produce longer document lists with a reduced number of rows in the index.

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

5.7.3.2 Document Invalidation and Garbage Collection

When you remove documents 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 process is called **garbage collection.** Optimizing in `FULL` mode for garbage collection is necessary when you perform frequent updates or deletes to the base table.

5.7.3.3 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`.

5.7.3.4 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 the `CTX_REPORT.INDEX_STATS` procedure

5.8 Using Automatic Maintenance for an Index

Instead of manually managing synchronization tasks for your indexes, you can automate `CTX_DDL.SYNC_INDEX` operations using automatic maintenance.

- About Automatic Maintenance
- Requirements and Restrictions for Automatic Maintenance
- Asynchronous Maintenance Framework

- Enabling and Disabling Automatic Maintenance
- Switching between Automatic and Manual Maintenance
- Monitoring Maintenance Events and Errors

5.8.1 About Automatic Maintenance

Indexes with automatic maintenance are synchronized in the background without any user intervention.

Overview

Index maintenance is the process of updating index data structures (in-memory and on-disk) as a result of performing DML operations.

Automatic maintenance is the default method for synchronizing Oracle Text `CONTEXT` and search indexes (Oracle Text, JSON, and XML search indexes) that are created in Oracle Database 23ai and later releases.

Both the automatic maintenance and synchronization (`SYNC`) methods involve processing pending updates, inserts, and deletes to the base table. However, the automatic maintenance and `SYNC` specifications are orthogonal. Automatic maintenance uses an asynchronous maintenance framework to perform `SYNC` operations in the background, and provides the following capabilities:

- **Eliminates time-based or manual SYNC operations**:

  In an automatic maintenance mode, IRnn background processes automatically perform index maintenance operations in an optimal manner. This feature internally determines an optimal synchronization interval (based on the DML arrival) and automatically schedules background `SYNC` operations, as required. You cannot override the automatically determined intervals.

  For detailed information about this background mechanism process, see Asynchronous Maintenance Framework.

- **Reduces the frequency of background jobs**:

  Background processes maintain indexes rather than the database scheduler. The background mechanism breaks each `CTX_DDL.SYNC_INDEX` operation into separate events (sync stages) and launches each event only when needed.

- **Provides the default maintenance configuration**:

  These indexes do not require you to configure a `SYNC` type or set any synchronization interval. By default, indexes are configured with a combination of automatic maintenance and `SYNC (MANUAL)`. No other `SYNC` settings are compatible with these indexes.

  Note that the `SYNC (MANUAL)` behavior is different in this mode. Unlike the regular `SYNC (MANUAL)` type (where you must manually synchronize an index), here `CTX_DDL.SYNC_INDEX` is automatically called in the background at optimal intervals.

Why and When to Use Automatic Maintenance?

Oracle recommends that you use automatic maintenance in cases where sync requirements for indexes are not clear or you want to synchronize a large number of indexes in an optimal manner.

In addition to reducing the administrative tasks of managing your indexes, the benefit of using this framework is that it automatically determines when a background `SYNC` operation needs to

be performed, by tracking the DML queue. It also provides more control over the frequency of different background jobs running at any given time, instead of creating independent jobs for each index or index partition per pluggable database (PDB). As a result, automatic maintenance helps in reducing the workload on database resources, eliminates scheduling conflicts, and enhances query performance.

With `SYNC (EVERY)`, which also enables automatic background synchronization, you must manually specify sync interval using *`interval-string`*. Although `SYNC (EVERY)` allows you to explicitly control the synchronization interval, automatic maintenance provides an efficient usage of database resources especially when supporting multiple PDBs. In addition, `SYNC (EVERY)` may result in excessive launching of background sync jobs, based on the user's estimate of how frequently new index data may arrive.

What is Manual Maintenance?

Manual maintenance is a non-automatic maintenance mode that provides the pre-release 23ai synchronization behavior.

In a manual maintenance mode, you can specify `SYNC` types, such as `SYNC MANUAL`, `SYNC EVERY` *`interval-string`*, or `SYNC ON COMMIT`. The `MAINTENANCE MANUAL` index parameter sets your indexes to manual maintenance.

After upgrading to a new release, existing indexes continue to use the previously specified method of synchronization. For example, after upgrading to Oracle Database 23ai, existing indexes are set to manual maintenance with the previously specified `SYNC` settings. If you did not specify any `SYNC` setting before the upgrade, then the index uses the default `SYNC` type. That is, `SYNC MANUAL` for Oracle Text `CONTEXT` indexes and `SYNC ON COMMIT` for JSON and XML search indexes. If required, you can manually enable automatic maintenance for such indexes.

How to Configure the MAINTENANCE Parameter?

The `MAINTENANCE` parameter controls the maintenance type (mode) for your index. You can set the `MAINTENANCE` parameter globally and not per partition. This means that the maintenance type specified for an index applies to all index partitions.

The supported maintenance types are:

- `MAINTENANCE AUTO` (default): Sets your indexes to automatic maintenance.

  By default, you do not need to configure automatic maintenance while creating an index. This example creates a JSON search index that has the default behavior (no `PARAMETERS` clause):

  ```
  CREATE SEARCH INDEX po_search_idx ON j_purchaseorder (po_document)
      FOR JSON;
  ```

  This example creates a JSON search index by explicitly specifying `MAINTENANCE AUTO` using the `PARAMETERS` clause:

  ```
  CREATE SEARCH INDEX po_search_idx ON j_purchaseorder (po_document)
      FOR JSON PARAMETERS ('MAINTENANCE AUTO');
  ```

- `MAINTENANCE MANUAL`: Sets your indexes to manual maintenance.

This example disables automatic maintenance on a new JSON search index by specifying `MAINTENANCE MANUAL` using the `PARAMETERS` clause:

```
CREATE SEARCH INDEX po_search_idx ON j_purchaseorder (po_document)
    FOR JSON PARAMETERS('MAINTENANCE MANUAL');
```

For detailed information about configuring these parameters, see Enabling and Disabling Automatic Maintenance.

You can switch between the automatic and manual maintenance modes using `ALTER INDEX`. This command alters only the synchronization options, and thus you do not need to rebuild the index. When set to manual maintenance, if you do not explicitly specify any `SYNC` type, then the index uses the default `SYNC` type.

5.8.2 Requirements and Restrictions for Automatic Maintenance

Review these requirements and restrictions (such as database compatibility, supported parameter combinations, and supported indexes) when using the automatic maintenance mode.

- Database compatibility for the asynchronous maintenance framework is Oracle Database 21.0.0.0 and later.
- The combination of automatic maintenance with the following parameters is not supported:
  - `FAST_QUERY`
  - `ASYNCHRONOUS_UPDATE`
  - `TRANSACTIONAL`
  - `SYNC (ON COMMIT)` and `SYNC (EVERY)`

  Running any of the preceding combinations results in an error, prompting you to use a compatible mode for your index.
- Shadow indexes do not support automatic maintenance.

5.8.3 Asynchronous Maintenance Framework

In an automatic maintenance mode, IRnn background processes perform index maintenance operations, which provides a better scalability of background jobs and enhances query performance.

List of Maintenance Events

Each `SYNC` operation consists of separate events (stages) that can concurrently run in the background.

Event	Description
`SYNC-Mapping` (Sync-M)	Reads the $B catalog table to find the next DocID, and allocates DocID to each ROWID. Next, it reads the contents of the $C commit journal, sorts it on ROWID, and decides which DocIDs must be removed or added. Next, it performs deletes and inserts on the $K mapping table. Then, it adds the removed DocIDs to the $N garbage collection table. Finally, it deletes all the rows read from $C. During a failure, these events are retried and also broadcasted to other Oracle Real Application Clusters (Oracle RAC) nodes.

Event	Description
SYNC-Mapping Timeout (Sync-MT)	Specifies the Sync-M timeout events, generated on a commit. These are processed only by timeout actions and not by interrupt actions. During a timeout, the Sync-MT events are converted into the Sync-M events.
SYNC-Ranges (Sync-R)	Assigns DocID ranges generated by Sync-M as READY for postings stage. It reads all available NEW ranges to determine the minimum of the first DocID and the maximum on the last DocID. Next, it deletes all NEW ranges and inserts a number of equally-sized READY ranges. The size of the range is determined based on the average size of each document for the index or index partition. During a failure, these events are retried and also broadcasted to other RAC nodes.
SYNC-Scheduler (Sync-S)	Schedules Sync-P based on the number of READY ranges generated by Sync-R. Depending on the value, it schedules either a serial or concurrent Sync-P event.
SYNC-Postings (Sync-P)	Generates postings lists that contain new index data. Sync-P starts by getting a READY range. During scheduling, it decides the number of workers for running the event. On Oracle RAC systems, concurrent events are also broadcasted to other nodes for running as the SYNC-Postings Concurrent (Sync-PC) events. During a failure, the Sync-S events are scheduled with increased iteration. On Oracle RAC systems, the failed Sync-PC and SYNC-Postings Serial (Sync-PS) events are broadcasted as Sync-PS events.
SYNC-Postings Serial (Sync-PS)	Runs Sync-P serially. The postings are generated in SGA batches.
SYNC-Postings Concurrent (Sync-PC)	Runs Sync-P concurrently. Can schedule multiple ranges to run concurrently and independently without contention.
SYNC-Writer (Sync-W)	Writes SGA batches of postings lists to disk. Sync-W events can run concurrently with itself. However, these events cannot run with SYNC-Cleanup batches (Sync-C). Sync-W events are never broadcasted to other RAC nodes because they process SGA batches, which are local to the node that generated them.
SYNC-Cleanup batches (Sync-C)	Cleans up WRITE ranges in $B that do not have any SGA batches associated with them due to a failure. These events are retried in the subsequent run of Sync-P.
SYNC-Inspect (Sync-I)	Inspects index and index partitions (checks the $C and $B tables) to find if any events are missing. During a failure, the Sync-I events are not retried and are also not broadcasted to other RAC nodes.
MONITOR	Schedules the Sync-I events for each index or index partition and the EClean events for each pluggable database (PDB). The Scheduler schedules the actual Monitor event, which is further processed by the Monitor worker.
EVENT Stats (EStat)	Terminal event that is processed by Writer workers (Sync-W events). It writes event stats for all completed events in a PDB.
EVENT Stats Clean up (EClean)	Cleans up persisted event stats (older than the PDB-specific threshold) from the dictionary.
OPTIMIZE-Scheduler Timeout (Opti-ST)	Prevents dealing with large postings lists gaps that are crated after the order completion of the Sync-W events.
OPTIMIZE-Scheduler (Opti-S)	Determines the maximum DocID for which there are no gaps in postings lists due to the out-of-order processing of the WRITE ranges by the Sync-W events. The Opti-S event aggregates $G token counts that are stored in $B by the Sync-W events, and it schedules the Opti-M event when the aggregate count reaches a user-specified threshold.

Event	Description
OPTIMIZE-Merge (Opti-M)	Terminal event that does not perform the actual MERGE operation. Instead, it schedules the DBMS_SCHEDULER Optimize Merge operation.

Background Mechanism Process

The main **Scheduler** background process checks for the workload from all indexes at predefined intervals (that is, every 3 seconds by default). It then assigns the workload to a **Worker** background process, which reads events one at a time and processes them based on the event type. The index is synchronized immediately after the Worker process runs CTX_DDL.SYNC_INDEX. Apart from the Scheduler and Worker processes, the **Monitor** background process helps in recovering lost events.

A CTX_DDL.SYNC_INDEX call performs the following steps in an order:

1. Resets all waiting events for an index or index partition:

   When an event fails, it adds the event to the Waiting Queue with progressively increasing delay. Even after the issue is corrected, the event continues to wait for the current delay to elapse. To track such delays, it schedules retry events at an incremented level. This means that the Scheduler process first moves a retry event into an Event Queue. From the Event Queue, the Scheduler moves it into the Waiting Queue, then to the Ready Queue, and finally allocates the event to the Worker process.

   In addition to incremental levels, each event has a retry iteration. On long retries (that is, when a retry event is not the same as the original event), it increments the iteration instead of the level and initializes the level to the iteration.

   CTX_DDL.SYNC_INDEX can force immediate re-execution of the event, which moves all relevant Waiting Queue events to the Event Queue and also resets the level and iteration. If the event fails again, then it restarts from the starting incremental level or iteration.

2. Performs Sync-M in the foreground:

   CTX_DDL.SYNC_INDEX waits for the background maintenance to finish. However, instead of waiting for all events to finish, it calls Sync-M in the foreground and gets the maximum DocID that is allocated. It uses this DocID to ignore all future events.

3. Schedules other stages of SYNC in the background:

   CTX_DDL.SYNC_INDEX posts the Scheduler process so that it can immediately start processing pending events.

4. Waits for the completion of background processing:

   The waiting is controlled by the locking parameter when it is set to CTX_DDL.LOCK_WAIT. For all other values, CTX_DDL.SYNC_INDEX returns after completing Sync-M.

   The values of the memory, parallel_degree, maxtime, and direct_path parameters are ignored.

   If some background events are delayed or cannot complete, then CTX_DDL.SYNC_INDEX returns ORA-30608 and logs an error message in the catalog views.

Differences in SYNC Behavior Between Automatic Maintenance and Manual Maintenance

Compare the differences in the synchronization behavior between automatic maintenance and manual maintenance, and how different events are processed during a CTX_DDL.SYNC_INDEX operation:

Behavior	Automatic Maintenance	Manual Maintenance
Background mechanism	In an automatic maintenance mode, background processes maintain indexes. The background mechanism breaks each SYNC operation into separate events that are run concurrently with each other, as needed. For example, Sync-S launches Sync-P to pick up new index data only when Sync-R generates READY ranges.	In a manual maintenance mode, the DBMS_SCHEDULER background jobs maintain indexes. The background mechanism implements all sync events (Sync-M, Sync-R, and Sync-P) together as a single SYNC operation.
SYNC types	These indexes are preconfigured with a combination of automatic maintenance and SYNC (MANUAL). Unlike the regular SYNC (MANUAL) type (where you must manually call CTX_DDL.SYNC_INDEX), here CTX_DDL.SYNC_INDEX is automatically called in the background at optimal intervals. The other SYNC types, such as SYNC ON COMMIT and SYNC EVERY are not supported with automatic maintenance.	Running SYNC (MANUAL), SYNC (ON COMMIT), or SYNC (EVERY) launches a SYNC operation in the foreground for each index or index partition.
Catalog views	• CTX_BACKGROUND_EVENTS • CTX_USER_BACKGROUND_EVENTS • V$TEXT_WAITING_EVENTS	• CTX_AUTOSYNC_JOBS • CTX_AUTOSYNC_STATUS • CTX_USER_AUTOSYNC_JOBS • CTX_USER_AUTOSYNC_STATUS

Related Topics

- Monitoring Maintenance Events and Errors
  The SYS and CTXSYS users can query catalog views to monitor the status of all background maintenance events for indexes with automatic maintenance.

5.8.4 Enabling and Disabling Automatic Maintenance

Automatic maintenance is enabled by default for new Oracle Text CONTEXT and search indexes. Learn how to explicitly specify automatic maintenance while creating an index, or disable it to override the default behavior and enable SYNC instead.

1. To explicitly specify automatic maintenance for a new index, use the MAINTENANCE AUTO keyword in the PARAMETERS clause of the CREATE INDEX or CREATE SEARCH INDEX statement.
   - For an Oracle Text index:

     ```
     CREATE INDEX CTX_IDX ON CTX_TAB(DOC)
         INDEXTYPE IS CTXSYS.CONTEXT
         PARAMETERS('MAINTENANCE AUTO');
     ```

   - For an Oracle Text search index:

     ```
     CREATE SEARCH INDEX CTX_IDX ON CTX_TAB(DOC)
         PARAMETERS('MAINTENANCE AUTO');
     ```

- For a JSON search index:

```
CREATE SEARCH INDEX JSON_IDX ON CTX_TAB(JSON_DOC) FOR JSON
    PARAMETERS('MAINTENANCE AUTO');
```

- For an XML search index:

```
CREATE SEARCH INDEX XML_IDX ON CTX_TAB(XML_DOC) FOR XML
    PARAMETERS('MAINTENANCE AUTO');
```

2. To disable automatic maintenance for a new index, use the `MAINTENANCE MANUAL` keyword in the `CREATE INDEX` or `CREATE SEARCH INDEX` clause.

   This will set your index to manual maintenance. If you do not specify any `SYNC` type, then the index will use the default `SYNC` settings. For example, `SYNC MANUAL` for Oracle Text `CONTEXT` indexes and `SYNC ON COMMIT` for JSON and XML search indexes.

   - For an Oracle Text index:

```
CREATE INDEX CTX_IND ON CTX_TAB(DOC)
INDEXTYPE IS CTXSYS.CONTEXT
PARAMETERS('MAINTENANCE MANUAL');
```

   - For an Oracle Text search index:

```
CREATE SEARCH INDEX CTX_IDX ON CTX_TAB(DOC)
    PARAMETERS('MAINTENANCE MANUAL');
```

   - For a JSON search index:

```
CREATE SEARCH INDEX JSON_IDX ON CTX_TAB(JSON_DOC) FOR JSON
    PARAMETERS('MAINTENANCE MANUAL');
```

   - For an XML search index:

```
CREATE SEARCH INDEX XML_IDX ON CTX_TAB(XML_DOC) FOR XML
    PARAMETERS('MAINTENANCE MANUAL');
```

3. To override the default `SYNC` settings for the index set to manual maintenance, specify the required `SYNC` type:

   - `MANUAL`: To manually synchronize the index on demand.

     For example:

```
ALTER INDEX CTX_IDX REBUILD
    PARAMETERS('REPLACE METADATA MAINTENANCE MANUAL);
```

   - `ON COMMIT`: To synchronize the index immediately after a commit.

     For example:

```
ALTER INDEX CTX_IDX REBUILD
    PARAMETERS('REPLACE METADATA SYNC(ON COMMIT) MAINTENANCE MANUAL');
```

   - `EVERY "interval-string"`: To synchronize the index at a regular interval.

For example, starting every 20 minutes:

```
ALTER INDEX CTX_IDX REBUILD
    PARAMETERS('REPLACE METADATA SYNC(EVERY
"freq=minutely;interval=20") MAINTENANCE MANUAL');
```

Related Topics

- Switching between Automatic and Manual Maintenance
  You can use `ALTER INDEX` to switch between `MAINTENANCE AUTO` and `MAINTENANCE MANUAL`, without rebuilding the index. While changing modes, you must specify compatible Maintenance type and `SYNC` type combinations.
- *Oracle Text Reference*

5.8.5 Switching between Automatic and Manual Maintenance

You can use `ALTER INDEX` to switch between `MAINTENANCE AUTO` and `MAINTENANCE MANUAL`, without rebuilding the index. While changing modes, you must specify compatible Maintenance type and `SYNC` type combinations.

Guidelines for Switching between Modes

- `MAINTENANCE AUTO` is supported with `SYNC (MANUAL)`. By default, all indexes with automatic maintenance are specified with this combination.
- A combination of `MAINTENANCE AUTO` and `SYNC ON COMMIT` or `SYNC (EVERY)` is not supported. If you want to specify `MAINTENANCE AUTO` for indexes that also use `SYNC (ON COMMIT)` or `SYNC (EVERY)`, then you must first set such indexes to `SYNC (MANUAL)`.
- Static dictionary view `CTX_USER_INDEXES` contains information about existing Oracle Text `CONTEXT` and search indexes for the current user. For example, this query lists the `SYNC` and Maintenance types for an Oracle Text index set to `MAINTENANCE AUTO`:

```
SQL> SELECT IDX_NAME, IDX_SYNC_TYPE, IDX_MAINTENANCE_TYPE FROM
CTX_USER_INDEXES;

IDX_NAME                IDX_SYNC_TYPE               IDX_MAINTENANCE_TYPE
----------------------- --------------------------- ---------------------
DOCIDX                  MANUAL                      AUTO
```

Switching Indexes to Automatic Maintenance

This table provides `ALTER INDEX` examples with various `SYNC` types, while altering your index from `MAINTENANCE MANUAL` to `MAINTENANCE AUTO`:

Setting	Maintenance and SYNC Type	Example
`SYNC (MANUAL)` to `MAINTENANCE AUTO`	`IDX_SYNC_TYPE: MANUAL` `IDX_MAINTENANCE_TYPE: AUTO`	`ALTER INDEX CTX_IDX REBUILD PARAMETERS('REPLACE METADATA MAINTENANCE AUTO');`

Setting	Maintenance and SYNC Type	Example
SYNC ON COMMIT to MAINTENANCE AUTO	IDX_SYNC_TYPE: ON COMMIT IDX_MAINTENANCE_TYPE: AUTO	ALTER INDEX CTX_IDX REBUILD PARAMETERS('REPLACE METADATA SYNC (MANUAL) MAINTENANCE AUTO');
SYNC (EVERY) to MAINTENANCE AUTO	IDX_SYNC_TYPE: EVERY IDX_MAINTENANCE_TYPE: AUTO	ALTER INDEX CTX_IDX REBUILD PARAMETERS('REPLACE METADATA SYNC (MANUAL) MAINTENANCE AUTO');

Switching Indexes to Manual Maintenance

This table provides ALTER INDEX examples with various SYNC types, while altering your index from MAINTENANCE AUTO to MAINTENANCE MANUAL:

Setting	Maintenance and SYNC Type	Example
MAINTENANCE AUTO to SYNC (MANUAL)	IDX_SYNC_TYPE: MANUAL IDX_MAINTENANCE_TYPE: MANUAL	ALTER INDEX CTX_IDX REBUILD PARAMETERS('REPLACE METADATA MAINTENANCE MANUAL); The default SYNC type for a CONTEXT index is MANUAL.
MAINTENANCE AUTO to SYNC ON COMMIT	IDX_SYNC_TYPE: ON COMMIT IDX_MAINTENANCE_TYPE: MANUAL	ALTER INDEX CTX_IDX REBUILD PARAMETERS('REPLACE METADATA SYNC(ON COMMIT) MAINTENANCE MANUAL');
MAINTENANCE AUTO to SYNC (EVERY)	IDX_SYNC_TYPE: EVERY IDX_MAINTENANCE_TYPE: MANUAL	ALTER INDEX CTX_IDX REBUILD PARAMETERS('REPLACE METADATA MAINTENANCE MANUAL); ALTER INDEX CTX_IDX REBUILD PARAMETERS('REPLACE METADATA SYNC(EVERY "freq=minutely;interval=20")');

Related Topics

- *Oracle Text Reference*

5.8.6 Monitoring Maintenance Events and Errors

The SYS and CTXSYS users can query catalog views to monitor the status of all background maintenance events for indexes with automatic maintenance.

In an automatic maintenance mode, indexes are asynchronously maintained without any user intervention. Oracle recommends that you periodically examine the CTX and dynamic

performance views to know the status of all background maintenance events that are complete or delayed.

Querying Data and Status about Maintenance Events

Use the following views to monitor events at an index or index partition level:

- `CTX_BACKGROUND_EVENTS`:

  This Oracle Text view displays historical information about the execution of events for the `SYS` or `CTXSYS` user.

- `CTX_USER_BACKGROUND_EVENTS`:

  This Oracle Text view displays historical information about the execution of events for the current user, based on the index owner.

- `V$TEXT_WAITING_EVENTS`:

  This dynamic performance view displays historical information about events that are delayed or cannot complete due to errors or contentions.

For example, you can query the index object number and name, index owner number and name, base table (or table partition) object number and name, type of event (such as `SYNC-Postings`, `SYNC-Mapping`, `SYNC-Ranges`, and so on), status of the event (such as successful, running, failed, and so on), status of the retry iterations (such as retry delays, waiting time, and so on), elapsed time since the event started waiting, logged error messages, and so on.

Handling Errors

When the system encounters an indexing error (such as an index failure, event delay, or event retry), it logs the error in a catalog view. The error is not directly reported to the user. You must periodically query views to examine such errors and take corrective actions, as follows:

- Some errors are transient and do not reproduce on a retry. Such error types do not require user intervention.
- Some failed events may automatically succeed after a retry. If a retry event does not succeed, then try restarting the event from another event.

  For example, when `SYNC-Postings` (Sync-P) fails after a retry, you can restart `SYNC-Scheduler` (Sync-S) so that the system can schedule a serial or concurrent operation. A successful completion of Sync-S clears the queue for Sync-P, and Sync-P immediately runs at a starting level without overloading the system.
- If an event does not succeed even after periodic retries, then contact your database administrator.
- To limit the load on the system due to periodic retries, the delays between successive retries may progressively increase.

  To track such delays, all retry events are scheduled at an incremented level. This means that the Scheduler process first moves a retry event into an Event Queue. From the Event Queue, the Scheduler process moves it into the Waiting Queue, then to the Ready Queue, and finally allocates the event to the Worker process.

  In addition to incremental levels, each event has a retry iteration. On long retries (that is, when a retry event is not the same as the original event), the iteration is incremented instead of the level and the level is initialized to the iteration.
- On Oracle Real Application Clusters (Oracle RAC) systems, there may be errors that occur only on some nodes but not on others. In such cases, an event may successfully complete only when the event is sent to the other nodes.

Related Topics

- CTX_BACKGROUND_EVENTS
- CTX_USER_BACKGROUND_EVENTS
- V$TEXT_WAITING_EVENTS

6
Querying with Oracle Text

Become familiar with Oracle Text querying and associated features.

This chapter contains the following topics:

6.1 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. You can use the scores 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 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.

6.1.1 Querying with CONTAINS

When you create an index of type `CONTEXT`, you must use the `CONTAINS` operator to enter your query. This index is suitable 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.

6.1.1.1 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 by 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 shown in the previous example.

6.1.1.2 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 shown as output by using the cursor dot notation.

6.1.1.3 Structured Query with CONTAINS Example

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

To enter a structured query, 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* 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;
```

6.1.2 Querying with CATSEARCH

When you create an index of type CTXCAT, you must use the CATSEARCH operator to enter your query.

This index is suitable when your application stores short text fragments in the text column and associated information in related columns.

For example, an application serving an online auction site includes a table that stores item descriptions in a text column and date and price information in other columns. With a CTXCAT index, you can create b-tree indexes on one or more columns, so that query performance is generally faster for mixed queries.

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

- CATSEARCH SQL Query
- CATSEARCH Example

> **Note:**
>
> The Oracle Text indextype CTXCAT is deprecated with Oracle Database 23ai. The indextype itself, and it's operator CTXCAT, can be removed in a future release. Both CTXCAT and the use of CTXCAT grammar as an alternative grammar for CONTEXT queries is deprecated. Instead, Oracle recommends that you use the CONTEXT indextype, which can provide all the same functionality, except that it is not transactional. Near-transactional behavior in CONTEXT can be achieved by using SYNC(ON COMMIT) or, preferably, SYNC(EVERY [*time-period*]) with a short time period.
>
> CTXCAT was introduced when indexes were typically a few megabytes in size. Modern, large indexes, can be difficult to manage with CTXCAT. The addition of index sets to CTXCAT can be achieved more effectively by the use of FILTER BY and ORDER BY columns, or SDATA, or both, in the CONTEXT indextype. CTXCAT is therefore rarely an appropriate choice. Oracle recommends that you choose the more efficient CONTEXT indextype.

6.1.2.1 CATSEARCH SQL Query Example

A typical query with CATSEARCH includes the following structured clause 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 tht you can enter depends on how you create your sub-indexes.

> **See Also:**
>
> "Creating a CTXCAT Index"

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

For example, assuming that `category_id` and `bid_close` have a subindex 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;
```

6.1.2.2 CATSEARCH Example

The following example shows a field section search against a `CTXCAT` index. It uses `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('BOOK_INDEX_SET','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;
```

6.1.3 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. You can 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

- MATCHES SQL Query
- MATCHES PL/SQL Example

> **See Also:**
>
> Classifying Documents in Oracle Text

6.1.3.1 MATCHES SQL Query

A `MATCHES` query finds all rows in a query table that match a given document. Assuming that a `querytable` table is associated with a `CTXRULE` index, enter the following query:

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

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

Here is 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 returns 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).

In this example, the document was passed in as a string for simplicity. Your document is typically 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` by using `AUTO_FILTER`. The following example makes two assumptions:

- The document data is in the `:doc_blob` bind variable.
- You have already defined `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 `CTX_DOC.POLICY_FILTER` procedure filters the `BLOB` into the `CLOB` data, because you must get the text into a `CLOB` to enter a `MATCHES` query. It takes, as one argument, the name of a policy that you already created with `CTX_DDL.CREATE_POLICY`.

> **See Also:**
>
> *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` by using the `DBMS_LOB.LOADFROMFILE` function, 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:**
>
> Classifying Documents in Oracle Text for more extended classification examples

6.1.3.2 MATCHES PL/SQL Examples

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

This example loops through the newsfeed table, categorizing each article by 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:**
>
> Classifying Documents in Oracle Text for more extended classification examples

6.1.4 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. 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, you insert the `AND` operator between words in phrases. For example, a query such as international law is interpreted as *international AND law.*

6.1.5 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.

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 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.

Because the Oracle Text index records the position of stopwords even though it does not create an index entry for them, you can query phrases that contain stopwords as well as indexable words, such as *this boy talks to that girl*.

When you include a stopword within your query phrase, the stopword matches any word. For example, the following query assumes that *was* is a stopword. It matches phrases such as *Jack is big* and *Jack grew big.* It also matches *grew,* even though it is not a stopword.

```
'Jack was big'
```

Starting with Oracle Database 12*c* Release 2 (12.2), stopwords and unary operators on stopwords are ignored at the initial stages of a query result in different query results than earlier releases. For example, the following query does not return documents because `the` is a stopword and the `$` operator and the stopword are ignored during query processing:

```
SQL> select count(1) from tabx where contains(text,'$the')>0;
 .
  COUNT(1)
 ----------
         0
```

The next query returns documents containing `first` because the `the` stopword and the `$` operator are ignored.

```
SQL> select count(1) from tabx where contains(text,'first and $the')>0;
 .
  COUNT(1)
 ----------
         2
```

6.1.6 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 that 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_DDL` package. You can add stopthemes after indexing with the `ALTER INDEX` statement.

6.2 Oracle Text Query Features

Oracle Text has various query features. You can use these query features in your query application.

- Query Expressions
- Case-Sensitive Searching
- Query Feedback
- Query Explain Plan
- Using a Thesaurus in Queries
- About Document Section Searching
- Using Query Templates
- Query Analysis
- Other Query Features

6.2.1 Query Expressions

A query expression is everything in between the single quotes in the `text_query` argument of the `CONTAINS` or `CATSEARCH` operator. The contents of a query expression in a `CONTAINS` query differs from the contents of a `CATSEARCH` operator.

- CONTAINS Operators
- CATSEARCH Operator
- MATCHES Operator

6.2.1.1 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"

6.2.1.2 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 the `ORDER BY` clause to order your output.

> **See Also:**
>
> "The CTXCAT Grammar"

6.2.1.3 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"

6.2.2 Case-Sensitive Searching

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

Word queries are *not* 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*, but not *Dog* or *DOG.*

You can enable or disable case-sensitive searching with the `MIXED_CASE` attribute in your `BASIC_LEXER` index preference. With a case-sensitive index, your queries must be entered in exact case. For example, a query on *Dog* matches only documents with *Dog.* Documents with *dog* or *DOG* are not returned as hits.

To enable case-insensitive searching, set the `MIXED_CASE` attribute in your `BASIC_LEXER` index preference to `NO`.

> **Note:**
>
> If you enable case-sensitivity for word queries and you query a phrase containing stopwords and indexable words, 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 give the best results when your query is formulated with proper case because the normalization of your query is based on the knowledge catalog. The knowledge catalog 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 do not have to 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 lookup.

6.2.3 Query Feedback

Feedback 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 returned feedback information is obtained from the knowledge base and contains only those terms that are also in the index. This process 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`

6.2.4 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
- Stopword transformations
- Breakdown of composite-word tokens for supported languages

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

6.2.5 Using a Thesaurus in Queries

Oracle Text enables you to define a thesaurus for your query application and process queries more intelligently.

Because users 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:**
>
> Working With a Thesaurus in Oracle Text

6.2.6 Document Section Searching

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

You can implement section searching 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 by 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:**
>
> Searching Document Sections in Oracle Text

6.2.7 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
- Ordering By SDATA Sections
- Alternative and User-defined Scoring
- Alternative Grammar

> **Note:**
>
> The Oracle Text indextype `CTXCAT` is deprecated with Oracle Database 23ai. The indextype itself, and it's operator `CTXCAT`, can be removed in a future release. Both `CTXCAT` and the use of `CTXCAT` grammar as an alternative grammar for `CONTEXT` queries is deprecated. Instead, Oracle recommends that you use the `CONTEXT` indextype, which can provide all the same functionality, except that it is not transactional. Near-transactional behavior in `CONTEXT` can be achieved by using `SYNC(ON COMMIT)` or, preferably, `SYNC(EVERY [time-period])` with a short time period.
>
> `CTXCAT` was introduced when indexes were typically a few megabytes in size. Modern, large indexes, can be difficult to manage with `CTXCAT`. The addition of index sets to `CTXCAT` can be achieved more effectively by the use of `FILTER BY` and `ORDER BY` columns, or `SDATA`, or both, in the `CONTEXT` indextype. `CTXCAT` is therefore rarely an appropriate choice. Oracle recommends that you choose the more efficient `CONTEXT` indextype.

6.2.7.1 Query Rewrite

Query applications sometimes parse end-user queries, interpreting a query string in one or more ways by using different operator combinations. For example, if a user enters a query of *kukui nut,* your application enters the *{kukui nut}* and *{kukui or nut}* queries 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 following is the query rewrite template for these transformations:

```
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;
```

6.2.7.2 Query Relaxation

The query relaxation feature enables your application to execute the most restrictive version of a query first and progressively relax the query until the required number of hits are obtained.

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

The following query template defines a query relaxation sequence. The query of *green pen* is entered in sequence.

{green} {pen}

{green} NEAR {pen}

{green} AND {pen}

{green} ACCUM {pen}

The following is the query relaxation template for these transformations:

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

```
    <progression>
      <seq>{green} {pen}</seq>
      <seq>{green} NEAR {pen}</seq>
      <seq>{green} AND {pen}</seq>
      <seq>{green} 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 in a PL/SQL cursor.

Using query templating to relax a query is more efficient than reexecuting a query.

6.2.7.3 Query Language

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

```
select id from docs where CONTAINS (text,
'<query><textquery lang="french">bon soir</textquery></query>')>0;
```

> **See Also:**
>
> *Oracle Text Reference* for information on `LANGUAGE` and `lang` with ALTER INDEX and document sublexer

6.2.7.4 Ordering by SDATA Sections

You can order the query results according to the content of `SDATA` sections by using the `<order>` and `<orderkey>` elements of the query template.

In the following example, the first level of ordering is performed on the `SDATA price` section, which is sorted in ascending order. The second and third level of ordering are performed by the `SDATA pub_date` section and score, both of which are sorted in descending order.

```
select id from docs where CONTAINS (text, '
<query>
  <textquery lang="ENGLISH" grammar="CONTEXT"> Oracle </textquery>
  <score datatype="INTEGER" algorithm="COUNT"/>
  <order>
      <orderkey> SDATA(price) ASC </orderkey>
      <orderkey> SDATA(pub_date) DESC </orderKey>
      <orderkey> Score DESC </orderkey>
  </order>
</query>', 1)>0;
```

> **Note:**
>
> - You can add additional SDATA sections to an index. Refer to the ADD SDATA SECTION parameter string under ALTER INDEX in *Oracle Text Reference*.
> - Documents that were indexed before adding an SDATA section do not reflect this new preference. Rebuild the index in this case.

> **See Also:**
>
> *Oracle Text Reference* for syntax of <order> and <orderkey> elements of the query template

6.2.7.5 Alternative and User-Defined Scoring

You can use query templating to specify alternative scoring algorithms. Those algorithms help you customize how CONTAINS is scored. They also enable SDATA to be used as part of the scoring expressions. In this way, you can mathematically define the scoring expression by using not only predefined scoring components, but also SDATA components.

With alternative user-defined scoring, you can specify:

- 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
- Scoring expressions at the term level
- Terms that should not be taken into account when calculating the score
- How the score from child elements of OR and AND operators should be merged
- Use

You can also use the 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"

6.2.7.6 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;
```

6.2.8 Query Analysis

Oracle Text enables you to create a log of queries and to analyze the queries. For example, suppose you have an application that searches a database of large animals, and your analysis of its queries shows that users search for the word *mouse.* This analysis shows you that you should rewrite your application to avoid returning an unsuccessful search. Instead, a search for *mouse* redirects users to a database of small animals.

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 contains 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.

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

6.2.9 Other Query Features

In your query application, you can use other query features such as proximity searching. Table 6-1 lists some of these features.

Table 6-1 Other Oracle Text Query Features

Feature	Description	Implement With
Case-Sensitive Searching	Enables you to search on words or phrases exactly as they are 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 the specified term as subcomposite.	`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
Expanded operator containing the functionality of `PHRASE`, `NEAR` and `AND` operators.	Breaks a document into clumps based on the given query. Each clump is classified based on a primary feature, and is scored based on secondary features. The final document score adds clump scores such that the ordering of primary features determines the initial ordering of document scores.	`NEAR2` operator when you enter the query
Stemming	Searches for words with the same root as the specified term.	$ operator at when you enter the query
Fuzzy Searching	Searches for words that have a similar spelling as the 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

Table 6-1 (Cont.) Other Oracle Text Query Features

Feature	Description	Implement With
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 the thesaurus.)

7

Working with CONTEXT and CTXCAT Grammars in Oracle Text

Become familiar with CONTEXT and CTXCAT grammars.

This chapter contains the following topics:

- The CONTEXT Grammar
- The CTXCAT Grammar

> **Note:**
>
> The Oracle Text indextype `CTXCAT` is deprecated with Oracle Database 23ai. The indextype itself, and it's operator `CTXCAT`, can be removed in a future release. Both `CTXCAT` and the use of `CTXCAT` grammar as an alternative grammar for `CONTEXT` queries is deprecated. Instead, Oracle recommends that you use the `CONTEXT` indextype, which can provide all the same functionality, except that it is not transactional. Near-transactional behavior in `CONTEXT` can be achieved by using `SYNC(ON COMMIT)` or, preferably, `SYNC(EVERY [time-period])` with a short time period.
>
> `CTXCAT` was introduced when indexes were typically a few megabytes in size. Modern, large indexes, can be difficult to manage with `CTXCAT`. The addition of index sets to `CTXCAT` can be achieved more effectively by the use of `FILTER BY` and `ORDER BY` columns, or `SDATA`, or both, in the `CONTEXT` indextype. `CTXCAT` is therefore rarely an appropriate choice. Oracle recommends that you choose the more efficient `CONTEXT` indextype.

7.1 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 `AND` logical operator 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 searches; the `NEAR` operator for proximity searches; and the stem, fuzzy, and thesaurus 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 and HTML and XML

- Proximity Queries with NEAR, NEAR_ACCUM, and NEAR2 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

7.1.1 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 in your index, you must include that is created by default in English and French.

Enter a theme query by 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

7.1.2 Logical Operators

Use logical operators to limit your search criteria in a number of ways. Table 7-1 describes some of these operators.

Table 7-1 Logical Operators

Operator	Symbol	Description	Example Expression
AND	&	Use to search for documents that contain at least one occurrence of *each* of the query terms. The returned score is the minimum of the operands.	`'cats AND dogs'` `'cats & dogs'`
OR	|	Use to search for documents that contain at least one occurrence of *any* of the query terms. The returned score is the maximum of the operands.	`'cats | dogs'` `'cats OR dogs'`
NOT	~	Use 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 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 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'`

7.1.3 Section Searching and HTML and XML

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

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

> **See Also:**
>
> Searching Document Sections in Oracle Text

7.1.4 Proximity Queries with NEAR, and NEAR2 Operators

Use the `NEAR` operator to search for terms that are near to one another in a document.

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

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

The `NEAR` operator is now modified to change how the distance is measured between phrases in `NESTED NEAR`.

The `NEAR2` operator combines the functionality of `PHRASE`, `NEAR`, and `AND` operators. In addition, the `NEAR2` operator can use position information to boost the scores of its hits. That is, if one phrase hit occurs at the beginning of a document and another at the end of the document, then a higher weight is given to the first hit as compared to the second hit.

> **See Also:**
>
> *Oracle Text Reference* for more information about using the `NEAR`, and `NEAR2` operators

7.1.5 Fuzzy, Stem, Soundex, Wildcard and Thesaurus Expansion Operators

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?"

7.1.6 Using CTXCAT Grammar

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

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

7.1.7 Defined Stored Query Expressions

Use the `CTX_QUERY.STORE_SQE` procedure 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, you can use the stored query expressions to define 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.

Related Topics

- Defining a Stored Query Expression
- SQE Example
- *Oracle Text Reference*

7.1.7.1 Defining a Stored Query Expression

To define and use a stored query expression:

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. Use the `SQE` operator to call the stored query expression in a query expression. Oracle Text returns the results of the stored query expression in the same way that it returns the results of a regular query. The query is evaluated when the stored query expression is called.

   You can delete a stored query expression by using `REMOVE_SQE`.

7.1.7.2 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`

7.1.8 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, if the `french` function 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

7.1.9 Optimizing for Response Time

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

The following example returns the first twenty hits as output. 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;
/
```

The following factors can also influence query response time:

- Collection of table statistics
- Memory allocation
- Sorting
- Presence of large object columns in your base table
- Partitioning
- Parallelism
- Number of term expansions in your query

> **See Also:**
>
> "Frequently Asked Questions About Query Performance"

7.1.10 Counting Hits

Use `CTX_QUERY.COUNT_HITS` in PL/SQL or `COUNT(*)` in a SQL `SELECT` statement to count the number of hits returned from a query with only a `CONTAINS` predicate.

If you want a rough hit count, 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.

Use the `COUNT(*)` function in a `SELECT` statement to count the number of hits returned from a query that contains a structured predicate.

To find the number of documents that contain the word *oracle*, enter the query with the `SQL COUNT` function.

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

To find the number of documents returned by a query with a structured predicate, use `COUNT(*)`.

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

To find the number of documents that contain the word *oracle*, use `COUNT_HITS`.

```
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`

7.1.11 Using DEFINESCORE and DEFINEMERGE for User-Defined Scoring

Use the `DEFINESCORE` operator 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" for information about the alternative scoring template
- *Oracle Text Reference* to learn more about the syntax of `DEFINESCORE` and `DEFINEMERGE`

7.2 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.

Note:

The Oracle Text indextype `CTXCAT` is deprecated with Oracle Database 23ai. The indextype itself, and it's operator `CTXCAT`, can be removed in a future release. Both `CTXCAT` and the use of `CTXCAT` grammar as an alternative grammar for `CONTEXT` queries is deprecated. Instead, Oracle recommends that you use the `CONTEXT` indextype, which can provide all the same functionality, except that it is not transactional. Near-transactional behavior in `CONTEXT` can be achieved by using `SYNC(ON COMMIT)` or, preferably, `SYNC(EVERY [time-period])` with a short time period.

`CTXCAT` was introduced when indexes were typically a few megabytes in size. Modern, large indexes, can be difficult to manage with `CTXCAT`. The addition of index sets to `CTXCAT` can be achieved more effectively by the use of `FILTER BY` and `ORDER BY` columns, or `SDATA`, or both, in the `CONTEXT` indextype. `CTXCAT` is therefore rarely an appropriate choice. Oracle recommends that you choose the more efficient `CONTEXT` indextype.

The `CATSEARCH` query operators have the following syntax:

Table 7-2 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 you define the hyphen as a skipjoin, then words such as *vice-president* are treated as the single query term *vicepresident.* Likewise, if you define the hyphen as a printjoin, then words such as *vice-president* are treated as *vice president* with the space in the `CTXCAT` query language.

Table 7-2 (Cont.) CATSEARCH Query Operator Syntax

Operation	Syntax	Description of Operation
" "	"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.

To use the `CONTEXT` grammar in `CATSEARCH` queries, 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.

Related Topics

- *Oracle Text Reference*

8

Presenting Documents in Oracle Text

Oracle Text provides various methods for presenting documents in results for query applications.

This chapter contains the following topics:

8.1 Highlighting Query Terms

In 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

This section contains the following topics:

8.1.1 Text highlighting

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

8.1.2 Theme Highlighting

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

8.1.3 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.

> **Note:**
>
> `SNIPPET` can also be generated using the Result Set Interface.

> **See Also:**
>
> *Oracle Text Reference* for information on `CTX_QUERY.RESULT_SET`

This section contains these topics:

- Markup Procedure
- Highlight Procedure
- Concordance

8.1.3.1 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 plain text or marked-up HTML. For example, 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 CONTEXT Query Application. The `showDoc` procedure takes an HTML document and a query, creates the highlight markup—in this case, the query term is displayed 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`

8.1.3.2 Highlight Procedure

`CTX_DOC.HIGHLIGHT` and `CTX_DOC.POLICY_HIGHLIGHT` take a query and a document and return offset information for the query in plain text or HTML format. You can use this offset information 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 a document (such as different font types or colors) instead of 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`

8.1.3.3 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 (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 how 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. `CTX_DOC.POLICY_SNIPPET` and `CTX_DOC.SNIPPET` include two new attributes: `radius` specifies the approximate desired length of each segment, whereas, `max_length` puts an upper bound on the length of the sum of all segments.

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

8.2 Obtaining Part-of-Speech Information for a Document

The `CTX_DOC` package contains procedures to create policies for obtaining part-of-speech information for a given document. This approach is described under `POLICY_NOUN_PHRASES` in *Oracle Text Reference* and `POLICY_PART_OF_SPEECH` in *Oracle Text Reference*.

8.3 Obtaining Lists of Themes, Gists, and Theme Summaries

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

Table 8-1 Lists of Themes, Gists, and Theme Summaries

Output Type	Description
List of Themes	A list of the main concepts of a document. Each theme is a single word, a single phrase, or 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 lists of themes, gists, and theme summaries, use procedures in the `CTX_DOC` package to:

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

8.3.1 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`

The following in-memory theme 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;
```

The following example create a result table theme:

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

In this example, you obtain a list of themes where each element in the list is a single theme:

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

In this example, you obtain a list of themes where each element in the list is a hierarchical list of parent themes:

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

8.3.2 Gist and Theme Summary

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

Use the `CTX_DOC.GIST` procedure 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 Example

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 Example

To create a gist table, enter the following:

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

The following example returns a default-sized paragraph 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 10 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 10 percent of the total paragraphs in the document:

```
begin
ctx_doc.gist('newsindex','34','CTX_GIST',1, 'PARAGRAPH', pov =>'GENERIC', maxPercent =>
10);
end;
```

Theme Summary Example

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;
```

8.4 Presenting and Highlighting Documents

Typically, a query application enables the user to view the documents returned by a query. The user selects a document from the hitlist, and then the application presents the document in some form.

With Oracle Text, you can display a document in different ways, such as highlighting 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 8-2 describes the different output you can obtain and which procedure to use to obtain each type.

Table 8-2 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`
Highlighted offset information for plain-text version	`CTX_DOC.HIGHLIGHT`
Highlighted 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

9

Classifying Documents in Oracle Text

Oracle Text offers various approaches to document classification.

This chapter contains the following topics:

9.1 Overview of Document Classification

Each theme is a single word, a single phrase, or a hierarchical list of parent themes.

To sift through numerous documents you can 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 about the specific 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.

Alternatively, you can sort through documents and classify them by content. This approach is not feasible for very large volumes of documents.

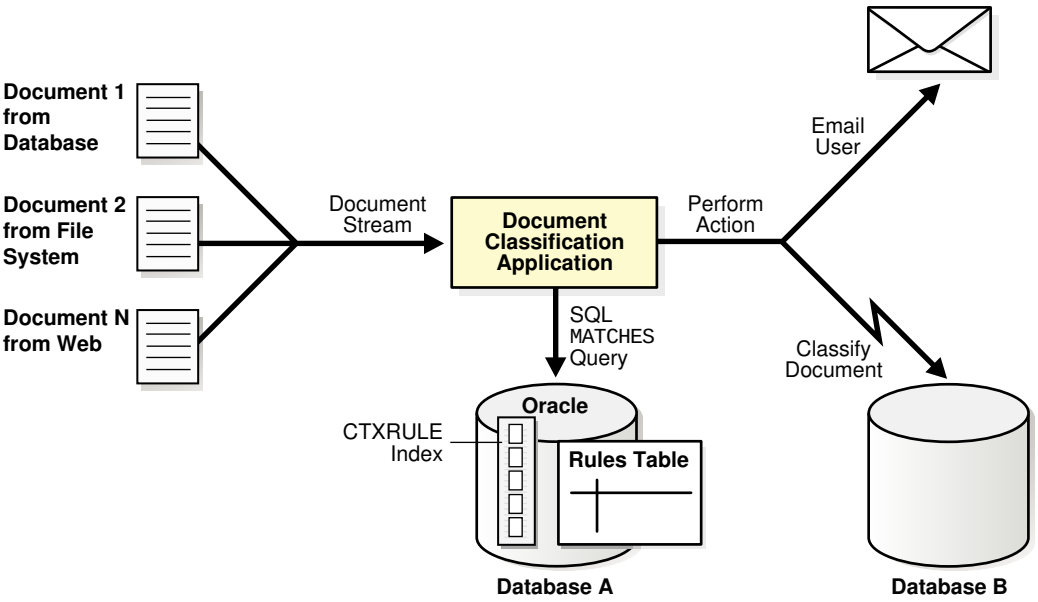
Oracle Text offers various approaches to document classification. Under *rule-based classification* (sometimes called *simple classification*), you write the classification rules yourself. With *supervised classification*, Oracle Text creates classification rules based on a set of sample documents that you preclassify. Finally, with *unsupervised classification* (also known as *clustering*), Oracle Text performs all steps, from writing the classification rules to classifying the documents, for you.

9.2 Classification Applications

Oracle Text enables you to build document classification applications that perform 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 9-1 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 9-1 Overview of a Document Classification Application



9.3 Classification Solutions

Oracle Text enables you to classify documents in the following ways:

- Rule-Based Classification. For this solution, you group your documents, choose 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.

  Advantages: This solution 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 solution is similar to rule-based classification, but the rule-writing step is automated with `CTX_CLS.TRAIN`. This procedure formulates a set of classification rules from a sample set of preclassified documents that you provide. As with rule-based classification, you use the `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 preferences are discussed in "Supervised Classification".

  Advantages: Rules are written for you automatically. This method 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 the classification rules or the sample documents as a training set.
- This solution helps to discover overlooked patterns and content similarities in your document set.

  In fact, you can use this solution when you do not have a clear idea of rules or classifications. For example, use it to provide an initial set of categories and to build on the categories through supervised classification.

Disadvantages:

- Clustering is based on an internal solution. It might result in unexpected groupings, because the clustering operation is not user-defined.
- 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.

9.4 Rule-Based Classification

Rule-based classification is the basic solution for 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 then 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 define the "medicine" category as documents that include the words "hospital," "doctor," or "disease." Therefore, you would set up a rule in the form of "hospital OR doctor OR disease."
3. Create a `CTXRULE` index on the rule table.
4. Classify the documents.

> **See Also:**
>
> "CTXRULE Parameters and Limitations" for information on which operators are allowed for queries

9.4.1 Rule-Based Classification Example

In this example, you gather news articles about different subjects and then classify them. After you create the rules, you can index them and then use the `MATCHES` statement to classify documents.

To classify documents:

1. Create the schema to store the data.

The `news_table` stores the documents to be classified. The `news_categories` table stores the categories and rules that define the 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);
```

2. Load the documents with SQLLDR.

   Use the `SQLLDR` program to load the HTML news articles into the `news_table`. The file names 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)
```

3. Create the categories and write the rules for each category.

   The defined categories are Asia, Europe, Africa, Middle East, Latin America, United States, Conflicts, Finance, Technology, Consumer Electronics, World Politics, U.S. Politics, Astronomy, Paleontology, Health, Natural Disasters, Law, and Music News.

   A rule is a query that selects documents for the category. For example, the 'Asia' category has a rule of 'China or Pakistan or India or Japan'. Insert the rules in the `news_categories` table.

```
insert into news_categories values
  (1,'United States','Washington or George Bush or Colin Powell');

insert into news_categories values
  (2,'Europe','England or Britain or Germany');

insert into news_categories values
  (3,'Middle East','Israel or Iran or Palestine');

insert into news_categories values(4,'Asia','China or Pakistan or India or Japan');

insert into news_categories values(5,'Africa','Egypt or Kenya or Nigeria');

insert into news_categories values
  (6,'Conflicts','war or soldiers 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');

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 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','Supreme Court 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;
```

4. Create the `CTXRULE` index on the `news_categories` query column.

   ```
   create index news_cat_idx on news_categories(query)
   indextype is ctxsys.ctxrule;
   ```

5. To classify the documents, use the `CLASSIFIER.THIS` PL/SQL procedure (a simple procedure designed for this example).

   The procedure 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
```

9.4.2 CTXRULE Parameters and Limitations

The following considerations apply to indexing a `CTXRULE` index:

- If you use the `SVM_CLASSIFIER` classifier, then you may use the `BASIC_LEXER`, `CHINESE_LEXER`, `JAPANESE_LEXER`, or `KOREAN_MORPH_LEXER` lexers. If you do not use `SVM_CLASSIFIER`, then you can use only the `BASIC_LEXER` lexer type to index your query set.
- Filter, memory, datastore, and [no]populate parameters are not applicable to the `CTXRULE` index type.
- 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 the `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.
- You must drop the `CTXRULE` index before exporting or downgrading the database.

> **See Also:**
>
> - *Oracle Text Reference* for more information on lexer and classifier preferences
> - "Creating a CTXRULE Index"

9.5 Supervised Classification

With supervised classification, you use 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 training set is the major advantage over rule-based classification, where 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 Also:**
>
> *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 can select one of the following classification algorithms for supervised classification:

- Decision Tree Supervised Classification

  The advantage of this classification is that the generated rules are easily observed (and modified).
- SVM-Based Supervised Classification

  This classification uses the Support Vector Machine (SVM) algorithm for creating rules. The advantage of this classification is that it is often more accurate than the Decision Tree classification. The disadvantage is that it generates binary rules, so the rules themselves are opaque.

> **See Also:**
>
> - "Decision Tree Supervised Classification Example"
> - "SVM-Based Supervised Classification Example"

9.5.1 Decision Tree Supervised Classification

To use Decision Tree classification, you set the preference argument of `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, the attributes include:

- words from the document
- stems of words from the document (for 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 that are suitable for use by a `CTXRULE` index. For example, one category has a training document for "Japanese beetle," and another category has a document for "Japanese currency." The algorithm may create decision trees based on "Japanese," "beetle," and "currency," and then classify documents accordingly.

The decision trees include the concept of *confidence.* Each generated rule is allocated a percentage value that represents the accuracy of the rule, given the current training set. In trivial examples, the accuracy is almost always 100 percent, but this percentage 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 they sufficiently distinguish the different categories in the current training set.

The advantage of the Decision Tree classification is that it can generate rules that users can easily inspect and modify. The 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.

9.5.2 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 `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.

After you generate the rules, you can test them by first indexing them and then using `MATCHES` to classify new documents.

To create and index the category rules:

1. Create and load a table of training documents.

   This example uses a simple set of three fast food documents and three computer documents.

```
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');
```

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;

-- descriptions for categories

insert into category_descriptions values (1, 'fast food');
insert into category_descriptions values (2, 'computer networking');
```

3. Assign each document to a category.

   In this case, the fast food documents all go into category 1, and the computer documents go 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);
```

4. Create a `CONTEXT` index to be used by `CTX_CLS.TRAIN`.

   To experiment with the effects of turning themes on and off, create an Oracle Text preference for the index.

```
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');
```

5. Create the rules table that will be populated by the generated rules.

```
create table rules(
  rule_cat_id     number,
  rule_text       varchar2(4000),
  rule_confidence number
);
```

6. Generate category rules.

   All 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;
/
```

7. Fetch the generated rules, viewed by category.

   For convenience's sake, the `rules` table is joined with `category_descriptions` so that you can see the category that each rule applies to.

   ```
   select cd_description, rule_confidence, rule_text from rules,
   category_descriptions where cd_category = rule_cat_id;
   ```

8. Use the `CREATE INDEX` statement to create the `CTXRULE` index on the previously generated rules.

   ```
   create index rules_idx on rules (rule_text) indextype is ctxsys.ctxrule;
   ```

9. Test an incoming document by 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;
   /
   ```

9.5.3 SVM-Based Supervised Classification

The second method that you can use for training purposes is 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, except for the following differences:

- In the call to `CTX_CLS.TRAIN`, use the `SVM_CLASSIFIER` preference instead of the `RULE_CLASSIFIER` preference. (If you do not want to modify any attributes, use the predefined `CTXSYS.SVM_CLASSIFIER` preference.)
- Use the `NOPOPULATE` keyword if you do not want to populate the `CONTEXT` index on the table. 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.

- In the generated rules table, use at least the following 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 incurs 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, increase one of the following memories:

- SGA (if in shared server mode)
- PGA (if in dedicated server mode)

9.5.4 SVM-Based Supervised Classification Example

This example uses SVM-based classification. The steps are essentially the same as the Decision Tree example, except for the following differences:

- Set the `SVM_CLASSIFIER` preference with `CTX_DDL.CREATE_PREFERENCE` rather than setting it in `CTX_CLS.TRAIN`. (You can do it either way.)
- Include category descriptions in the category table. (You can do it either way.)
- Because rules are opaque to the user, use fewer arguments in `CTX_CLS.TRAIN`.

To create a SVM-based supervised classification:

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');
```

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');
```

3. Create the `CONTEXT` index on the document table without populating it.

```
create index docx on doc(text) indextype is ctxsys.context
       parameters('nopopulate');
```

4. Set the `SVM_CLASSIFIER`.

   You can also set it in `CTX.CLS_TRAIN`.

```
exec ctx_ddl.create_preference('my_classifier','SVM_CLASSIFIER');
exec ctx_ddl.set_attribute('my_classifier','MAX_FEATURES','100');
```

5. Create the result (rule) table.

```
create table restab (
  cat_id number,
  type number(3) not null,
  rule blob
 );
```

6. Perform the training.

```
exec ctx_cls.train('docx', 'id','testcategory','doc_id','cat_id',
     'restab','my_classifier');
```

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 you can classify two unknown documents, as follows:

```
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');
```

9.6 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 preclassify. With *unsupervised classification* (also known as *clustering*), you do not 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 about animals. `CTX_CLS.CLUSTERING` creates one leaf cluster about dogs, another about cats, another about fish, and a fourth about bears. (The first three might be grouped under a node cluster about pets.) Suppose further that you have a document about one breed of dogs, such as Chihuahuas. `CTX_CLS.CLUSTERING` assigns the dog cluster to the document with a very high relevancy score, whereas the cat cluster is assigned a lower score and the fish and bear clusters are still assigned lower scores. After scores for all clusters are assigned to all documents, an application can then take action based on the scores.

As noted in "Decision Tree Supervised Classification", 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 the document's similarity 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 a generated cluster. This table contains cluster identification, cluster description text, a suggested cluster label, and a quality score for the cluster.

`CTX_CLS.CLUSTERING` uses 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 information on cluster types and hierarchical clustering

9.7 Unsupervised Classification (Clustering) Example

This 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 is then 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 Oracle 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),
       size   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

10

Tuning Oracle Text

Oracle Text provides ways to improve your query and indexing performance.

This chapter contains the following topics:

10.1 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. Optimizing with statistics enables a more accurate estimation of the selectivity and costs of the `CONTAINS` predicate and thus a better execution plan.

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

The following topics discuss how to use statistics with the extensible query optimizer:

> **Note:**
>
> Importing and exporting statistics on domain indexes, including Oracle Text indexes, is not supported with the `DBMS_STATS` package. For more information on importing and exporting statistics, see the *Oracle Database PL/SQL Packages and Types Reference*.

> **See Also:**
>
> *Oracle Text Reference* for information on the `CONTAINS` query operator

10.1.1 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, calculate the statistics on the table you queried table:

```
ANALYZE TABLE <table_name> COMPUTE STATISTICS;
```

Alternatively, estimate the statistics on a sample of the table:

```
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 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 CBO in Oracle Database can perform the following tasks:

- Estimate the selectivity of the `CONTAINS` predicate
- Estimate the I/O and CPU costs of using the Oracle Text index (that is, the cost of processing the `CONTAINS` predicate by 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 CBO can better decide whether to use the domain index to evaluate CONTAINS or to apply the CONTAINS predicate as a post filter.

> **See Also:**
>
> - *Oracle Database SQL Language Reference* for more information about the ANALYZE statement
> - *Oracle Database PL/SQL Packages and Types Reference* for information about DBMS_STATS package

10.1.2 Query Optimization with Statistics Example

The following structured query provides an example for optimizing statistics:

```
select score(1) from tab where contains(txt, 'freedom', 1)  > 0 and author = 'King' and
year > 1960;
```

Assume the following:

- The author column is of type VARCHAR2 and the year column is of type NUMBER.
- A b-tree index on the author column.
- 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 scanning a b-tree index range on the structured predicate (author = 'King'), then accessing a table 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 Oracle Text index, the CBO assumes low selectivity and index costs for the CONTAINS predicate.

10.1.3 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 was reanalyzed before the synchronization, it is sufficient to analyze the index after the synchronization without reanalyzing the entire table.

To re-collect statistics, enter one of the following statements:

```
ANALYZE INDEX <index_name> COMPUTE STATISTICS;

ANALYZE INDEX <index_name> ESTIMATE STATISTICS SAMPLE 50 PERCENT;
```

10.1.4 Deleting Statistics

Delete the statistics associated with a table:

```
ANALYZE TABLE <table_name> DELETE STATISTICS;
```

Delete statistics on one index:

```
ANALYZE INDEX <index_name> DELETE STATISTICS;
```

10.2 Optimizing Queries for Response Time

By default, Oracle Text optimizes queries for throughput so that queries return all rows in the shortest time possible.

However, in many cases, especially in a web application, you must optimize queries for response time, because 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:

- Other Factors that Influence Query Response Time
- Improved Response Time with FIRST_ROWS(n) Hint for ORDER BY Queries
- Improved Response Time Using the DOMAIN_INDEX_SORT Hint
- Improved Response Time using Local Partitioned CONTEXT Index
- Improved Response Time with Local Partitioned Index for Order by Score
- Improved Response Time with Query Filter Cache
- Improved Response Time using BIG_IO Option of CONTEXT Index
- Improved Response Time using SEPARATE_OFFSETS Option of CONTEXT Index
- Improved Response Time Using the STAGE_ITAB, STAGE_ITAB_MAX_ROWS, and STAGE_ITAB_PARALLEL Options of CONTEXT Index

10.2.1 Other Factors That Influence Query Response Time

The following factors can influence query response time:

- Collection of table statistics
- Memory allocation
- Sorting
- Presence of large object (LOB) columns in your base table
- Partitioning
- Parallelism
- The number term expansions in your query

> **See Also:**
>
> "Frequently Asked Questions About Query Performance"

10.2.2 Improved Response Time with the 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.

You use the `FIRST_ROWS(n)` hint in cases where you want the first *n* number of rows in the shortest possible time. For example, consider the following PL/SQL block that uses a cursor to retrieve the first 10 hits of a query and 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 `c` cursor 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 `t_s` table.

With the `FIRST_ROWS(n)` hint, the optimizer instructs the Oracle 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 Oracle Text index returns *all* rows in unsorted order that satisfy the `CONTAINS` predicate. Retrieving the entire result set 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.

10.2.3 Improved Response Time Using the DOMAIN_INDEX_SORT Hint

You can also optimize for response time by 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:

```
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 hint might result in suboptimal performance for queries where the `CONTAINS` clause is very selective. In these cases, Oracle recommends that you use the `FIRST_ROWS(n)` hint, which is fully cost-based.

10.2.4 Improved Response Time Using the 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 are combined as necessary to produce the final result set.

Create the `CONTEXT` index with the `LOCAL` keyword:

```
CREATE INDEX index_name ON table_name (column_name)
INDEXTYPE IS ctxsys.context
PARAMETERS ('...')
LOCAL
```

With partitioned tables and indexes, you can improve performance of the following types of queries:

- **Range Search on Partition Key Column:** This query 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 query requires only the first `n` hits, and the `ORDER BY` clause names the partition key. Consider an `ORDER BY` query on a `price` column to fetch the first 20 hits:

  ```
  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.

10.2.5 Improved Response Time with the Local Partitioned Index for Order by Score

The `DOMAIN_INDEX_SORT` hint on a local partitioned index might result in poor performance, especially when you order by score. All hits to the query across all partitions must be obtained before the results can be sorted.

Instead, use an inline view when you use the `DOMAIN_INDEX_SORT` hint. Specifically, use the `DOMAIN_INDEX_SORT` hint to improve query performance on a local partitioned index under the following conditions:

- The Oracle Text query itself, including the order by `SCORE`() clause, is expressed as an in-line view.
- The Oracle Text query inside the in-line view contains the `DOMAIN_INDEX_SORT` hint.
- The query on the in-line view has a `ROWNUM` predicate that limits the number of rows to fetch from the view.

For example, the following Oracle Text query and local Oracle Text index are created on a partitioned `doc_tab` table:

```
select doc_id, score(1) from doc_tab
   where contains(doc, 'oracle', 1)>0
   order by score(1) desc;
```

If you are interested in fetching only the top 20 rows, you can rewrite the query as follows:

```
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 SQL Language Reference* for more information about the `EXPLAIN PLAN` statement

10.2.6 Improved Response Time with the Query Filter Cache

Oracle Text provides a cache layer called the query filter cache that you can use to cache the query results. The query filter cache is sharable across queries. Multiple queries can reuse cached query results to improve the query response time.

Use the `ctxfiltercache` operator to specify which query results to cache. The following example uses the operator to store the results of the `common_predicate` query in the cache:

```
select * from docs where contains(txt, 'ctxfiltercache((common_predicate), FALSE)')>0;
```

In this example, the cached results of the `common_predicate` query are reused by the `new_query` query, to improve the query response time.

```
select * from docs where contains(txt, 'new_query & ctxfiltercache((common_predicate),
FALSE)')>0;
```

Note:

- You can specify the size of the query filter cache by using the basic `query_filter_cache_size` storage attribute.
- The `ctx_filter_cache_statistics` view provides various statistics about the query filter cache.

Note:

The `CTXFILTERCACHE` query operator was designed to speed up commonly-used expressions in queries. In Oracle Database Release 21c, this function is replaced by other internal improvements. The `CTXFILTERCACHE` operator is deprecated (and will pass through its operands to be run as a normal query). Because they no longer have a function, the view `CTX_FILTER_CACHE_STATISTICS` is also deprecated, and also the storage attribute `QUERY_FILTER_CACHE_SIZE`.

See Also:

Oracle Text Reference for more information about:

- `ctxfiltercache` operator
- `query_filter_cache_size` basic storage attribute
- `ctx_filter_cache_statistics` view

10.2.7 Improved Response Time Using the BIG_IO Option of CONTEXT Index

Oracle Text provides the `BIG_IO` option for improving the query performance for the `CONTEXT` indexes that extensively use IO operations.

The query performance improvement is mainly for data stored on rotating disks, not for data stored on solid state disks.

When you enable the `BIG_IO` option, a `CONTEXT` index creates token type pairs with one large object (LOB) data type for each unique token text. Tokens with the same text but different token types correspond to different rows in the `$I` table.

> **Note:**
>
> The BIG_IO attribute of the CONTEXT indextype is deprecated with Oracle Database 23ai, and can be disabled or removed in a future release.
>
> Oracle recommends that you allow this value to be set to its default value of N. BIG_IO was introduced to reduce the cost of seeks when index postings exceeded 4KB in length. However, the internal code is relatively inefficient, and the attribute cannot be combined with newer index options. Seek cost is much less relevant for solid state disks or non-volatile memory devices (NVMe), and seek cost is irrelevant when postings are cached. This setting is therefore of little benefit for most indexes.

The indexes with the BIG_IO option enabled should have the token LOBs created as SecureFile LOBs, so that the data is stored sequentially in multiple blocks. This method improves the response time of the queries, because the queries can now perform longer sequential reads instead of many short reads.

> **Note:**
>
> If you use SecureFiles, you must set the COMPATIBLE setting to 11.0 or higher. In addition, you must create the LOB on an automatic segment space management (ASSM) tablespace. When you migrate the existing Oracle Text indexes to SecureFiles, use an ASSM tablespace. To help migrate the existing indexes to SecureFiles, you can extend ALTER INDEX REBUILD to provide storage preferences that only affect the $I table.

To create a CONTEXT index with the BIG_IO index option, first create a basic storage preference by setting the value of its BIG_IO storage attribute to YES, and then specify this storage preference while creating the CONTEXT index.

The following example creates a basic mystore storage preference and sets the value of its BIG_IO storage attribute to YES:

```
exec ctx_ddl.create_preference('mystore', 'BASIC_STORAGE');
exec ctx_ddl.set_attribute('mystore', 'BIG_IO', 'YES');
```

To disable the BIG_IO option, update the existing storage preference (mystore) by setting the value of its BIG_IO storage attribute to NO, and then rebuild the index.

```
exec ctx_ddl.set_attribute('mystore', 'BIG_IO', 'NO');
alter index idx rebuild('replace storage mystore');
```

> **WARNING:**
>
> Do not use the replace metadata operation to disable the BIG_IO index option. It can leave the index in an inconsistent state.

To enable the BIG_IO option for a partitioned index without rebuilding the index, modify the basic storage preference by setting the value of its BIG_IO storage attribute to YES, replace the

global index metadata using `ctx_ddl.replace_index_metadata`, and then call `optimize_index` in `REBUILD` mode for each partition of the partitioned index table.

The following example enables the `BIG_IO` option for the `idx` partitioned index:

```
exec ctx_ddl.set_attribute('mystore', 'BIG_IO', 'YES');
exec ctx_ddl.replace_index_metadata('idx', 'replace metadata storage mystore');
exec ctx_ddl.optimize_index('idx', 'rebuild', part_name=>'part1');
```

> **Note:**
>
> If a procedure modifies the existing index tables with only the `BIG_IO` option enabled, then it will not result in reindexing of the data.

> **Note:**
>
> Because the `BIG_IO` index option performs longer sequential reads, the queries that use the `BIG_IO` index option require a large program global area (PGA) memory.

10.2.8 Improved Response Time Using the SEPARATE_OFFSETS Option of the CONTEXT Index

Oracle Text provides the `SEPARATE_OFFSETS` option to improve the query performance for the `CONTEXT` indexes that use IO operations, and whose queries are mainly single-word or Boolean queries.

The `SEPARATE_OFFSETS` option creates a different postings list structure for the tokens of type `TEXT`. Instead of interspersing docids, frequencies, info-length (length of the offsets information), and the offsets in the postings list, the `SEPARATE_OFFSETS` option stores all docids and frequencies at the beginning of the postings list, and all info-lengths and offsets at the end of the postings list. The header at the beginning of the posting contains the information about the boundary points between docids and offsets. Because separation of docids and offsets reduces the time for the queries to read the data, it improves the query response time.

To create a `CONTEXT` index with the `SEPARATE_OFFSETS` option, first create a basic storage preference by setting the value of its `SEPARATE_OFFSETS` storage attribute to `T`. Next, specify this storage preference when you create the `CONTEXT` index.

The following example creates a basic `mystore` storage preference and sets the value of its `SEPARATE_OFFSETS` storage attribute to `T`:

```
exec ctx_ddl.create_preference('mystore', 'BASIC_STORAGE');
exec ctx_ddl.set_attribute('mystore', 'SEPARATE_OFFSETS', 'T');
```

To disable the `SEPARATE_OFFSETS` option, update the existing storage preference (`mystore`) by setting the value of its `SEPARATE_OFFSETS` storage attribute to `F`, and then rebuild the index.

```
exec ctx_ddl.set_attribute('mystore', 'SEPARATE_OFFSETS', 'F');
alter index idx rebuild('replace storage mystore');
```

> **WARNING:**
>
> Do not use replace metadata operation to disable the SEPARATE_OFFSETS index option, as it can leave the index in an inconsistent state.

To enable the SEPARATE_OFFSETS option for a partitioned index without rebuilding the index, modify the basic storage preference by setting the value of its SEPARATE_OFFSETS storage attribute to T, replace the global index metadata by using ctx_ddl.replace_index_metadata, and then call optimize_index in REBUILD mode for each partition in the partitioned index table.

The following example enables the SEPARATE_OFFSETS option for the partitioned idx index:

```
exec ctx_ddl.set_attribute('mystore', 'SEPARATE_OFFSETS', 'T');
exec ctx_ddl.replace_index_metadata('idx', 'replace storage mystore');
exec ctx_ddl.optimize_index('idx', 'rebuild', part_name=>'part1');
```

> **Note:**
>
> If a procedure modifies the existing index tables with only the SEPARATE_OFFSETS option enabled, then the data is not reindexed.

10.2.9 Improved Response Time Using the STAGE_ITAB, STAGE_ITAB_MAX_ROWS, and STAGE_ITAB_PARALLEL Options of CONTEXT Index

Oracle Text provides the STAGE_ITAB option for improving the query performance for CONTEXT and search indexes that extensively use insert, update, and delete operations for near real-time indexing.

The STAGE_ITAB option is the default index option only for search indexes.

If you do not use the STAGE_ITAB index option, then when you add a new document to the CONTEXT index, SYNC_INDEX is called to make the documents searchable. This call creates new rows in the $I table, and increases the fragmentation in the $I table. The result is deterioration of the query performance.

When you enable the STAGE_ITAB index option, the following happens:

- Information about the new documents is stored in the $G staging table, not in the $I table. This storage ensures that the $I table is not fragmented and does not deteriorate the query performance.
- The $H b-tree index is created on the $G table. The $G table and $H b-tree index are equivalent to the $I table and $X b-tree index.

Rows are merged automatically from the $G table to the $I table when the number of rows in $G exceeds the storage setting, STAGE_ITAB_MAX_ROWS (10K by default). You can also force an immediate merge of the rows from $G to $I by running index optimization in MERGE optimization mode.

> **Note:**
>
> The $G table is stored in the KEEP pool. To improve query performance, you should allocate sufficient KEEP pool memory and maintain a large enough $G table size by using the new stage_itab_max_rows option.

To create a CONTEXT index with the STAGE_ITAB index option, first create a basic storage preference by setting the value of its STAGE_ITAB storage attribute to YES. Next, specify this storage preference when you create the CONTEXT index.

The following example creates a basic mystore storage preference and sets the value of its STAGE_ITAB storage attribute to YES:

```
exec ctx_ddl.create_preference('mystore', 'BASIC_STORAGE');
exec ctx_ddl.set_attribute('mystore', 'STAGE_ITAB', 'YES');
```

You can also enable the STAGE_ITAB index option for an existing nonpartitioned CONTEXT index by using the rebuild option of the ALTER INDEX statement.

```
alter index IDX rebuild parameters('replace storage mystore');
```

To disable the STAGE_ITAB option for a nonpartitioned CONTEXT index, update the existing storage preference (mystore) by setting the value of its STAGE_ITAB storage attribute to NO, and then rebuild the index.

```
exec ctx_ddl.set_attribute('mystore', 'STAGE_ITAB', 'NO');
alter index idx rebuild parameters('replace storage mystore');
```

This operation runs the optimization process by using the MERGE optimization mode and then drops the $G table.

The rebuild option of the ALTER INDEX statement does not work with the partitioned CONTEXT index for enabling and disabling the STAGE_ITAB option.

The following example enables the STAGE_ITAB option for the partitioned CONTEXT idx index:

```
alter index idx parameters('add stage_itab');
```

The following example disables the STAGE_ITAB option for the partitioned CONTEXT idx index:

```
alter index idx parameters('remove stage_itab');
```

The contents of $G were automatically moved to $I during index synchronization when $G had more than 1 million rows in Oracle Database 12*c* Release 2 (12.2) or 100K rows in Oracle Database Release 18c. Starting with Oracle Database Release 21c, the contents of $G are automatically moved to $I during index synchronization when $G has more than 10K rows by default. This value is controlled by the STAGE_ITAB_MAX_ROWS attribute of the STORAGE preference.

> **Note:**
>
> To use the STAGE_ITAB index option for a CONTEXT index, you must specify the g_index_clause and g_table_clause BASIC_STORAGE preferences.

The query performance is deteriorated when $G table is too fragmented. To avoid deterioration, starting with Oracle Database Release 18c, Oracle Text provides automatic background optimize merge for every index or partition. To enable automatic background optimize merge, you must set the STAGE_ITAB storage preference attribute to TRUE, and you must create the index with a storage preference which uses the STAGE_ITAB attribute.

By default, if you had enabled STAGE_ITAB in indexes before you upgraded to Oracle Database Release 18c, then STAGE_ITAB_AUTO_OPT is not enabled. If STAGE_ITAB and AUTO_OPTIMIZE are enabled in existing indexes, then you must disable AUTO_OPTIMIZE before you enable STAGE_ITAB_AUTO_OPT. Starting with Oracle Database Release 19c, STAGE_ITAB_AUTO_OPT is set to TRUE by default for automatic background optimize merge. If you set STAGE_ITAB_AUTO_OPT to FALSE, the merge is run as part of SYNC INDEX. It is recommended to set STAGE_ITAB and STAGE_ITAB_AUTO_OPT to TRUE instead of using AUTO_OPTIMIZE.

> **Note:**
>
> In Oracle Database Release 21c, the procedures ADD_AUTO_OPTIMIZE and REMOVE_AUTO_OPTIMIZE, and the views CTX_AUTO_OPTIMIZE_INDEXES, CTX_USER_AUTO_OPTIMIZE_INDEXES and CTX_AUTO_OPTIMIZE_STATUS are deprecated.

The following example creates a basic mystore storage preference and sets the value of its STAGE_ITAB_AUTO_OPT storage attribute to TRUE:

```
exec ctx_ddl.create_preference('mystore', 'basic_storage');
exec ctx_ddl.set_attribute('mystore', 'stage_itab', 'TRUE');
exec ctx_ddl.set_attribute('mystore', 'stage_itab_auto_opt', 'TRUE');
exec ctx_ddl.set_attribute('mystore', 'stage_itab_parallel', 16);
```

Related Topics

- *Oracle Text Reference*

10.3 Optimizing Queries for Throughput

When you optimize a query for throughput, the default behavior returns all hits in the shortest time possible.

Here is how you can explicitly optimize queries for throughput:

- **CHOOSE and ALL ROWS Modes:** By default, you optimize queries with the CHOOSE and ALL_ROWS modes. 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 throughput with FIRST_ROWS(n), then use the DOMAIN_INDEX_NO_SORT hint. Better throughput means that you are interested in getting all query rows in the shortest time possible.

  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 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 SQL Tuning Guide* for more information about the query optimizer and using hints such as `FIRST_ROWS(n)` and `CHOOSE`

10.4 Composite Domain Index in Oracle Text

The Composite Domain Index (CDI) feature of the Extensibility Framework in Oracle Database 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 queries is improved:

- Oracle Text query with structured criteria in the SQL `WHERE` clause
- Oracle Text query with structured `ORDER BY` criteria
- A combination of the previous two query types

As with concatenated b-tree indexes or bitmap indexes, applications experience a slowdown in data manipulation language (DML) performance as the number of `FILTER BY` and `ORDER BY` columns increases. Where `SCORE`-sort pushdown is optimized for response time, the structured sort or combination of `SCORE` and structured sort pushdown is also optimized for response time, but not for throughput. However, using `DOMAIN_INDEX_SORT` or `FIRST_ROWS(n)` hints to force the sort to be pushed into the CDI while fetching the entire hitlist may result in poor query response time.

10.5 Performance Tuning with CDI

Because you can map a `FILTER BY` column to `MDATA`, you can optimize query performances for equality searches by restricting the supported functionality of `RANGE` and `LIKE`. However, Oracle does not recommend mapping a `FILTER BY` column to `MDATA` 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 the `DATE` stamp. For such sequential columns, mapping to `SDATA` is recommended.

Use the following hints to push or not push the `SORT` and `FILTER BY` predicates into the CDI:

- `DOMAIN_INDEX_SORT`: The query optimizer tries to push the applicable sorting criteria into the specified CDI.
- `DOMAIN_INDEX_NO_SORT`: The query optimizer tries **not** to push sorting criteria into the specified CDI.
- `DOMAIN_INDEX_FILTER`(*table name index name*): The query optimizer tries to push the applicable `FILTER BY` predicates into the specified CDI.
- `DOMAIN_INDEX_NO_FILTER`(*table name index name*): The query optimizer does not try to push the applicable `FILTER BY` predicate(s) into the specified CDI.

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

Example 10-1 Performance Tuning an Oracle Text Query with CDI Hints

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

```
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;
```

10.6 Solving Index and Query Bottlenecks by Using Tracing

Tracing 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. `CTX_OUTPUT` includes a symbol for this number. Each trace measures a specific numeric quantity, such as 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 is 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 synchronization or optimization, the trace profile is copied to the secondary sessions if and only if tracing is currently enabled. Each secondary session accumulates its own traces and implicitly writes all trace values to its logfile before termination.

Related Topics

- *Oracle Text Reference*

10.7 Using Parallel Queries

Oracle Text supports parallel queries on a local `CONTEXT` index and across Oracle Real Application Clusters (Oracle RAC) nodes.

In general, parallel queries are optimal for Decision Support System (DSS). They are also optimal for analytical systems that have large data collections, multiple CPUs with a low number of concurrent users, or Oracle RAC nodes.

Related Topics

- Parallel Queries on a Local Context Index
  Parallel query refers to the parallelized processing of a local `CONTEXT` index.
- Parallelizing Queries Across Oracle RAC Nodes
  Oracle Real Application Clusters (Oracle RAC) enables you to improve query throughput and scalability as the query load increases.

10.7.1 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 workers to be spawned to process the index. Each parallel query worker processes one or more index partitions. This default query behavior applies to local indexes that are created in parallel.

However, for heavily loaded systems with a high number of concurrent users, query throughput is usually not effective with parallel query; if the query is run serially, the top-N hits can usually be satisfied by the first few partitions. For example, take the typical top-N text queries with an `ORDER BY` partition key column:

```
select * from (
        select story_id from stories_tab where contains(...)>0 order by
publication_date desc)
    where rownum <= 10;
```

These text queries generally do not perform well with a parallel query.

You can disable parallel querying after a parallel index operation with an `ALTER INDEX` statement:

```
Alter index <text index name> NOPARALLEL;
Alter index <text index name> PARALLEL 1;
```

You can also enable or increase the parallel degree:

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

10.7.2 Parallelizing Queries Across Oracle RAC Nodes

Oracle Real Application Clusters (Oracle RAC) enables you to improve query throughput and scalability as the query load increases.

You can achieve further improvements in Oracle Text performance by physically partitioning the text data and Oracle 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.

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

Although Oracle RAC offers solutions for improving query throughput and performance, it does 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 amount of memory available to the system global area (SGA) cache or by partitioning your data so that queries do not have to hit all table partitions in order to provide the required set of query results.

10.8 Tuning Queries with Blocking Operations

If you issue a query with more than one predicate, you 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 CBO 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 `BITMAP AND` is a blocking operation, Oracle Text must temporarily save the rowid and score pairs returned from the Oracle Text domain index before it runs the `BITMAP AND` operation.

Oracle Text attempts to save these rowid and score pairs in memory. However, when the size of the result set 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.

```
alter session set SORT_AREA_SIZE = <new memory size in bytes>;
```

For example, set the buffer to approximately 8 megabytes.

```
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`

10.9 Frequently Asked Questions About Query Performance

This section answers some of the frequently asked questions about query performance.

10.9.1 What is query performance?

Answer: There are two measures of query performance:

- **Response time:** The time to get an answer to an individual query
- **Throughput:** The number of queries that can be run in any given time period; for example, queries each second

These two measures are related, but they are not the same. In a heavily loaded system, you 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 hits to the user, whereas others only require the first 20 hits from an ordered set. It is important to distinguish between these two scenarios.

10.9.2 What is the fastest type of Oracle Text query?

Answer: The fastest type of query meets the following conditions:

- Single `CONTAINS` clause

- No other conditions in the `WHERE` clause
- No `ORDER BY` clause
- Returns only the first page of results (for example, the first 10 or 20 hits)

10.9.3 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.

10.9.4 How does the size of my data affect queries?

Answer: The speed at which the Oracle Text index can deliver rowids is not affected by the actual size of the data. Oracle Text query speed is related to the number of rows that must be fetched from the index table, the number of hits requested, the number of hits produced by the query, and the presence or absence of sorting.

10.9.5 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 makes a difference. Spell-checked and subedited text for publication tends to have a much smaller total vocabulary (and therefore size of the index table) than informal text such as email, which contains spelling errors and abbreviations. For a given index memory setting, the extra text takes up memory, creates more fragmented rows, and adversely affects query response time.

10.9.6 What is the difference between an indexed lookup and a functional lookup

Answer: The kernel can query the Oracle Text index with an indexed lookup and a functional lookup. In the indexed lookup, the first and most common case, the kernel asks the Oracle Text index for all rowids that satisfy a particular text search. These rowids are returned in batches.

In the functional lookup, the kernel passes individual rowids to the Oracle Text index and asks whether that particular rowid satisfies a certain text criterion. The functional lookup is most commonly used with a very selective structured clause, so that only a few rowids must be checked against the Oracle Text index. Here is an example of a search where a functional lookup is useful:

```
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 an Oracle Text query that is ordered by a structured column (for example date, price) and if the Oracle Text query contains unselective words.

10.9.7 What tables are involved in queries?

Answer: All queries look at the index token table. The table's name has the form of `DR$indexname$I` and contains the list of tokens (`TOKEN_TEXT` column) and the information about the row and word positions where the token occurs (`TOKEN_INFO` column).

The row information is stored as internal docid values that must be translated into external rowid values. The table that you use depends on the type of lookup:

- For functional lookups, use the `$K` table, `DR$indexname$K`. This simple Index Organized Table (IOT) contains a row for each docid/rowid pair.
- For indexed lookups, use the `$R` table, `DR$indexname$R`. This table holds the complete list of rowids in a BLOB column.

Starting with Oracle Database 12*c* Release 2 (12.2), a new storage attribute, `SMALL_R_ROW`, was introduced to reduce the size of the `$R` row. It populates `$R` rows on demand instead of creating 22 static rows, thereby reducing the Data Manipulation Language contention. The contention happens when parallel insert, update, and delete operations try to lock the same `$R` row.

You 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.

10.9.8 How is the $R table contention reduced?

The `$R` contention during base table delete and update operations has become a recurring theme over the past few years. Currently, each `$R` index table has 22 static rows, and each row can contain up to 200 million rowids. The contention happens when the parallel insert, update, and delete operations try to lock the same `$R` row for insert or delete operations. The following enhancements made during this release reduce the contention:

- The maximum number of rowids that each `$R` row can contain is 70,000, which translates to 1 MB of data stored on each row. To use this feature, you must set the `SMALL_R_ROW` storage attribute.
- The `$R` rows are created on demand instead of just populating a pre-determined number of rows.

10.9.9 Does sorting the results slow a text-only query?

Answer: Yes, it certainly does.

If Oracle Text does not sort, then it can return results as it finds them. This approach is quicker when the application needs to display only a page of results at a time.

10.9.10 How do I make an ORDER BY score query faster?

Answer: Sorting by relevance (`SCORE(n)`) can be fast if you use the `FIRST_ROWS(n)` hint. In this case, Oracle Text performs a high-speed internal sort when fetching from the Oracle Text index tables.

Here is an example of this query:

```
SELECT /*+ FIRST_ROWS(10) */ ID, SCORE(1), TEXT FROM mytable
  WHERE CONTAINS (TEXT, 'searchterm', 1) > 0
  ORDER BY SCORE(1) DESC;
```

It is important to note that, there must be no other criteria in the `WHERE` clause, other than a single `CONTAINS`.

10.9.11 Which memory settings affect querying?

Answer: For querying, you want to strive for a large system global area (SGA). You can set these SGA parameters in your Oracle Database initialization file. You can also set these parameters dynamically.

The `SORT_AREA_SIZE` parameter controls the memory that is available for sorting `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
- *Oracle Database Reference* for more information on setting the `SORT_AREA_SIZE` parameter

10.9.12 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 large objects (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. It makes it easier for Oracle Text to cache the entire table to memory, and so reduces the cost of selecting columns from the base table, and speeds up text queries.

In addition, smaller base tables cached in memory enables more index table data to be cached during querying, which improves performance.

10.9.13 How can I speed up a CONTAINS query on more than one column?

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
```

You can get the same result with section searching and the `WITHIN` operator:

```
SELECT title, isbn FROM booklist
  WHERE CONTAINS (alltext,
    'horse WITHIN title AND racing WITHIN abstract')>0
```

This query is completed more quickly than the single `CONTAINS` clause. To use a query like this, you must copy all data into a single text column for indexing, with section tags around each column's data. You can do that with PL/SQL procedures before indexing, or you can use the `USER_DATASTORE` datastore during indexing to synthesize structured columns with the text column into one document.

10.9.14 Can I 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 a large number of expansions (80 or 100) in a query. Use the query feedback mechanism to determine the number of expansions for any particular query expression.

For wildcard and stem queries, you can avoid 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 creates 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
```

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;
```

10.9.15 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, the results from the multiple indexes are combined as necessary to produce the final result set.

Use the `LOCAL` keyword to create the index:

```
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 query restricts the search to a particular range of values on a column that is also the partition key.
- **ORDER BY Partition Key Column:** This query 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"

10.9.16 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 degrades the overall query throughput on heavily loaded systems.

Parallel queries are optimal for Decision Support System (DSS). They are also optimal for analytical systems that have large data collections, multiple CPUs with a low number of concurrent users, or Oracle Real Application Clusters (Oracle RAC) nodes.

Related Topics

- Using Parallel Queries
  Oracle Text supports parallel queries on a local `CONTEXT` index and across Oracle Real Application Clusters (Oracle RAC) nodes.

10.9.17 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 your application uses many `ABOUT` queries, 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"

10.9.18 When should I use a CTXCAT index?

Answer: CTXCAT indexes work best when the text is in small chunks (just a few lines), and you want searches to restrict or sort the result set according to certain structured criteria, such as numbers or dates.

For example, consider an online auction site. Each item for sale has a short description, a current bid price, and start and end dates for the auction. You want to see all records with *antique cabinet* in the description, with a current bid price less than $500. Because you are particularly interested in newly posted items, you want the results sorted by auction start time.

This search is not always efficient with a CONTAINS structured query on a CONTEXT index. The response time can vary significantly depending on the structured and CONTAINS clauses, because the intersection of structured and CONTAINS clauses or the Oracle Text query ordering is computed during query time.

By including structured information within the CTXCAT index, you ensure that the query response time is always in an optimal range regardless of search criteria. Because the interaction between text and structured query is precomputed during indexing, query response time is optimum.

> **Note:**
>
> The Oracle Text indextype CTXCAT is deprecated with Oracle Database 23ai. The indextype itself, and it's operator CTXCAT, can be removed in a future release. Both CTXCAT and the use of CTXCAT grammar as an alternative grammar for CONTEXT queries is deprecated. Instead, Oracle recommends that you use the CONTEXT indextype, which can provide all the same functionality, except that it is not transactional. Near-transactional behavior in CONTEXT can be achieved by using SYNC(ON COMMIT) or, preferably, SYNC(EVERY [*time-period*]) with a short time period.
>
> CTXCAT was introduced when indexes were typically a few megabytes in size. Modern, large indexes, can be difficult to manage with CTXCAT. The addition of index sets to CTXCAT can be achieved more effectively by the use of FILTER BY and ORDER BY columns, or SDATA, or both, in the CONTEXT indextype. CTXCAT is therefore rarely an appropriate choice. Oracle recommends that you choose the more efficient CONTEXT indextype.

10.9.19 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 they use considerably more disk space than CONTEXT indexes.

If you are tight on disk space, consider carefully whether CTXCAT indexes are appropriate for you.

With query operators, you can use the richer CONTEXT grammar in CATSEARCH queries with query templates. The older restriction of a single CATSEARCH query grammar no longer holds.

> **Note:**
>
> The Oracle Text indextype CTXCAT is deprecated with Oracle Database 23ai. The indextype itself, and it's operator CTXCAT, can be removed in a future release. Both CTXCAT and the use of CTXCAT grammar as an alternative grammar for CONTEXT queries is deprecated. Instead, Oracle recommends that you use the CONTEXT indextype, which can provide all the same functionality, except that it is not transactional. Near-transactional behavior in CONTEXT can be achieved by using SYNC(ON COMMIT) or, preferably, SYNC(EVERY [*time-period*]) with a short time period.
>
> CTXCAT was introduced when indexes were typically a few megabytes in size. Modern, large indexes, can be difficult to manage with CTXCAT. The addition of index sets to CTXCAT can be achieved more effectively by the use of FILTER BY and ORDER BY columns, or SDATA, or both, in the CONTEXT indextype. CTXCAT is therefore rarely an appropriate choice. Oracle recommends that you choose the more efficient CONTEXT indextype.

10.9.20 What optimizer hints are available and what do they do?

Answer: To drive the query with a text or b-tree index, you can use the INDEX(table column) optimizer hint in the usual way.

You can also use the NO_INDEX(table column) hint to disable a specific index.

The FIRST_ROWS(n) hint has a special meaning for text queries. Use it when you need the first n hits to a query. When you use the DOMAIN_INDEX_SORT hint in conjunction with ORDER BY SCORE(n) DESC, you tell the Oracle optimizer to accept a sorted set from the Oracle Text index and to sort no farther.

> **See Also:**
>
> "Optimizing Queries for Response Time"

10.10 Frequently Asked Questions About Indexing Performance

This section answers some of the frequently asked questions about indexing performance.

- How long should indexing take?
- Which index memory settings should I use?
- How much disk overhead will indexing require?
- How does the format of my data affect indexing?

10.10.1 How long should indexing take?

Answer: Indexing text is a resource-intensive process. The speed of indexing depends on the power of your hardware. Indexing speed depends on CPU and I/O capacity. With sufficient I/O capacity to read in the original data and write out index entries, the CPU is the limiting factor.

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

10.10.2 Which index memory settings should I use?

Answer: You can set your index memory with the `DEFAULT_INDEX_MEMORY` and `MAX_INDEX_MEMORY` system parameters. You can also set your index memory at runtime 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 fragmentation of the final indexes. However, if you set the index memory setting too high, then memory paging reduces 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 SGA block buffer cache as possible. So you should allocate a large amount of memory to the block buffer cache. Because this approach does not make any difference to indexing, you would be better off reducing the size of the SGA to make more room for 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
- *Oracle Database Reference* for more information on setting the `SORT_AREA_SIZE` parameter

10.10.3 How much disk overhead will indexing require?

Answer: The overhead, the amount of space needed for the index tables, varies between about 50 and 200 percent of the original text volume. Generally, larger amounts of text result in smaller overhead, but many small records use more overhead than fewer large records. Also, clean data (such as published text) requires less overhead than dirty data such as emails or discussion notes, because the dirty data is likely to include many misspelled and abbreviated words.

A text-only index is smaller than a combined text and theme index. A prefix and substring index makes the index significantly larger.

10.10.4 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 the documents tend to be very large compared to the actual text held in them. So 1 GB of Word documents might only require 50 MB of index space, whereas 1 GB of plain text might require 500 MB, 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 has an obvious effect, you must balance this against the cost of filtering the documents with the `AUTO_FILTER` filter or other user-defined filters.

10.10.5 Can parallel indexing improve performance?

Answer: Parallel indexing can improve index performance when you have a large amount of data and multiple CPUs.

Use the `PARALLEL` keyword to create an index with up to three separate indexing processes, depending on your resources.

```
CREATE INDEX index_name ON table_name (column_name)
INDEXTYPE IS ctxsys.context PARAMETERS ('...') PARALLEL 3;
```

You can also use parallel indexing to create local partitioned indexes on partitioned tables. However, indexing performance improves only with multiple CPUs.

> **Note:**
>
> Using `PARALLEL` to create a local partitioned index enables parallel queries. (Creating a nonpartitioned 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`.

10.10.6 How can I improve index performance when I create a local partitioned index?

Answer: When you have multiple CPUs, you can improve indexing performance by creating a local index in parallel.

You can create a local partitioned index in parallel in the following ways:

- Use the `PARALLEL` clause with the `LOCAL` clause in the `CREATE INDEX` statement. In this case, the maximum parallel degree is limited to the number of partitions.
- Create an unusable index, and 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 of the second method. The base table has three partitions. You create a local partitioned unusable index first, and then run the `DBMS_PCLUTIL.BUILD_PART_INDEX`, to build the three partitions in parallel (inter-partition parallelism). Inside each partition, index creation occurs 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);
```

10.10.7 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. The filename is normally written to `$ORACLE_HOME/ctx/log`, but you can change the directory by using the `LOG_DIRECTORY` parameter in `CTX_ADM.SET_PARAMETER`.

> **See Also:**
>
> *Oracle Text Reference* to learn more about the `CTX_OUTPUT` package

10.11 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?
- How can I tell when my indexes are getting fragmented?
- Does memory allocation affect index synchronization?

10.11.1 How often should I index new or updated records?

Answer: If you run reindexing with `CTX_DDL.SYNC_INDEX` less often, your indexes will be less fragmented, and you will not have to optimize them as often.

However, your data becomes progressively more out-of-date, and that may be unacceptable to 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, 10-minute, or 5-minute updates.

> **See Also:**
>
> - *Oracle Text Reference* to learn more about using `CTX_DDL.SYNC_INDEX`
> - "Managing DML Operations for a CONTEXT Index"

10.11.2 How can I tell when my indexes are fragmented?

Answer: The best way is to time some queries, run index optimization, and 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 do not, then 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"

10.11.3 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.

11

Searching Document Sections in Oracle Text

You can use document sections in a text query application.

This chapter contains the following topics:

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

11.1 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
- Oracle Text Section Attributes

11.1.1 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 Searching with the WITHIN Operator
5. Path Searching with INPATH and HASPATH Operators
6. Marking an SDATA Section to be Searchable

11.1.1.1 Create a Section Group

You enable section searching by defining section groups. Use one of the system-defined section groups to create an instance of a section group.

You use section groups to specify the type of document set that you have and implicitly indicate the tag structure. Choose a section group that is appropriate for your document collection. For instance, to index HTML tagged documents, use `HTML_SECTION_GROUP`. Likewise, to index XML tagged documents, use `XML_SECTION_GROUP`.

Table 11-1 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 does 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 to index HTML documents and for defining sections in HTML documents.
XML_SECTION_GROUP	Use this group type to index 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. As in XML, the section names derived from XML tags are case-sensitive. 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.) • The length of the indexed tags, including prefix and namespace, cannot exceed 64 bytes. Tags longer than 64 bytes are not indexed.
PATH_SECTION_GROUP	Use this group type to index XML documents. This preference behaves like AUTO_SECTION_GROUP. The difference is that you can search paths with the INPATH and HASPATH operators. Queries are also case-sensitive for tag and attribute names.
NEWS_SECTION_GROUP	Use this group to define sections in newsgroup-formatted documents according to RFC 1036.

Notes

- Documents sent to the HTML, XML, AUTO, and PATH sectioners must begin with \s*<. The \s* represents zero or more whitespace characters. Otherwise, the document is treated as a plain-text document, and no sections are recognized.
- Do not use left-angle-brackets within a section data. If a left-angle-bracket is followed by a non-blank character, then the section parser treats the free text (between the left-angle and right-angle brackets) as a tag name.

  For example:

```
<DOCUMENT> <BODYTEXT> ABC
< R1 NOMISS
```

```
<R2 MISSED1 </BODYTEXT> <FIELDS> DEF
<R3 MISSED2 </FIELDS> </DOCUMENT> GHI FALSEPOSITIVE JKL
```

In the preceding example, the section parser treats `R2 MISSED1 </BODYTEXT` and `R3 MISSED2 </FIELDS` as tag names. This may result in false-positive hits or missed hits, which may cause the following issues:

- Any word in the `R2 MISSED1 </BODYTEXT` and `R3 MISSED2 </FIELDS` phrases are not searchable within a section.
- Any text outside a section, such as `GHI`, `FALSEPOSITIVE`, and `JKL` are wrongly included in a section if it is not closed.

- 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;
  ```

- Starting with Oracle Database 18c, use of `NEWS_SECTION_GROUP` is deprecated in Oracle Text. Use external processing instead.
  If you want to index USENET posts, then preprocess the posts to use `BASIC_SECTION_GROUP` or `HTML_SECTION_GROUP` within Oracle Text. USENET is rarely used commercially.

11.1.1.2 Define Your Sections

You define sections as part of the section group. The following example defines a 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 `AUTO_SECTION_GROUP` or `PATH_SECTION_GROUP` to index an XML document collection, then you do not have to explicitly define sections. The system defines the sections during indexing.

> **See Also:**
>
> - "Oracle Text Section Types" for more information about sections
> - "XML Section Searching with Oracle Text" for more information about section searching with XML

11.1.1.3 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');
```

11.1.1.4 Search Sections with the WITHIN Operator

When your documents are indexed, you can query within sections by using the `WITHIN` operator. For example, to find all 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

11.1.1.5 Search Paths with INPATH and HASPATH Operators

When you use `PATH_SECTION_GROUP`, the system automatically creates XML sections. 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" to learn more about using these operators
> - *Oracle Text Reference* to learn more about using the `INPATH` operator

11.1.1.6 Mark an SDATA Section to Be Searchable

To mark an `SDATA` section to be searchable and have a `$S`*`datatype`* table created, use the `CTX_DDL.SET_SECTION_ATTRIBUTE` API.
The following tables are created:

- `$SN` – `NUMBER`
- `$SD` – `DATE`
- `$SV` – `VARCHAR2`, `CHAR`
- `$SR` – `RAW`
- `$SBD` – `BINARY DOUBLE`
- `$SBF` – `BINARY FLOAT`
- `$ST` – `TIMESTAMP`

- $STZ — TIMESTAMP WITH TIMEZONE

The following example creates a $SV table for this SDATA section to allow efficient searching on that section.

```
ctx_ddl.add_sdata_section('sec_grp', 'sdata_sec', 'mytag', 'varchar');
ctx_ddl.set_section_attribute('sec_grp', 'sdata_sec', 'optimized_for',
'search');
```

The default value of this attribute is FALSE.

11.1.2 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
- NDATA Section
- SDATA Section
- Attribute Section (for XML documents)
- Special Sections (sentence or paragraphs)

Table 11-2 shows which section types may be used with each kind of section group.

Table 11-2 Section Types and Section Groups

Section Group	ZONE	FIELD	STOP	MDATA	NDATA	SDATA	ATTRIBUTE	SPECIAL
NULL	NO	NO	NO	NO	NO	NO	NO	YES
BASIC	YES	YES	NO	YES	YES	YES	NO	YES
HTML	YES	YES	NO	YES	YES	YES	NO	YES
XML	YES	YES	NO	YES	YES	YES	YES	YES
NEWS	YES	YES	NO	YES	YES	YES	NO	YES
AUTO	NO	NO	YES	NO	NO	NO	NO	NO
PATH	NO	NO	NO	NO	NO	NO	NO	NO

11.1.2.1 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, define the text between the <TITLE> and </TITLE> tags 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 `booktitle` section 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 documents that contain the term *Cities* within the `booktitle` section 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, the heading can be repeated in the same documents as follows:

```
<H1> The Brown Fox </H1>
<H1> The Gray Wolf </H1>
```

Assuming that these zone sections are named `Heading`, a query of *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 be nested, 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 that the `BOOK1`, `BOOK2`, and `AUTHOR` zone sections occur as follows in the doc1 and doc2 documents:

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. It returns only doc1.

```
'(Scott within author) within book1'
```

11.1.2.2 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. Field sections are more efficient from zone sections and are different than zone sections in that the region is indexed separately from the rest of the document. You can create an unlimited number of field sections.

Because field sections are indexed differently, you can also get better query performance over zone sections when a large number of documents are indexed.

Field sections are more suited to a single occurrence of a section in 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:

- They cannot overlap.
- They cannot repeat.
- They 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. You can query text within a visible field section with or without the `WITHIN` operator.

The following example shows the difference using invisible and visible field sections. The code defines a `basicgroup` section group 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.

```
'(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

You cannot nest field sections. For example, if you define a field section to start with `<TITLE>` and define another field section to start with `<FOO>`, you *cannot* nest the two sections 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. Here is an example of a 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.

11.1.2.3 Stop Section

When you add a stop section to an automatic section group, the automatic section indexing operation ignores 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.

You can add an unlimited number of stop sections.

Stop sections do not have section names and are not recorded in the section views.

11.1.2.4 MDATA Section

You use an `MDATA` section to reference user-defined metadata for a document.

`MDATA` sections can speed up mixed queries, and there is no limit to the number of `MDATA` sections that can be returned in a query.

Consider the case where you want to query according to text content and document type (magazine, newspaper, or novel). You can 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, search 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) in a field section, rather than using a separate column, and then using a single CONTAINS query.

```
SELECT id FROM documents
  WHERE CONTAINS(text, 'Adam Thorpe AND novel WITHIN doctype')>0;
```

This approach has two drawbacks:

- Each time the attribute is updated, the entire text document must be reindexed, resulting in increased index fragmentation and slower rates of data manipulation language (DML) processing.
- Field sections tokenize the section value. Tokenization has several effects. Special characters in metadata, such as decimal points or currency characters, are not easily searchable; value searching (searching for *John Smith* but not *John Smith, Jr.*) is difficult; multiword values are queried by phrase, which is slower than single-token searching; and multiword values do not show up in browsed 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 you can add and remove metadata values from documents without the need to reindex 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.

Starting with Oracle Database 12*c* Release 2 (12.2), the MDATA section can be updatable or nonupdatable depending on the value of its read-only tag, which can be set to either FALSE or TRUE.

Use CTX_DDL.ADD_MDATA_SECTION to add an MDATA section to a section group. By default, the value of a read-only MDATA section is FALSE. It implies that you want to permit calling CTX_DDL.ADD_MDATA() and CTX_DDL.REMOVE_MDATA() for this MDATA section, otherwise you can set it to TRUE. When set to FALSE, the queries on the MDATA section run less efficiently because a cursor must be opened on the index table to track the deleted values for that MDATA section. 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');
```

You can change MDATA values with CTX_DDL.ADD_MDATA, and you can remove them with CTX_DDL.REMOVE_MDATA. Also, MDATA sections can have multiple values. Only the owner of the index may call CTX_DDL.ADD_MDATA and CTX_DDL.REMOVE_MDATA.

Neither CTX_DDL.ADD_MDATA nor CTX_DDL.REMOVE_MDATA is supported for CTXCAT and CTXRULE indexes.

MDATA values are not passed through a lexer. Instead, all values undergo the following simplified normalization:

- Leading and trailing whitespace on the value is removed.
- The value is truncated to 255 bytes.
- 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 you add MDATA metadata to a document, you can query for that metadata by using the CONTAINS query operator:

```
SELECT id FROM documents
   WHERE CONTAINS(text, 'Tokyo and MDATA(author, Soseki Natsume)')>0;
```

This query is only successful if an AUTHOR tag has the exact value *Soseki Natsume* (after simplified tokenization). *Soseki* or *Natsume Soseki* returns no rows.

The following are considerations for MDATA:

- MDATA values are not highlightable, do not appear in the output of CTX_DOC.TOKENS, and do not appear when you enable FILTER PLAINTEXT.
- MDATA sections must be unique within section groups. For example, do not use FOO as the name of an MDATA section and a zone or field section 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. 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, because 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:

- "ALTER INDEX" in *Oracle Text Reference*
- "ADD_MDATA_SECTION" in *Oracle Text Reference*
- 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

11.1.2.5 NDATA Section

For fields containing data to be indexed for name searching, you can specify them 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, by using two possible datastores: MULTI_COLUMN_DATASTORE or USER_DATASTORE. The following example uses 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');
 commit;
 begin
   ctx_ddl.create_preference('nameds', 'MULTI_COLUMN_DATASTORE');
   ctx_ddl.set_attribute('nameds', 'columns', 'firstname,surname');
```

```
 end;
 /
```

This example 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;
/
```

Next, create the index by using the datastore preference and section group preference that you created earlier:

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

NDATA sections support both single- and multibyte data with character- and term-based limitations. NDATA section data that is indexed is constrained as follows:

- The number of characters in a single, whitespace-delimited term: 511
- The number of whitespace-delimited terms: 255
- The total number of characters, including whitespaces: 511

11.1.2.6 SDATA Section

The value of an SDATA section is extracted from the document text like other sections, but it is indexed as structured data, also referred to as SDATA.

SDATA sections support operations such as projection, range searches, and ordering. SDATA sections also enable SDATA indexing of section data (such as embedded tags) and detail table or function invocations. You can perform various combinations of text and structured searches in one single SQL statement.

Use SDATA operators 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 example, "find documents with DOG that also have price > 5", rather than "find documents with rating > 4".

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

> **Note:**
>
> The `UPDATE_SDATA` API in Oracle Text is deprecated in Oracle Database 23ai. Instead of modifying the index, Oracle recommends that you update the underlying data.

After you create an `SDATA` section, you can further modify the attributes of the `SDATA` section by using `CTX_DDL.SET_SECTION_ATTRIBUTE`.

Create the `items` table:

```
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 the `my_sec_group SDATA` section:

```
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>
```

Consider a document whose rowid is 1. This example updates the value of the `price` `SDATA` section to a new value of 30000:

```
BEGIN
    SELECT ROWID INTO rowid_to_update FROM items WHERE id=1;

    CTX_DDL.UPDATE_SDATA('items$doc',
                         'price',
                         SYS.ANYDATA.CONVERTVARCHAR2('30000'),
                         rowid_to_update);
END;
```

After executing the query, the price of `Honda Pilot` is changed from 27000 to 30000.

Note:

- You can also add an `SDATA` section to an existing index. Use the `ADD SDATA SECTION` parameter of the `ALTER INDEX PARAMETERS` statement. See the "ALTER INDEX" section of the *Oracle Text Reference* for more information.
- Documents that were indexed before adding an `SDATA` section do not reflect this new preference. Rebuild the index in this case.

See Also:

- The "`CONTAINS`" query section of the *Oracle Text Reference* for information on the `SDATA` operator
- The "`CTX_DDL`" package section of the *Oracle Text Reference* for information on adding and updating the `SDATA` sections and changing their attributes by using the `ADD_SDATA_SECTION`, `SET_SECTION_ATTRIBUTE`, and the `UPDATE_SDATA` procedures

Storage

For `optimized_for` search `SDATA` sections, use `CTX_DDL.SET_ATTRIBUTE` to specify the storage preferences for the `$S`*`datatype`* tables and the indexes on these tables.

By default, large object (LOB) caching is turned on for `$S*` tables and off for `$S*` indexes. These attributes are valid only on `SDATA` sections.

Query Operators

`optimized_for search SDATA` supports the following query operators:

- =
- <>

- between
- not between
- <=
- <
- >=
- >
- is null
- is not null
- like
- not like

11.1.2.7 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"

11.1.2.8 Special Sections

Special sections are not recognized by tags. Currently, sentence and paragraph are the only supported special sections, and they enable you to search for a combination of words within sentences or paragraphs.

The sentence and paragraph boundaries are determined by the lexer. For example, `BASIC_LEXER` recognizes sentence and paragraph section boundaries as follows:

Table 11-3 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, then 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 searches within sentences in HTML documents:

```
begin
ctx_ddl.create_section_group('htmgroup', 'HTML_SECTION_GROUP');
ctx_ddl.add_special_section('htmgroup', 'SENTENCE');
end;
```

To enable zone and sentence searches, add zone sections to the group. The following example adds the `Headline` zone section to the `htmgroup` section group:

```
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;
```

11.1.3 Oracle Text Section Attributes

Section attributes are the settings for the Oracle Text sections of tokenized type, such as field, zone, hybrid, and `SDATA`. Section attributes improve query performance because of the finer control at the section level, rather than at the document level or index level.

By using the section attributes, you can specify:

- Lexer preferences on certain sections of a document. The preferences are useful for part-name searches, when a section of a document containing a part name needs to be lexed differently than the rest of the document. You can also use the lexer preferences for handling multilanguage documents, where there is a section to language mapping.
- A substring index only on certain sections of a document. This index helps reduce the index size.
- Prefix tokens only on certain sections of a document. The prefix tokens improve the performance of right-truncated queries, but can also cause the index size to grow rapidly. Specifying prefix indexing only on certain sections provides improved performance for the right-truncated queries on the specific sections, without rapidly growing the size of the index.
- Stoplists for certain sections of a document.
- A new section type that combines the flexibility of zone sections with the performance of field sections. Currently, zone sections have poor performance compared with field sections. However, field sections do not support nested section search.

To set section attributes, use the `CTX_DDL.SET_SECTION_ATTRIBUTE` procedure.

Table 11-4 lists the section attributes that you can use:

Table 11-4 Section Attributes

Section Attribute	Description
`visible`	Use the `visible` attribute for all section types that are tokenized, except the zone section type. Thus, the `visible` attribute can be used for field, hybrid, and `SDATA` section types. Specify `TRUE` to make the text visible within a document. The text in the field section is indexed as part of the enclosing document. The default is `FALSE`. The text in the field section is indexed separately from the rest of the document. For the Field section type, the visible attribute overrides the value specified in the `CTX_DDL.ADD_FIELD_SECTION` procedure.

Table 11-4 (Cont.) Section Attributes

Section Attribute	Description
`lexer`	Use the `lexer` attribute for all section types that are tokenized (field, zone, hybrid, and `SDATA` sections). Specify the lexer preference name to decide the tokenization of an SDATA section. The default is `NULL`, and the lexer for the main document is used. The lexer preference must be valid at the time of calling the `set_section_attribute` procedure. If you try to drop one of the preferences when an existing field section refers to a lexer preference, then the `drop_preference` procedure fails.
`wordlist`	Use the `wordlist` attribute for all section types that are tokenized (field, zone, hybrid, and `SDATA` sections). To enable section-specific prefix indexing and substring indexing, specify the wordlist preference name for a section. The default is `NULL`, and the wordlist for the main document is used. The wordlist preference must be valid at the time of calling the `set_section_attribute` procedure. If you try to drop one of the preferences when an existing field section refers to a wordlist preference, then the `drop_preference` procedure fails.
`stoplist`	Use the `stoplist` attribute for all section types that are tokenized (field, zone, hybrid, and `SDATA` sections). To enable a section-specific stoplist, specify the stoplist preference name. The default is `NULL`, and the stoplist for the main document is used. The stoplist preference must be valid at the time of calling the `set_section_attribute` procedure. If you try to drop one of the preferences when an existing field section refers to a stoplist preference, then the `drop_preference` procedure fails.

The following example enables the `visible` attribute of a Field section:

```
begin
ctx_ddl.create_section_group('fieldgroup', 'BASIC_SECTION_GROUP');
ctx_ddl.add_field_section('fieldgroup', 'author', 'AUTHOR');
ctx_ddl.set_section_attribute('fieldgroup', 'author', 'visible', 'true');
end;
```

> **See Also:**
>
> *Oracle Text Reference* for the syntax of `CTX_DDL.SET_SECTION_ATTRIBUTE` procedure.

11.2 HTML Section Searching with Oracle Text

HTML has internal structure in the form of tagged text that you can use for section searching. For example, define a section called `headings` for the `<H1>` tag, and then 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. For example, if you want 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

11.2.1 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 the htmgroup section group, you can query within the heading section by issuing this query:

```
'Oracle WITHIN heading'
```

11.2.2 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.

Consider an HTML document that has the following META tag:

```
<META NAME="author" CONTENT="ken">
```

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 the htmgroup section group, you can query the document:

```
'ken WITHIN author'
```

11.3 XML Section Searching with Oracle Text

Like HTML documents, XML documents have tagged text that you can use to define blocks of text for section searching. You can search the contents of a section with the WITHIN or INPATH operators.

The following sections describe the different types of XML searching:

- Automatic Sectioning
- Attribute Searching
- Document Type Sensitive Sections
- Path Section Searching

11.3.1 Automatic Sectioning

To set up your indexing operation to automatically create sections from XML documents, use the `AUTO_SECTION_GROUP` section group. The system creates zone sections for XML tags. Attribute sections are created for the tags that have attributes and for the sections named in the form `tag@attribute`.

For example, the following statement uses the `AUTO_SECTION_GROUP` to create the *myindex* index on a column containing the XML files:

```
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'
           );
```

11.3.2 Attribute Searching

You can search XML attribute text in one of two ways:

- **Creating Attribute Sections**

  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.

  Consider an XML file that defines the `BOOK` tag with a `TITLE` attribute:

  ```
  <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:

  ```
  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'
             );
  ```

To query the *booktitle* XML attribute section:

```
'Cities within booktitle'
```

- **Searching Attributes with the INPATH Operator**

  Index with the `PATH_SECTION_GROUP` and query attribute text with the `INPATH` operator.

> **See Also:**
>
> "Path Section Searching"

11.3.3 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 to improve search capability. The following scenario shows you how to create book sections for each document type.

Assume that `mydocname1` is declared as an XML document type (root element):

```
<!DOCTYPE mydocname1 ... [...
```

Within `mydocname1`,, the `<book>` element is declared. For this tag, you can create a section named `mybooksec1` that is sensitive to the tag's document type:

```
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):

```
<!DOCTYPE mydocname2 ... [...
```

Within `mydocname2`,, the `<book>` element is declared. For this tag, you can create a section named `mybooksec2` that is sensitive to the tag's document type:

```
begin

ctx_ddl.create_section_group('myxmlgroup', 'XML_SECTION_GROUP');
ctx_ddl.add_zone_section('myxmlgroup', 'mybooksec2', 'mydocname2(book)');

end;
```

To query within the `mybooksec1` section, use `WITHIN`:

```
'oracle within mybooksec1'
```

11.3.4 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 search paths 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`

This section contains the following topics.

11.3.4.1 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.

11.3.4.2 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)
```

11.3.4.3 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>
```

11.3.4.4 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 it does not find:

```
<C><B>My dog is friendly.</B></C>
```

11.3.4.5 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>
```

11.3.4.6 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>
```

11.3.4.7 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 it does not find:

```
<A B="work address">San Francisco, California, USA</A>
```

11.3.4.8 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.

11.3.4.9 Section Equality Testing with HASPATH

You can use the `HASPATH` operator for section quality tests. For example, consider the following query:

```
dog INPATH A
```

It 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, not for the second document.

> **See Also:**
>
> *Oracle Text Reference* to learn more about using the `INPATH` and `HASPATH` operators

12

Using Oracle Text Name Search

Oracle Text provides a name search feature to handle inaccurate data and misspelled names.

This chapter contains the following topics:

12.1 Overview of Name Search

Someone accustomed to the spelling rules of one culture can have difficulty applying those same rules to a name 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. The main advantage of name searching is the ability to handle somewhat inaccurate data.

12.2 Name Search Examples

These examples illustrate how to use `NDATA` sections to search on names.

```
drop table people;

create table people (
  full_name varchar2(2000)
);

insert into people values
('John Doe 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 Doesmith'

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', 'Doe', '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 Doesmith'

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;
```

13

Performing Ubiquitous Search with DBMS_SEARCH APIs

Starting with Oracle Database 23ai, you can use the `DBMS_SEARCH PL/SQL` package for indexing of multiple schema objects in a single index, enabling you to search across the entire database.

13.1 About Ubiquitous Search and Ubiquitous Search Indexes

Ubiquitous search enables you to perform full-text and range-based queries across multiple objects within an entire schema. You can use a ubiquitous search index (or simply a `DBMS_SEARCH` index) to perform ubiquitous searches.

A **ubiquitous search index** is a JSON `SEARCH INDEX` type with predefined set of preferences and settings that are enabled for performing full-text search on tables, views, or JSON Duality views. You use the `DBMS_SEARCH PL/SQL` package to create, manage, and query these indexes.

You can create a `DBMS_SEARCH` index on tables or views over schemas that you have `SELECT` privileges on. You can add data sources, that is tables and views, into this index (without the need to materialize the views). All the columns in the specified sources are indexed and available for full-text or range-based search.

Why Choose a Ubiquitous Search Index?

This indexing technique lets you create indexes across multiple objects, add or remove data sources, and perform full-text or range-based searches within a single data source or across multiple sources using the same index. This simplifies the indexing tasks that previously (prior to Oracle Database 23ai) required you to create multiple individual indexes and manually combine various data sources using the `MULTI_COLUMN_DATASTORE` or `USER_DATASTORE` procedures along with materialized views. Previously, this also required additional methods, such as triggers, to ensure that the index remained synchronized with DML operations.

With a simplified set of `DBMS_SEARCH` APIs, you can perform ubiquitous searches across the database as follows:

- **Create index**:

  The `DBMS_SEARCH.CREATE_INDEX` API allows you to create a `DBMS_SEARCH` index.

  By default, this index is created with key indexing preferences, such as `BASIC_WORDLIST` to allow wildcard search and `SEARCH_ON` to allow both full-text and range-search queries. These indexes are asynchronously maintained in the background at predefined intervals, and thus you do not need to explicitly run the `SYNC_INDEX` and `OPTIMIZE_INDEX` operations on such indexes.

- **Manage data sources**:

  You can define which tables or views should be indexed by adding them as data sources into your index.

The `DBMS_SEARCH.ADD_SOURCE` API allows you to automatically add one or more data sources (such as tables, views, or duality views) from different schemas to this index.

The `DBMS_SEARCH.REMOVE_SOURCE` API allows you to remove a source and all its associated data from the index.

- **View combined indexed data**:

  The `DBMS_SEARCH.GET_DOCUMENT` API allows you to view a virtual document that is indexed, which displays metadata values as indexed for each row of all your data sources.

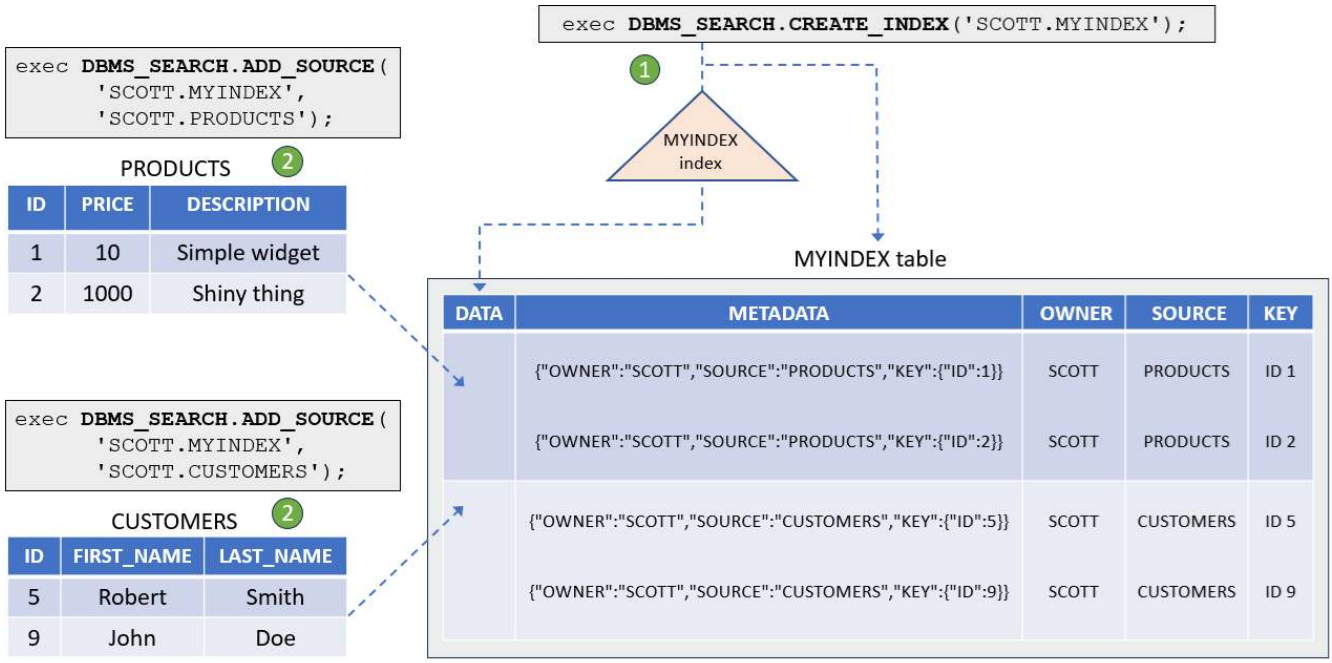
- **Query multiple objects**:

  The `DBMS_SEARCH.FIND` API allows you to retrieve a hitlist of all documents based on the specified filter conditions.

This index creates background jobs at predefined intervals to synchronize the DML changes and optimize the index using the `AUTO_DAILY` mode on all data sources. You do not need to explicitly run the `SYNC_INDEX` and `OPTIMIZE_INDEX` operations on this index.

Ubiquitous Search Index Creation Overview

You create a `DBMS_SEARCH` index by simply specifying an index name and then adding various data sources to it. This is illustrated in the following diagram:



1. The first command (`DBMS_SEARCH.CREATE_INDEX` procedure) creates an index table as *`[schema].index_name`*. A ubiquitous search index, also named *`[schema].index_name`*, is created on the `DATA` column of the index table. Note that the index table name matches your index name.

   Here, the schema owner name (`SCOTT`) is specified along with the index name (`MYINDEX`) as `SCOTT.MYINDEX`.

2. The second command (`DBMS_SEARCH.ADD_SOURCE` procedure) adds one or more data sources such as tables, views, or duality views from different schemas.

Here, this procedure combines contents from all the columns of the PRODUCTS and CUSTOMERS tables in Scott's schema into the MYINDEX table.

The MYINDEX table contains the following columns:

- **DATA** (JSON datatype):

  This is an empty column, and is a placeholder for querying the DBMS_SEARCH index. You add your data sources into the DATA column. You can then run PL/SQL queries against this DATA column using the CONTAINS(), JSON_TEXTCONTAINS(), and JSON_EXISTS operators.

  The DATA column creates a JSON representation of the following form for each indexed row of the table or view that is added as a data source to this index:

```
{"OWNER":
  {
   "TABLE_NAME":{"COLUMN1_NAME":"COLUMN1_VALUE",…}
  }
}
```

  Note that the DATA column does not store actual data. Instead, the data resides in the original base tables. This index references your data source tables to create a virtual indexed JSON document on the fly. After the data is fetched and indexed, this column is effectively emptied to avoid duplication.

- **METADATA** (JSON datatype):

  The METADATA column helps the DBMS_SEARCH index to uniquely identify each row of the table or view that is indexed. After adding data sources to this index, you can see that the METADATA column stores a JSON representation of the following form for each indexed row of your data source:

```
{
 "OWNER"  : "Table_Owner or View_Owner",
 "SOURCE" : "Table_Name or View_Name",
 "KEY"    : "{PrimaryKey_COLUMN_i" : PrimaryKey_VALUE_i}
}
```

  OWNER specifies the owner of the table or view added as a data source into this index.

  SOURCE specifies the table name or view name of the data source.

  KEY is composed of all the primary key columns of the data source table. If the table does not have a primary key, then a ROWID is used instead. However, Oracle strongly recommends defining a primary key.

  As the diagram illustrates, the METADATA column stores corresponding JSON entries of the following form for each indexed row:

  For the PRODUCTS table:

```
{"OWNER":"SCOTT","SOURCE":"PRODUCTS","KEY":{"ID":1}}
{"OWNER":"SCOTT","SOURCE":"PRODUCTS","KEY":{"ID":2}}
```

For the CUSTOMERS table:

```
{"OWNER":"SCOTT","SOURCE":"CUSTOMERS","KEY":{"ID":5}}
{"OWNER":"SCOTT","SOURCE":"CUSTOMERS","KEY":{"ID":9}}
```

> **Note:**
>
> The DBMS_SEARCH index stores all supported SQL data types (including Object Type columns) as JSON objects, except for the XMLTYPE and LONG data types. Therefore, you cannot add a table or view as data source if it has a column with the XMLTYPE or LONG data type.

- **OWNER, SOURCE, KEY** (VARCHAR2 datatype):

  Each JSON key of the METADATA column, that is, OWNER, SOURCE, and KEY, is also a separate virtual column in the MYINDEX table.

  Note that the MYINDEX table is partitioned by OWNER and SOURCE. When querying a particular data source, you can add a WHERE clause condition on the OWNER and SOURCE virtual columns to restrict your query search to a specific partition of that source using partition pruning.

> **Note:**
>
> All the data sources (such as table, view, or each table in the view definition) that are added to the DBMS_SEARCH index must include at least one Primary Key column. Each table that is part of a view source having a foreign key must also have the Foreign Key constraint, referencing the relevant primary keys defined on the table. If the source table does not have a primary key, then a ROWID is used instead.

Query Indexed Data

As discussed earlier, you can use the DBMS_SEARCH.GET_DOCUMENT procedure to view all the contents extracted from the original base tables by querying a **virtual document**. This document contains a JSON representation for each indexed row of a table or view that is added as data source to your index.

The syntax for DBMS_SEARCH.GET_DOCUMENT is:

```
SELECT DBMS_SEARCH.GET_DOCUMENT('[schema].index_name', METADATA)
 from [schema].index_name;
```

For example, using our earlier PRODUCTS and CUSTOMERS source tables scenario, the following statement returns a virtual document with combined metadata values as indexed in the MYINDEX index:

```
SELECT DBMS_SEARCH.GET_DOCUMENT('SCOTT.MYINDEX',METADATA)
  from SCOTT.MYINDEX;

DBMS_SEARCH.GET_DOCUMENT('SCOTT.MYINDEX', METADATA)
--------------------------------------------------------------------
```

```
{
  "SCOTT" :
  {
    "PRODUCTS" :
    {
      "ID"          : 1,
      "PRICE"       : 10,
      "DESCRIPTION" : "simple widget"
    }
  }
}
{
  "SCOTT" :
  {
    "PRODUCTS" :
    {
      "ID"          : 2,
      "PRICE"       : 2000,
      "DESCRIPTION" : "shiny thing"
    }
  }
}
{
  "SCOTT" :
  {
    "CUSTOMERS" :
    {
      "ID"          : 5,
      "FIRSTNAME"   : "Robert",
      "LASTNAME"    : "Smith"
    }
  }
}
{
  "SCOTT" :
  {
    "CUSTOMERS" :
    {
      "ID"          : 9,
      "FIRSTNAME"   : "John",
      "LASTNAME"    : "Doe"
    }
  }
}
```

You can now run queries against your index using the `CONTAINS`, `JSON_TEXTCONTAINS`, and `JSON_EXISTS` operators.

DBMS_SEARCH Dictionary Views

You can use the following dictionary views to examine your ubiquitous search indexes:

- `USER_DBMS_SEARCH_INDEXES`: To query information about the indexes that are created in a user's schema.

- `ALL_DBMS_SEARCH_INDEXES`: To query information about all existing indexes, corresponding to each index owner.
- `USER_DBMS_SEARCH_INDEX_SOURCES`: To query information about the data sources that are added to indexes, created in a user's schema.
- `ALL_DBMS_SEARCH_INDEX_SOURCES`: To query information about all existing data sources added to indexes, corresponding to each index owner.

Related Topics

- Perform Ubiquitous Search: End-to-End Examples
  Learn how to create `DBMS_SEARCH` indexes for performing various ubiquitous search use cases by running these end-to-end example scenarios.
- DBMS_SEARCH Package
- Oracle Text Views

13.2 Perform Ubiquitous Search: End-to-End Examples

Learn how to create `DBMS_SEARCH` indexes for performing various ubiquitous search use cases by running these end-to-end example scenarios.

- Create and Query DBMS_SEARCH Indexes Using Multiple Tables and Views
- Use JSON Duality Views with DBMS_SEARCH Indexes
- Examine DBMS_SEARCH Indexes Using Dictionary Views

13.2.1 Create and Query DBMS_SEARCH Indexes Using Multiple Tables and Views

In this example, you can see how to create a ubiquitous search index, add multiple tables and views to it, and then query against the index using the `CONTAINS`, `JSON_TEXTCONTAINS`, and `JSON_EXISTS` operators.

1. Connect to Oracle Database as a local user.
   a. Log in to SQL*Plus as the `SYS` user, connecting as `SYSDBA`:

```
conn sys/password as sysdba


CREATE TABLESPACE tbs1
DATAFILE 'tbs5.dbf' SIZE 20G AUTOEXTEND ON
EXTENT MANAGEMENT LOCAL
SEGMENT SPACE MANAGEMENT AUTO;


SET ECHO ON
SET FEEDBACK 1
SET NUMWIDTH 10
SET LINESIZE 80
SET TRIMSPOOL ON
SET TAB OFF
SET PAGESIZE 10000
SET LONG 10000
```

b. Create a local user (`docuser`) and grant necessary privileges:

```
DROP USER docuser cascade;


GRANT DB_DEVELOPER_ROLE, DEFAULT TABLESPACE tbs1 quota unlimited on tbs1
TO docuser IDENTIFIED BY password;
```

c. Connect as the local user (`docuser`):

```
CONN docuser/password
```

2. Create and populate the `customers`, `items`, `orders`, and `lineitems` tables. You will later add these tables to your ubiquitous search index.

a. `customers`:

Here, the schema owner name `DOCUSER` is explicitly specified.

```
CREATE TABLE docuser.customers (
    cust_id number PRIMARY KEY,
    first varchar2(30),
    last varchar2(30));


insert into customers values (1, 'Robert', 'Smith');
insert into customers values (2, 'John', 'Doe');
insert into customers values (3, 'James', 'Martin');
insert into customers values (5, 'Jane', 'Y');
```

b. `items`:

```
CREATE TABLE items (
    item_id number PRIMARY KEY,
    name    varchar2(30),
    price   number(5,2),
    stock_quantity number);


insert into items values (122, 'Potato Gun', 29.99, 10);
insert into items values (232, 'Rubber Christmas Tree', 65.00, 0);
insert into items values (345, 'Border Patrol Costume', 19.99, 20);
insert into items values (845, 'Meteor Impact Survival Kit', 299.00, 0);
insert into items values (429, 'Air Guitar', 9.99, 14);
```

c. `orders`:

```
CREATE TABLE orders (
    order_id number PRIMARY KEY,
    cust_id number REFERENCES customers(cust_id) ON DELETE CASCADE);


insert into orders values (1, 1);
insert into orders values (2, 1);
insert into orders values (3, 3);
insert into orders values (4, 2);
```

d. lineitems:

```
CREATE TABLE lineitems (
    order_id number REFERENCES orders(order_id) ON DELETE CASCADE,
    item_id  number REFERENCES items(item_id) ON DELETE CASCADE,
    quantity number,
    PRIMARY KEY(order_id, item_id));


insert into lineitems values(1, 845, 1);
insert into lineitems values(2, 232, 1);
insert into lineitems values(2, 429, 4);
insert into lineitems values(3, 122, 1);
insert into lineitems values(4, 345, 1);
```

3. Create a view named `search_view` based on the tables you created. You will later add this view to your ubiquitous search index.

```
CREATE OR REPLACE VIEW search_view(cust_id, business_object,
  CONSTRAINT search_view_pk
    PRIMARY KEY(cust_id)
      RELY DISABLE NOVALIDATE,
  CONSTRAINT search_view_fk
    FOREIGN KEY(cust_id) REFERENCES customers(cust_id)
      DISABLE NOVALIDATE) AS
SELECT c.cust_id, JSON_OBJECT(
          'id' VALUE c.cust_id,
          'name' VALUE (c.first || ' ' || c.last),
          'num_orders' VALUE (
               SELECT COUNT(*)
               FROM orders o
               WHERE o.cust_id = c.cust_id),
          'orders' VALUE (
               SELECT JSON_ARRAYAGG(
                    JSON_OBJECT(
                         'order_id' VALUE o.order_id,
                         'items' VALUE (
                              SELECT JSON_ARRAYAGG (
                                   JSON_OBJECT(
                                        'id' VALUE l.item_id,
                                        'name' VALUE i.name,
                                        'quantity' VALUE l.quantity,
                                        'single_item_price' VALUE i.price,
                                        'total_price' VALUE (i.price *
l.quantity)))
                                FROM lineitems l, items i
                                WHERE l.order_id = o.order_id
                                AND i.item_id = l.item_id)))
          FROM orders o
          WHERE o.cust_id = c.cust_id) ABSENT ON NULL) business_object
FROM customers c;
```

4. Create a ubiquitous search index named `MY_SEARCH_INDEX`.

```
EXEC DBMS_SEARCH.CREATE_INDEX('DOCUSER.MY_SEARCH_INDEX', NULL, 'JSON');
```

Note that you can omit `tablespace` and `datatype` (defaults to `JSON`), as follows:

```
exec dbms_search.create_index('MY_SEARCH_INDEX');
```

Run the following command to determine the structure of your index, which is created in the `DOCUSER` schema:

```
DESC MY_SEARCH_INDEX;
```

```
 Name             Null?      Type
 ---------------- -------- ----------------------------
 METADATA         NOT NULL   JSON
 DATA                        JSON
 OWNER                       VARCHAR2(128)
 SOURCE                      VARCHAR2(128)
 KEY                         VARCHAR2(1024)
```

5. Add the `CUSTOMERS` table as data source to `MY_SEARCH_INDEX`.

```
EXEC DBMS_SEARCH.ADD_SOURCE('DOCUSER.MY_SEARCH_INDEX',
'DOCUSER.CUSTOMERS');
```

You can add all other table sources to this index, but it is not required to complete this example scenario.

6. Examine the `DATA` and `METADATA` columns of your index along with the `DBMS_SEARCH` dictionary views, as shown in the following steps:

   a. Query the `METADATA` column:

```
SELECT JSON_SERIALIZE(METADATA FORMAT JSON) META
FROM DOCUSER.MY_SEARCH_INDEX
ORDER BY META;
```

   The `METADATA` column helps the `DBMS_SEARCH` index to uniquely identify each row of the table or view that is indexed. You can see that the `METADATA` column stores a JSON representation of the following form for each indexed row of your `customers` table:

```
META
--------------------------------------------------------------------------------
--------
{"OWNER":"DOCUSER","SOURCE":"CUSTOMERS","KEY":{"CUST_ID":1}}
{"OWNER":"DOCUSER","SOURCE":"CUSTOMERS","KEY":{"CUST_ID":2}}
{"OWNER":"DOCUSER","SOURCE":"CUSTOMERS","KEY":{"CUST_ID":3}}
{"OWNER":"DOCUSER","SOURCE":"CUSTOMERS","KEY":{"CUST_ID":5}}

4 rows selected.
```

   b. Query the `DATA` column:

```
SELECT DATA
FROM DOCUSER.MY_SEARCH_INDEX;
```

Note that the `DATA` column does not store actual data. Instead, the data resides in the original base tables. This index references your data source tables to create a virtual indexed JSON document on the fly. After the data is fetched and indexed, this column is effectively emptied to avoid duplication.

```
DATA
--------------------------------------------------------------------------
--------


4 rows selected.
```

c. Get a virtual indexed document to examine the contents that are extracted from the `customers` table source.

```
SELECT
JSON_SERIALIZE(
  DBMS_SEARCH.GET_DOCUMENT('DOCUSER.MY_SEARCH_INDEX', METADATA) FORMAT
JSON) DOC
FROM DOCUSER.MY_SEARCH_INDEX
ORDER BY JSON_SERIALIZE(METADATA FORMAT JSON);
```

This document contains a JSON representation for each indexed row of the `customers` table that is added as data source to your index:

```
DOC
--------------------------------------------------------------------------
--------
{"DOCUSER":{"CUSTOMERS":{"CUST_ID":1,"FIRST":"Robert","LAST":"Smith"}}}
{"DOCUSER":{"CUSTOMERS":{"CUST_ID":2,"FIRST":"John","LAST":"Doe"}}}
{"DOCUSER":{"CUSTOMERS":{"CUST_ID":3,"FIRST":"James","LAST":"Martin"}}}
{"DOCUSER":{"CUSTOMERS":{"CUST_ID":5,"FIRST":"Jane","LAST":"Y"}}}

4 rows selected.
```

d. Query the `USER_DBMS_SEARCH_INDEXES` view to display metadata values for the index.

```
SELECT IDX_NAME FROM USER_DBMS_SEARCH_INDEXES ORDER BY IDX_NAME;
```

This view shows the index name added in your user schema, `DOCUSER`:

```
IDX_NAME
-------------------------------------------------
MY_SEARCH_INDEX

1 row selected.
```

e. Query the `USER_DBMS_SEARCH_INDEX_SOURCES` view to display metadata values for your data source.

```
SELECT IDX_NAME,SRC_OWNER,SRC_NAME,SRC_TYPE
FROM USER_DBMS_SEARCH_INDEX_SOURCES
ORDER BY IDX_NAME, SRC_OWNER,SRC_NAME;
```

This view shows the data source details associated with your index, from your user schema (DOCUSER). Here, the source type T implies a "table" source:

```
IDX_NAME
------------------------------------------------
SRC_OWNER
------------------------------------------------
SRC_NAME
------------------------------------------------
SRC_TYPE
-
MY_SEARCH_INDEX
DOCUSER
CUSTOMERS
T

1 row selected.
```

7. Add a view to your index, then examine the METADATA column and dictionary views again to compare how the changes are reflected in the indexed data.

   a. Add the view that you created (SEARCH_VIEW) as a data source to the index:

   ```
   EXEC DBMS_SEARCH.ADD_SOURCE('DOCUSER.MY_SEARCH_INDEX',
   'DOCUSER.SEARCH_VIEW');
   ```

   b. Query the METADATA column:

   ```
   SELECT JSON_SERIALIZE(METADATA FORMAT JSON) META
   FROM DOCUSER.MY_SEARCH_INDEX
   ORDER BY META;
   ```

   The METADATA column additionally shows each row of the view that is indexed:

   ```
   META
   ------------------------------------------------------------------------
   --------
   {"OWNER":"DOCUSER","SOURCE":"CUSTOMERS","KEY":{"CUST_ID":1}}
   {"OWNER":"DOCUSER","SOURCE":"CUSTOMERS","KEY":{"CUST_ID":2}}
   {"OWNER":"DOCUSER","SOURCE":"CUSTOMERS","KEY":{"CUST_ID":3}}
   {"OWNER":"DOCUSER","SOURCE":"CUSTOMERS","KEY":{"CUST_ID":5}}
   {"OWNER":"DOCUSER","SOURCE":"SEARCH_VIEW","KEY":{"CUST_ID":1}}
   {"OWNER":"CUSTOMER","SOURCE":"SEARCH_VIEW","KEY":{"CUST_ID":2}}
   {"OWNER":"DOCUSER","SOURCE":"SEARCH_VIEW","KEY":{"CUST_ID":3}}
   {"OWNER":"DOCUSER","SOURCE":"SEARCH_VIEW","KEY":{"CUST_ID":5}}

   8 rows selected.
   ```

   c. Query the USER_DBMS_SEARCH_INDEXES dictionary view.

   ```
   SELECT IDX_NAME FROM USER_DBMS_SEARCH_INDEXES
     ORDER BY IDX_NAME;
   ```

This view shows the index name present in your user schema, DOCUSER:

```
IDX_NAME
--------------------------------------------------------------------------------
--------
MY_SEARCH_INDEX

1 row selected.
```

d. Query the USER_DBMS_SEARCH_INDEX_SOURCES dictionary view.

```
SELECT IDX_NAME,SRC_OWNER,SRC_NAME,SRC_TYPE
FROM USER_DBMS_SEARCH_INDEX_SOURCES
ORDER BY IDX_NAME, SRC_OWNER,SRC_NAME;
```

The output shows an additional row for SEARCH_VIEW, added as a data source to your index. Here, the source types T and V imply "table" and "view" sources, respectively:

```
IDX_NAME
--------------------------------------------------
SRC_OWNER
--------------------------------------------------
SRC_NAME
--------------------------------------------------
SRC_TYPE
-
MY_SEARCH_INDEX
DOCUSER
CUSTOMERS
T

MY_SEARCH_INDEX
DOCUSER
SEARCH_VIEW
V

2 rows selected.
```

8. Run queries against your index using the JSON_EXISTS operator.

a. Search for documents in the view source of your index, where the DATA column contains a JSON element $.DOCUSER.SEARCH_VIEW.

```
SELECT JSON_SERIALIZE(METADATA FORMAT JSON) META
FROM DOCUSER.MY_SEARCH_INDEX
WHERE JSON_EXISTS(DATA,'$.DOCUSER.SEARCH_VIEW')
ORDER BY META;
```

The output returns four rows from the DOCUSER schema with combined customer IDs as 1, 2, 3, and 5:

```
META
--------------------------------------------------------------------------------
--------
```

```
{"OWNER":"DOCUSER","SOURCE":"SEARCH_VIEW","KEY":{"CUST_ID":1}}
{"OWNER":"DOCUSER","SOURCE":"SEARCH_VIEW","KEY":{"CUST_ID":2}}
{"OWNER":"DOCUSER","SOURCE":"SEARCH_VIEW","KEY":{"CUST_ID":3}}
{"OWNER":"DOCUSER","SOURCE":"SEARCH_VIEW","KEY":{"CUST_ID":5}}

4 rows selected.
```

b. Search for documents in the view source of you index as a virtual document, where the `DATA` column contains a JSON element `$.DOCUSER.SEARCH_VIEW`:

This is a similar query as shown in the previous step. However, here you can view an entire virtual indexed document with a JSON representation of all the metadata values:

```
SELECT
JSON_SERIALIZE(
  DBMS_SEARCH.GET_DOCUMENT('DOCUSER.MY_SEARCH_INDEX', METADATA) FORMAT
JSON) DOC
FROM DOCUSER.MY_SEARCH_INDEX
WHERE JSON_EXISTS(DATA,'$.DOCUSER.SEARCH_VIEW')
ORDER BY JSON_SERIALIZE(METADATA FORMAT JSON);
```

The output returns a JSON document with combined metadata values, as indexed in `MY_SEARCH_INDEX`:

```
DOC
--------------------------------------------------------------------------
--------
{"DOCUSER":{"SEARCH_VIEW":{"CUST_ID":1,"BUSINESS_OBJECT":
{"id":1,"name":"Robert
Smith","num_orders":2,"orders":[{"order_id":1,"items":
[{"id":845,"name":"Meteor
 Impact Survival
Kit","quantity":1,"single_item_price":299,"total_price":299}]},
{"order_id":2,"items":[{"id":232,"name":"Rubber Christmas
Tree","quantity":1,
"single_item_price":65,"total_price":65},{"id":429,"name":"Air
Guitar","quantity":4
,"single_item_price":9.99,"total_price":39.96}]}]}}}}

{"DOCUSER":{"SEARCH_VIEW":{"CUST_ID":2,"BUSINESS_OBJECT":
{"id":2,"name":"John
Doe","num_orders":1,"orders":[{"order_id":4,"items":
[{"id":345,"name":"Border
Patrol
Costume","quantity":1,"single_item_price":19.99,"total_price":19.99}]
}]}}}}

{"DOCUSER":{"SEARCH_VIEW":{"CUST_ID":3,"BUSINESS_OBJECT":
{"id":3,"name":"James
Martin","num_orders":1,"orders":[{"order_id":3,"items":
[{"id":122,"name":"Potato
Gun","quantity":1,"single_item_price":29.99,"total_price":29.99}]}]}}}}

{"DOCUSER":{"SEARCH_VIEW":{"CUST_ID":5,"BUSINESS_OBJECT":
{"id":5,"name":"Jane Y",
"num_orders":0}}}}
```

```
4 rows selected.
```

9. Perform a textual search query on targeted paths using the `JSON_TEXTCONTAINS` operator.

   a. Query the `$.DOCUSER.SEARCH_VIEW.BUSINESS_OBJECT.name` JSON path in the `DATA` column for the keywords "`Anon or Jane`".

```
SELECT
JSON_SERIALIZE(
  DBMS_SEARCH.GET_DOCUMENT('DOCUSER.MY_SEARCH_INDEX', METADATA) FORMAT
JSON) DOC
FROM DOCUSER.MY_SEARCH_INDEX
WHERE
JSON_TEXTCONTAINS(DATA,'$.DOCUSER.SEARCH_VIEW.BUSINESS_OBJECT.name',
                       'Anon or Jane')
ORDER BY JSON_SERIALIZE(METADATA FORMAT JSON);
```

   The output returns a JSON document with the customer ID as `5` and the name as `Jane Y`, from the `SEARCH_VIEW` source in the `DOCUSER` schema:

```
DOC
--------------------------------------------------------------------------
---------
{"DOCUSER":{"SEARCH_VIEW":{"CUST_ID":5,"BUSINESS_OBJECT":
{"id":5,"name":"Jane Y",
"num_orders":0}}}}

1 row selected.
```

   b. Use the `SCORE` operator with `JSON_TEXTCONTAINS` to obtain a relevance score for your search result.

```
SELECT METADATA, score(1) from DOCUSER.MY_SEARCH_INDEX
    WHERE JSON_TEXTCONTAINS(
        DATA,'$.DOCUSER.SEARCH_VIEW.BUSINESS_OBJECT.name','Anon or
Jane',1);
```

   The output returns metadata values and a relevance score of 5 for the matching record of customer ID `5`, Jane Y.

```
 METADATA
--------------------------------------------------------------------------
--------
  SCORE(1)
----------
{"OWNER":"DOCUSER","SOURCE":"SEARCH_VIEW","KEY":{"CUST_ID":5}}
         5

1 row selected.
```

10. Search across the entire schema using the `CONTAINS` operator.

a. Query the index to retrieve records that match the keywords "`Anon or Jane`".

```
SELECT
JSON_SERIALIZE(
  DBMS_SEARCH.GET_DOCUMENT('DOCUSER.MY_SEARCH_INDEX', METADATA) FORMAT
JSON) DOC
FROM DOCUSER.MY_SEARCH_INDEX
WHERE
CONTAINS(DATA, 'Anon or Jane') > 0
ORDER BY JSON_SERIALIZE(METADATA FORMAT JSON);
```

The output returns two JSON objects from the `DOCUSER` schema. One with the customer ID as `5` and the name as `Jane Y`, from the `CUSTOMERS` table source. Another also with the customer ID as `5` and the name as `Jane Y`, but from the `SEARCH_VIEW` view source.

Note that the business object here has zero orders associated with it.

```
DOC
--------------------------------------------------------------------------------
--------
{"DOCUSER":{"CUSTOMERS":{"CUST_ID":5,"FIRST":"Jane","LAST":"Y"}}}
{"DOCUSER":{"SEARCH_VIEW":{"CUST_ID":5,"BUSINESS_OBJECT":
{"id":5,"name":"Jane Y",
 "num_orders":0}}}}

2 rows selected.
```

b. Use the `SCORE` operator with `CONTAINS` to obtain a relevance score for your search result.

```
SELECT METADATA, score(1) as search_score
    from DOCUSER.MY_SEARCH_INDEX
    WHERE CONTAINS(DATA, 'Anon or Jane',1)>0;
```

Here, the output returns the matching records of customer ID `5`, Jane Y, from both the table source and view source. It also shows a search score of 5 for both the records.

```
METADATA
--------------------------------------------------------------------------------
--------
SEARCH_SCORE
------------
{"OWNER":"DOCUSER","SOURCE":"CUSTOMERS","KEY":{"CUST_ID":5}}
           5

{"OWNER":"DOCUSER","SOURCE":"SEARCH_VIEW","KEY":{"CUST_ID":5}}
           5

2 rows selected.
```

11. Use the `DBMS_SEARCH.FIND` procedure to retrieve a hitlist. This also facets an aggregations of JSON documents based on the specified query-by-example (QBE) filter conditions.

```
SELECT JSON_SERIALIZE(DBMS_SEARCH.FIND('DOCUSER.MY_SEARCH_INDEX',
JSON('{
  "$query" : { "DOCUSER.SEARCH_VIEW.*" : { "$contains" : "Gun or patrol
costume" }  },
  "$facet" : [
    { "$sum" :
"DOCUSER.SEARCH_VIEW.BUSINESS_OBJECT.orders.items.total_price"},
    { "$count" :
      { "path" :
"DOCUSER.SEARCH_VIEW.BUSINESS_OBJECT.orders.items.single_item_price",
        "bucket" : [ { "$lt" : 20  }, { "$gte" : 20 } ] } },
    { "$uniqueCount" : "DOCUSER.SEARCH_VIEW.BUSINESS_OBJECT.name"  }
  ]
}')) FORMAT JSON PRETTY) AGG FROM DUAL;
```

The output shows an aggregation result in JSON format.

Here, the query searches for the phrase `Gun or patrol costume` within `DOCUSER.SEARCH_VIEW`. The `$count` indicates two records that match the query criteria. The `$facet` key groups multiple aggregation results into separate buckets, performing the following aggregations:

- `DOCUSER.SEARCH_VIEW.BUSINESS_OBJECT.orders.items.total_price`:

  Aggregates the total price of all items in the `orders.items.total_price` field into a total of `49.98`.

- `DOCUSER.SEARCH_VIEW.BUSINESS_OBJECT.orders.items.single_item_price`:

  Aggregates the prices of individual items in the `single_item_price` field and groups them into price ranges (buckets).

- `DOCUSER.SEARCH_VIEW.BUSINESS_OBJECT.name`:

  Counts the unique occurrences of names, `John Doe` and `James Martin`, in the `BUSINESS_OBJECT.name` field. Both the names appear once.

```
AGG
--------------------------------------------------------------------------------
-----
{
  "$count" : 2,
  "$facet" :
  [
    {
      "DOCUSER.SEARCH_VIEW.BUSINESS_OBJECT.orders.items.total_price" :
      {
        "$sum" : 49.98
      }
    },
    {

"DOCUSER.SEARCH_VIEW.BUSINESS_OBJECT.orders.items.single_item_price" :
      [
        {
```

```
            "bucket" :
            {
              "$gte" : 19.99,
              "$lt" : 20
            },
            "$count" : 1
          },
          {
            "bucket" :
            {
              "$gte" : 20,
              "$lte" : 29.99
            },
            "$count" : 1
          }
        ]
      },
      {
        "DOCUSER.SEARCH_VIEW.BUSINESS_OBJECT.name" :
        [
          {
            "value" : "John Doe",
            "$uniqueCount" : 1
          },
          {
            "value" : "James Martin",
            "$uniqueCount" : 1
          }
        ]
      }
    ]
}

1 row selected.
```

Related Topics

- DBMS_SEARCH Package
- SCORE
- CONTAINS
- JSON_TEXTCONTAINS
- JSON_EXISTS

13.2.2 Use JSON Duality Views with DBMS_SEARCH Indexes

In this example, you can see how to create a ubiquitous search index, define and add a JSON duality view to it, and then query against the index using the `CONTAINS`, `JSON_TEXTCONTAINS`, and `JSON_EXISTS` operators.

Note that while querying your data, you might need to wait for the synchronization operation to complete depending on your specified `SYNC` setting.

1. Connect to Oracle Database as a local user.

a. Log in to SQL*Plus as the `SYS` user, connecting as `SYSDBA`:

```
conn sys/password as sysdba


CREATE TABLESPACE tbs1
DATAFILE 'tbs5.dbf' SIZE 20G AUTOEXTEND ON
EXTENT MANAGEMENT LOCAL
SEGMENT SPACE MANAGEMENT AUTO;


SET ECHO ON
SET FEEDBACK 1
SET NUMWIDTH 10
SET LINESIZE 80
SET TRIMSPOOL ON
SET TAB OFF
SET PAGESIZE 10000
SET LONG 10000
```

b. Create a local user (`docuser`) and grant necessary privileges:

```
DROP USER docuser cascade;


GRANT DB_DEVELOPER_ROLE, DEFAULT TABLESPACE tbs1 quota unlimited on tbs1
TO docuser IDENTIFIED BY password;
```

c. Connect as the local user (`docuser`):

```
CONN docuser/password
```

2. Prepare tables to implement your JSON duality view.

a. Create the `employees` and `departments` tables:

```
CREATE TABLE employees
        ( employee_id    NUMBER(6) primary key,
          first_name     varchar2(4000),
          last_name      varchar2(4000),
          department_id  NUMBER(4)
        );


CREATE TABLE departments
        ( department_id    NUMBER(5) primary key,
          department_name  VARCHAR2(30),
          manager_id       NUMBER(6)
        );
```

b. Alter the `employees` table to include a foreign key constraint (`emp_dept_fkey`) to it.

Here, you specify that the `department_id` column in the `employees` table uses a foreign key to the `department_id` column in the `departments` table:

```
ALTER TABLE employees
  ADD (
       CONSTRAINT  emp_dept_fkey
       FOREIGN KEY (department_id)
       REFERENCES departments
);
```

c. Populate the `employees` and `departments` tables:

```
INSERT INTO departments VALUES(10, 'Administration', 100);
INSERT INTO employees VALUES(100, 'Robert', 'Smith', 10);
INSERT INTO employees VALUES(101, 'James', 'Martin' ,10);
INSERT INTO employees VALUES(102, 'John', 'Doe', 10);

commit;
```

3. Create a `DBMS_SEARCH` index named `MY_SEARCH_INDEX`.

```
exec dbms_search.create_index('MY_SEARCH_INDEX',NULL,'JSON');
```

Note that you can omit `tablespace` and `datatype` (defaults to `JSON`), as follows:

```
exec dbms_search.create_index('MY_SEARCH_INDEX');
```

Run the following command to determine the structure of your index, which is created in the `DOCUSER` schema:

```
DESC MY_SEARCH_INDEX;
```

```
 Name              Null?      Type
 ----------------- -------- ----------------------------
 METADATA          NOT NULL   JSON
 DATA                         JSON
 OWNER                        VARCHAR2(128)
 SOURCE                       VARCHAR2(128)
 KEY                          VARCHAR2(1024)
```

4. Define a JSON duality view over the tables you created, and then add that view to `MY_SEARCH_INDEX`.

a. Create an employee-centric JSON duality view named `MY_EMP_VIEW`.

```
CREATE or replace JSON relational duality VIEW MY_EMP_VIEW
      AS
    select JSON {
    'EMPLOYEE_ID' is emp.EMPLOYEE_ID,
    'FIRST_NAME'  is emp.FIRST_NAME,
    'LAST_NAME' is emp.last_name,
    'department_info' is
    (
```

```
      select JSON
      {
      'DEPARTMENT_ID' is dept.department_id ,
      'departmentname' is dept.department_name WITH(UPDATE)
      }
      from departments dept WITH(UPDATE,CHECK ETAG)
      where dept.department_id = emp.department_id
    )
}
from employees emp WITH(INSERT,UPDATE,DELETE);
```

b. Add the duality view (`MY_EMP_VIEW`) to `MY_SEARCH_INDEX` as a data source:

```
exec dbms_search.add_source('MY_SEARCH_INDEX','MY_EMP_VIEW');
```

5. Examine what is indexed in `MY_SEARCH_INDEX`:

a. Query the `METADATA` column to review the source information from where the tables are extracted.

```
select JSON_SERIALIZE(METADATA FORMAT JSON) META from MY_SEARCH_INDEX
order by owner,source,key;
```

The index table's `METADATA` column stores a JSON representation of the following form for each indexed row:

```
META
--------------------------------------------------------------------------------
--------
{"OWNER":"DOCUSER","SOURCE":"MY_EMP_VIEW","KEY":{"EMPLOYEE_ID":100}}
{"OWNER":"DOCUSER","SOURCE":"MY_EMP_VIEW","KEY":{"EMPLOYEE_ID":101}}
{"OWNER":"DOCUSER","SOURCE":"MY_EMP_VIEW","KEY":{"EMPLOYEE_ID":102}}
```

b. Examine the metadata values as indexed in the `DATA` column.

```
select data from MY_EMP_VIEW
order by 1;
```

```
DATA
--------------------------------------------------------------------------------
--------
{"_metadata":
{"etag":"77AACDD6860BE5D14FCEB2A8633336D7","asof":"0000000000000000
{"_metadata":
{"etag":"4B233BD8A51C3B905A357F8446FBABDA","asof":"0000000000000000
{"_metadata":
{"etag":"5669EAC90AB697DA6C13D2045D6C6638","asof":"0000000000000000
```

c. View a virtual indexed document to examine the actual fields that are indexed.

```
select JSON_SERIALIZE(
  DBMS_SEARCH.GET_DOCUMENT('MY_SEARCH_INDEX', METADATA) FORMAT JSON)
DOC
```

```
from MY_SEARCH_INDEX
order by owner,source,key;
```

The output returns a JSON document with combined metadata values, as indexed in `MY_SEARCH_INDEX`:

```
DOC
--------------------------------------------------------------------------------
{"DOCUSER":{"MY_EMP_VIEW":{"DATA":{"_metadata":
{"etag":"77AACDD6860BE5D14FCEB2A8633336D7",
"asof":"0000000000000000"},"EMPLOYEE_ID":100,"FIRST_NAME":"Robert","LAST
_NAME":"Smith",
"department_info":
{"DEPARTMENT_ID":10,"departmentname":"Administration"}}}}}

{"DOCUSER":{"MY_EMP_VIEW":{"DATA":{"_metadata":
{"etag":"4B233BD8A51C3B905A357F8446FBABDA",
"asof":"0000000000000000"},"EMPLOYEE_ID":101,"FIRST_NAME":"James","LAST_
NAME":"Martin",
"department_info":
{"DEPARTMENT_ID":10,"departmentname":"Administration"}}}}}

{"DOCUSER":{"MY_EMP_VIEW":{"DATA":{"_metadata":
{"etag":"5669EAC90AB697DA6C13D2045D6C6638",
"asof":"0000000000000000"},"EMPLOYEE_ID":102,"FIRST_NAME":"John","LAST_N
AME":"Doe",
"department_info":
{"DEPARTMENT_ID":10,"departmentname":"Administration"}}}}}
```

6. Query your duality view using the `JSON_EXISTS` operator.

   - This statement generates an execution plan that retrieves the `EMPLOYEE_ID` from the `MY_SEARCH_INDEX` table, filtering the JSON data for entries where the `FIRST_NAME` is `Robert` and the `LAST_NAME` is `Smith`:

     ```
     explain plan for
     select
     t.metadata.KEY."EMPLOYEE_ID".number() as employee_id
     from MY_SEARCH_INDEX t
     where
     json_exists(data,
       '$.DOCUSER.MY_EMP_VIEW.DATA?(@.FIRST_NAME == "Robert" && @.LAST_NAME
     == "Smith")');

     SELECT PLAN_TABLE_OUTPUT FROM
       TABLE(DBMS_XPLAN.DISPLAY(NULL,NULL,'BASIC PREDICATE PROJECTION'));
     ```

     The output appears as:

     ```
     PLAN_TABLE_OUTPUT
     --------------------------------------------------------------------------------
     --------
     Plan hash value: 1710711056
     ```

```
-------------------------------------------------------------
| Id  | Operation                          | Name            |
-------------------------------------------------------------
|   0 | SELECT STATEMENT                   |                 |
|   1 |  PARTITION LIST ALL                |                 |
|*  2 |   TABLE ACCESS BY LOCAL INDEX ROWID| MY_SEARCH_INDEX |
|*  3 |    DOMAIN INDEX                    | MY_SEARCH_INDEX |
-------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 -
filter(JSON_EXISTS2("DBMS_SEARCH"."GET_DOCUMENT"('"DOCUSER"."MY_SEARCH_I
NDEX"',
              "T"."METADATA" /*+ LOB_BY_VALUE */ ) FORMAT OSON,
              '$.DOCUSER.MY_EMP_VIEW.DATA?(@.FIRST_NAME == "Robert" &&
@.LAST_NAME ==
              "Smith")' /* json_path_str  $.DOCUSER.MY_EMP_VIEW.DATA?
((@.FIRST_NAME.string()
              == "Robert") && (@.LAST_NAME.string() == "Smith"))  */
FALSE ON ERROR TYPE(LAX) )=1)

   3 - access("CTXSYS"."CONTAINS"("T"."DATA" /*+ LOB_BY_VALUE */
              ,'(sdata(FVCH_DFE32BED91ED02414AB59BAEC23126D1_FIRST_NAME
 = "Robert" )
              and
sdata(FVCH_9A8FA0A86CCA96ADF45C533C7C92EFF7_LAST_NAME  = "Smith"
              ))')>0)

Column Projection Information (identified by operation id):
-----------------------------------------------------------

   1 - "T"."METADATA" /*+ LOB_BY_VALUE */ [JSON,8200]
   2 - "T"."METADATA" /*+ LOB_BY_VALUE */ [JSON,8200]
   3 - "T".ROWID[ROWID,10]

31 rows selected.
```

- This statement retrieves the primary key defined for `MY_EMP_VIEW` by searching the `EMPLOYEE_ID` field from the `MY_SEARCH_INDEX` table, where the JSON data in the data column contains `FIRST_NAME` as `Robert` and `LAST_NAME` as `Smith`.

```
select
t.metadata.KEY."EMPLOYEE_ID".number() as employee_id
from MY_SEARCH_INDEX t
where
json_exists(data,
  '$.DOCUSER.MY_EMP_VIEW.DATA?(@.FIRST_NAME == "Robert" && @.LAST_NAME
== "Smith")');
```

The output returns a row with `EMPLOYEE_ID` as `100`:

```
 EMPLOYEE_ID
-----------
 100
```

7. Update some fields in the duality view directly, and then query the index again to analyze the changes.

   a. Add a new name for an employee with Employee ID as `100`.

   ```
   update my_emp_view
   set data =
     '{"EMPLOYEE_ID":100,"FIRST_NAME":"new_name",
       "LAST_NAME":"new_lastname",
       "department_info":
   {"DEPARTMENT_ID":10,"departmentname":"Administration"}}'
     where json_value(data,'$.EMPLOYEE_ID') = 100;

   commit;
   ```

   b. View all contents extracted from the original base tables querying the `DATA` column of your index.

   ```
   select data from my_emp_view order by 1;
   ```

   The `DATA` column creates a JSON representation of the following form for each indexed row of `MY_EMP_VIEW` that is added as a data source to this index:

   ```
   DATA
   --------------------------------------------------------------------------------
   --------------
   {"_metadata":
   {"etag":"B7CAD96918892950C1FBA12E58AC198E","asof":"0000000000000000"},
   "EMPLOYEE_ID":100,"FIRST_NAME":"new_name","LAST_NAME":"new_lastname","de
   partment_info":
   {"DEPARTMENT_ID":10,"departmentname":"Administration"}}

   {"_metadata":
   {"etag":"4B233BD8A51C3B905A357F8446FBABDA","asof":"0000000000000000"},
   "EMPLOYEE_ID":101,"FIRST_NAME":"James","LAST_NAME":"Martin","department_
   info":
   {"DEPARTMENT_ID":10,"departmentname":"Administration"}}

   {"_metadata":
   {"etag":"5669EAC90AB697DA6C13D2045D6C6638","asof":"0000000000000000"},
   "EMPLOYEE_ID":102,"FIRST_NAME":"John","LAST_NAME":"Doe","department_info
   ":
   {"DEPARTMENT_ID":10,"departmentname":"Administration"}}

   {"_metadata":
   {"etag":"3F8A0577A0E8F8AAF38B088D1CD3EAE6","asof":"0000000000000000"},
   "EMPLOYEE_ID":103,"FIRST_NAME":"new_name2","LAST_NAME":"new_lastname2","
   department_info":
   {"DEPARTMENT_ID":10,"departmentname":"Administration"}}
   ```

```
4 rows selected.
```

c. Query the employee for whom you updated the new name:

```
select
t.metadata.KEY."EMPLOYEE_ID".number() as employee_id
from MY_SEARCH_INDEX t
where json_exists(data,
  '$.DOCUSER.MY_EMP_VIEW.DATA?(@.FIRST_NAME == "new_name" &&
@.LAST_NAME == "new_lastname")');
```

The output returns a row with `EMPLOYEE_ID` as `100`:

```
 EMPLOYEE_ID
-----------
 100

1 row selected.
```

d. Add a new employee to the `Administration` department.

```
insert into my_emp_view values('
  {"EMPLOYEE_ID":103,"FIRST_NAME":"new_name2",
   "LAST_NAME":"new_lastname2",
   "department_info":
{"DEPARTMENT_ID":10,"departmentname":"Administration"}}');
```

e. Add a new department named `HR`.

```
INSERT INTO departments VALUES
  ( 20, 'HR', 103);
```

f. Add a new employee to the `HR` department.

```
insert into  my_emp_view values('
  {"EMPLOYEE_ID":104,"FIRST_NAME":"new_name3",
   "LAST_NAME":"new_lastname3",
   "department_info":{"DEPARTMENT_ID":20,"departmentname":"HR"}}');

commit;
```

g. Query the `DATA` column of your index again to compare the updated indexed data.

```
select data from my_emp_view
  order by 1;


DATA
--------------------------------------------------------------------------------
--------
{"_metadata":
{"etag":"B7CAD96918892950C1FBA12E58AC198E","asof":"0000000000000000
```

```
"},"EMPLOYEE_ID":100,"FIRST_NAME":"new_name","LAST_NAME":"new_lastname",
"department_info":
{"DEPARTMENT_ID":10,"departmentname":"Administration"}}

{"_metadata":
{"etag":"4B233BD8A51C3B905A357F8446FBABDA","asof":"0000000000000000
"},"EMPLOYEE_ID":101,"FIRST_NAME":"James","LAST_NAME":"Martin",
"department_info":
{"DEPARTMENT_ID":10,"departmentname":"Administration"}}

{"_metadata":
{"etag":"5669EAC90AB697DA6C13D2045D6C6638","asof":"0000000000000000
"},"EMPLOYEE_ID":102,"FIRST_NAME":"John","LAST_NAME":"Doe",
"department_info":
{"DEPARTMENT_ID":10,"departmentname":"Administration"}}

{"_metadata":
{"etag":"3F8A0577A0E8F8AAF38B088D1CD3EAE6","asof":"0000000000000000
"},"EMPLOYEE_ID":103,"FIRST_NAME":"new_name2","LAST_NAME":"new_lastname2
",
"department_info":
{"DEPARTMENT_ID":10,"departmentname":"Administration"}}


4 rows selected.
```

View the `METADATA` column of your updated index.

```
select JSON_SERIALIZE(METADATA FORMAT JSON) META from MY_SEARCH_INDEX
  order by owner,source,key;
```

The `METADATA` column stores a JSON representation of the following form for each indexed row:

```
META
--------------------------------------------------------------------------------
--------
{"OWNER":"DOCUSER","SOURCE":"MY_EMP_VIEW","KEY":{"EMPLOYEE_ID":100}}
{"OWNER":"DOCUSER","SOURCE":"MY_EMP_VIEW","KEY":{"EMPLOYEE_ID":101}}
{"OWNER":"DOCUSER","SOURCE":"MY_EMP_VIEW","KEY":{"EMPLOYEE_ID":102}}
{"OWNER":"DOCUSER","SOURCE":"MY_EMP_VIEW","KEY":{"EMPLOYEE_ID":103}}
{"OWNER":"DOCUSER","SOURCE":"MY_EMP_VIEW","KEY":{"EMPLOYEE_ID":104}}
```

Compare the virtual document that returns a JSON document with combined metadata values, as indexed for each row.

```
select JSON_SERIALIZE(
  DBMS_SEARCH.GET_DOCUMENT('MY_SEARCH_INDEX', METADATA) FORMAT JSON) DOC
from MY_SEARCH_INDEX
order by owner,source,key;
```

8. Run queries against your index using the `JSON_TEXT_CONTAINS` operator.

a. Search for all recently updated employees whose names contain "new name":

```
select t.metadata.KEY."EMPLOYEE_ID".number() as employee_id
from MY_SEARCH_INDEX t
where
json_textcontains(data,'$.DOCUSER.MY_EMP_VIEW.DATA.FIRST_NAME','new
name%')
order by employee_id;
```

The output returns three rows with EMPLOYEE_IDs as 100, 103, and 104:

```
 EMPLOYEE_ID
-----------
  100
  103
  104
```

b. Search for an employee from the HR department:

```
select t.metadata.KEY."EMPLOYEE_ID".number() as employee_id
from MY_SEARCH_INDEX t
where json_exists(data,'$.DOCUSER.MY_EMP_VIEW.DATA?
(@.department_info.departmentname == "HR")');
```

The output returns a row with EMPLOYEE_ID as 104:

```
 EMPLOYEE_ID
-----------
  104
```

9. Run against your index across the schema, using the CONTAINS operator.

```
select JSON_SERIALIZE(
  DBMS_SEARCH.GET_DOCUMENT('MY_SEARCH_INDEX', METADATA) FORMAT JSON) DOC
from MY_SEARCH_INDEX
where contains(data,'Robert or HR')>0
order by owner,source,key;
```

The query retrieves a virtual indexed document with metadata values from MY_SEARCH_INDEX, where the DATA column contains keywords Robert and HR:

```
DOC
--------------------------------------------------------------------------------
----------------------
{"DOCUSER":{"MY_EMP_VIEW":{"DATA":{"_metadata":
{"etag":"7CB51D7BF53FD85174B3A1FC72EEECDB",
"asof":"0000000000000000"},"EMPLOYEE_ID":104,"FIRST_NAME":"new_name3","LAST
_NAME":"new_lastname3",
"department_info":{"DEPARTMENT_ID":20,"departmentname":"HR"}}}}}
```

Related Topics

- *JSON-Relational Duality Developer's Guide*

- DBMS_SEARCH Package
- CONTAINS
- JSON_TEXTCONTAINS
- JSON_EXISTS

13.2.3 Examine DBMS_SEARCH Indexes Using Dictionary Views

In this example, you can see how to use various dictionary views to query information about your `DBMS_SEARCH` indexes, such as index name, corresponding schema owner name, data source name added to indexes, or corresponding source owner name and ID.

1. Connect to Oracle Database as a local user.
   a. Log in to SQL*Plus as the `SYS` user, connecting as `SYSDBA`:

```
conn sys/password as sysdba


CREATE TABLESPACE tbs1
DATAFILE 'tbs5.dbf' SIZE 20G AUTOEXTEND ON
EXTENT MANAGEMENT LOCAL
SEGMENT SPACE MANAGEMENT AUTO;


SET ECHO ON
SET FEEDBACK 1
SET NUMWIDTH 10
SET LINESIZE 80
SET TRIMSPOOL ON
SET TAB OFF
SET PAGESIZE 10000
SET LONG 10000
```

   b. Create a local user (`docuser`) and grant necessary privileges:

```
DROP USER docuser cascade;


GRANT DB_DEVELOPER_ROLE, DEFAULT TABLESPACE tbs1 quota unlimited on tbs1
TO docuser IDENTIFIED BY password;
```

   c. Connect as the local user (`docuser`):

```
CONN docuser/password
```

2. Create the `departments` and `employees` tables.

```
CREATE TABLE departments
        ( department_id    NUMBER(5) primary key,
          department_name  VARCHAR2(30),
```

```
  manager_id       NUMBER(6)
);


CREATE TABLE employees
( employee_id    NUMBER(6) primary key,
  first_name     varchar2(4000),
  last_name      varchar2(4000),
  department_id  NUMBER(4)
);
```

3. Populate the tables with data.

```
INSERT INTO departments VALUES(10, 'Administration', 100);
INSERT INTO employees VALUES  (100, 'Robert', 'Smith', 10);
INSERT INTO employees VALUES  (101, 'James', 'Martin' ,10);
INSERT INTO employees VALUES  (102, 'John', 'Doe', 10);

commit;
```

4. Create a ubiquitous search index named `MY_SEARCH_INDEX`.

```
EXEC DBMS_SEARCH.CREATE_INDEX('DOCUSER.MY_SEARCH_INDEX', NULL, 'JSON');
```

5. Add the tables as data sources to your index.

```
exec dbms_search.add_source('DOCUSER.MY_SEARCH_INDEX', 'DEPARTMENTS');

exec dbms_search.add_source('DOCUSER.MY_SEARCH_INDEX', 'EMPLOYEES');
```

6. View information about the `DBMS_SEARCH` indexes that are created in a user's schema.

```
select * from user_dbms_search_indexes;


IDX_NAME
--------------------------------------------------
MY_SEARCH_INDEX
```

7. View information about the data sources that are added to the `DBMS_SEARCH` indexes, created in a user's schema.

```
select * from user_dbms_search_index_sources;
```

Here, the source type `T` implies a "table" source:

```
IDX_NAME
--------------------------------------------------
SRC_OWNER
--------------------------------------------------
SRC_NAME
--------------------------------------------------
SRC_TYPE   SRC_ID
```

```
---------  ----------
MY_SEARCH_INDEX
DOCUSER
EMPLOYEES
T        2


IDX_NAME
--------------------------------------------------
SRC_OWNER
--------------------------------------------------
SRC_NAME
--------------------------------------------------
SRC_TYPE   SRC_ID
---------  ----------
MY_SEARCH_INDEX
DOCUSER
DEPARTMENTS
T        1
```

8. View information about all existing `DBMS_SEARCH` indexes, corresponding to each index owner.

```
select * from all_dbms_search_indexes;
```

```
IDX_OWNER
--------------------------------------------------
IDX_NAME
--------------------------------------------------
DOCUSER
MY_SEARCH_INDEX


CUSTOMER
CUSTOMER_SEARCH_INDEX
```

9. View information about all existing data sources added to various `DBMS_SEARCH` indexes, corresponding to each index owner.

```
select * from all_dbms_search_index_sources;
```

Here, the source types `T` and `J` imply "table" and "JSON Duality view" sources, respectively.

```
IDX_OWNER
------------------------------------------------------
IDX_NAME
------------------------------------------------------
SRC_OWNER
------------------------------------------------------
SRC_NAME
------------------------------------------------------
SRC_TYPE   SRC_ID
---------  ----------
```

```
DOCUSER
MY_SEARCH_INDEX
DOCUSER
EMPLOYEES
T          2


IDX_OWNER
------------------------------------------------------------
IDX_NAME
------------------------------------------------------------
SRC_OWNER
------------------------------------------------------------
SRC_NAME
------------------------------------------------------------
SRC_TYPE   SRC_ID
---------  ----------
DOCUSER
MY_SEARCH_INDEX
DOCUSER
DEPARTMENTS
T       1


IDX_OWNER
------------------------------------------------------------
IDX_NAME
------------------------------------------------------------
SRC_OWNER
------------------------------------------------------------
SRC_NAME
------------------------------------------------------------
SRC_TYPE   SRC_ID
---------  ----------
CUSTOMER
CUSTOMER_SEARCH_INDEX
CUSTOMER
MY_EMP_VIEW
J       1
```

Related Topics

- USER_DBMS_SEARCH_INDEXES
- USER_DBMS_SEARCH_INDEX_SOURCES
- ALL_DBMS_SEARCH_INDEXES
- ALL_DBMS_SEARCH_INDEX_SOURCES

14

Working with a Thesaurus in Oracle Text

You can improve your query application with a thesaurus.

This chapter contains the following topics:

14.1 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.

This section contains the following topics.

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

14.1.1 Oracle Text Thesaurus Creation and Maintenance

If you have the `CTXAPP` role, you can create, modify, delete, import, and export thesauruses and thesaurus entries.

This section contains the following topics.

- **CTX_THES Package:** To maintain and browse your thesaurus programatically, you can use the `CTX_THES` PL/SQL package. With this package, you can browse terms and hierarchical relationships, add and delete terms, add and remove thesaurus relations, and import and export thesauruses in and out of the thesaurus tables.
- **Thesaurus Operators:** To expand query terms according to your loaded thesaurus, you can use the thesaurus operators in the `CONTAINS` clause. For example, use the `SYN` operator to expand a term such as *dog* to its synonyms:

  ```
  'syn(dog)'
  ```

- **ctxload Utility:** You can use the `ctxload` utility to load thesauruses from a plain-text file into the thesaurus tables, and to dump thesauruses from the tables into output (or dump) files.

  You can print the thesaurus dump files, you can use them as input for other applications, and you can use them to load a thesaurus into the thesaurus tables (useful when you want to use an existing thesaurus as the basis for a new thesaurus).

> **WARNING:**
>
> To ensure sound security practices, Oracle recommends that you enter the password for `ctxload` by 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.

> **Note:**
>
> You can also programatically import and export thesauruses in and out of the thesaurus tables using the PL/SQL package `CTX_THES` procedures `IMPORT_THESAURUS` and `EXPORT_THESAURUS`.
>
> Refer to *Oracle Text Reference* for more information about these procedures.

14.1.2 Using a Case-Sensitive Thesaurus

In a case-sensitive thesaurus, terms (words and phrases) are stored exactly as you enter them. For example, if you enter a term in mixed case (using either the `CTX_THES` package or a thesaurus load file), then 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 a case-sensitive thesaurus by using the `-thescase` parameter.

When creating a thesaurus with either `CTX_THES.CREATE_THESAURUS` or `CTX_THES.IMPORT_THESAURUS`, you can specify a case-sensitive thesaurus.

In addition, when you specify a case-sensitive thesaurus in a query, the thesaurus lookup uses the query terms exactly as you enter them 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.*

14.1.3 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 either `CTX_THES.CREATE_THESAURUS` or `CTX_THES.IMPORT_THESAURUS`, the thesaurus is created as case-insensitive by default.

In addition, when you specify a case-insensitive thesaurus 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.

14.1.4 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)
- `CTX_THES.IMPORT_THESAURUS` (PL/SQL)
- `ctxload` utility

> **See Also:**
>
> *Oracle Text Reference* to learn more about using `ctxload` and the `CTX_THES` package

14.1.5 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`, you can use to create a general-purpose, English-language thesaurus.

You can use the thesaurus load file to create a default thesaurus for Oracle Text, or you can use it as the basis for thesauruses tailored to a specific subject or range of subjects.

- **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.

  It 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 depends on the operating system; 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>
    <Oracle_Text_directory>
       sample
           thes
```

> **See Also:**
>
> - *Oracle Database Installation Guide* for the installation documentation specific to your operating system for more information about the directory structure of Oracle Text
> - *Oracle Text Reference* to learn more about using `ctxload` and the `CTX_THES` package

14.2 Defining Terms in a Thesaurus

You can create synonyms, related terms, and hierarchical relationships with a thesaurus.

This section contains the following topics.

14.2.1 Defining Synonyms

If you have a thesaurus of computer science terms, then you might define a synonym for the term *XML* as *extensible markup language.* This synonym enables queries on either of these terms to return the same documents.

```
XML

SYN Extensible Markup Language
```

You can use the `SYN` operator to expand XML into its synonyms:

```
'SYN(XML)'
```

is expanded to:

```
'XML, Extensible Markup Language'
```

14.2.2 Defining Hierarchical Relations

If your document set consists of news articles, you can use a thesaurus to define a hierarchy of geographical terms. Consider the following that describes a geographical hierarchy for the 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 use the `NT` operator to expand a query on California:

```
'NT(California)'
```

is expanded to:

```
'California, Northern California, San Francisco, San Jose, Central Valley,
  Fresno, Southern California, Los Angeles'
```

The resulting hitlist shows all documents related to the state of California regions and cities.

14.3 Using a Thesaurus in a Query Application

When you define a custom thesaurus, you can 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 that you can enhance your query application with a custom thesaurus so that you can process queries more intelligently. Each approach has its advantages and disadvantages.

- 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.

14.4 Loading a Custom Thesaurus and Issuing Thesaurus-Based Queries

You can build and load a custom thesaurus.

The advantage of this method is that you can modify the thesaurus after indexing.

The limitation of this method is that you must use thesaurus expansion operators in your query. Long queries can cause extra overhead in the thesaurus expansion and slow your query down.

To build a custom thesaurus:

1. Create your thesaurus. See "Defining Terms in a Thesaurus".
2. Load the 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 your user name 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)'
   ```

14.5 Augmenting the Knowledge Base with a 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 derived themes for document services.

When you augment the existing knowledge base with your new thesaurus, you query with the `ABOUT` operator. The query 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:

1. Create your custom thesaurus, linking new terms to existing knowledge base terms.
2. Load the thesaurus one of the following ways:
   - Using the `ctxload` utility. See "Example of Loading a Thesaurus with ctxload".
   - Using the PL/SQL procedure `CTX_THES.IMPORT_THESAURUS`. See "Example of Loading a Thesaurus with the CTX_THES.IMPORT_THESAURUS PL/SQL procedure".
3. Compile the loaded thesaurus with the `ctxkbtc` compiler.
4. Index your documents. By default the system creates a theme component for your index.
5. Use the `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 this query:

   ```
   'about(politics)'
   ```

> **See Also:**
>
> - "Defining Terms in a Thesaurus" and "Linking New Terms to Existing Terms"
> - "Compiling a Loaded Thesaurus"

14.5.1 Advantages

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 to create theme summaries and gists.

14.5.2 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 change the thesaurus, you must recompile your thesaurus and reindex your documents.

14.6 Linking New Terms to Existing Terms

When you add terms to the knowledge base, for best results in theme proving, Oracle recommends that you links new terms to one of the categories in the knowledge base.

> **See Also:**
>
> *Oracle Text Reference* for more information about the supplied English knowledge base

If you keep new terms separate from existing categories, fewer themes from new terms are proven. The result 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.

Consider the example: You purchase a `medthes` medical thesaurus containing a hierarchy of medical terms. The following are the top four terms in the thesaurus:

- Anesthesia and Analgesia
- Anti-Allergic and Respiratory System Agents
- Anti-Inflammatory Agents, Antirheumatic Agents, and Inflammation Mediators
- Antineoplastic and Immunosuppressive Agents

To map these terms to the existing *health and medicine* branch in the knowledge base, add the following entries to the medical thesaurus:

```
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
```

14.7 Example of Loading a Thesaurus with ctxload

Assuming the medical thesaurus is in the `med.thes` file, 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 `-user` and let `ctxload` prompt you for your user name and password.

14.8 Example of Loading a Thesaurus with the CTX_THES.IMPORT_THESAURUS PL/SQL procedure

This example creates a case-sensitive thesaurus named `mythesaurus` and imports the thesaurus content in `myclob` into the Oracle Text thesaurus tables:

```
declare
 myclob clob;
begin
 myclob := to_clob('peking SYN beijing BT capital country NT beijing tokyo');
 ctx_thes.import_thesaurus('mythesaurus', myclob, 'Y');
end;
```

The format of the thesaurus to be imported (`myclob` in this example) should be the same as the format in the `ctxload` utility. If the format of the thesaurus to be imported is not correct, then `IMPORT_THESAURUS` raises an exception.

14.9 Compiling a Loaded Thesaurus

To link the loaded `medthes` thesaurus 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.

> **WARNING:**
>
> To ensure sound security practices, Oracle recommends that you enter the password for `ctxload` and `ctxkbtc` in the interactive mode. This mode prompts you for the user password. Oracle strongly recommends that you do not enter a password on the command line.

14.10 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

The supplied knowledge base is like a thesaurus in that it is hierarchical and contains broader terms, narrower terms, and related terms. As such, to improve the accuracy of theme analysis, augment the knowledge base with your industry-specific thesaurus by linking new terms to existing terms.

> **See Also:**
>
> "Augmenting Knowledge Base with Custom Thesaurus"

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"

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.

This section contains the following topics:

- Adding a Language-Specific Knowledge Base
- Limitations for Adding Knowledge Bases

14.10.1 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.

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:

1. Load your custom thesaurus by using `ctxload`.
2. Set `NLS_LANG` so that the language portion is the target language. The charset portion must be a single-byte character set.
3. Compile the loaded thesaurus by using `ctxkbtc` and then enter the password for -user when you are prompted. This statement compiles your language-specific knowledge base from the loaded thesaurus.

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

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.

> **See Also:**
>
> - "Example of Loading a Thesaurus with ctxload"
> - "Compiling a Loaded Thesaurus"

14.10.2 Limitations for Adding Knowledge Bases

Here are the limitations for adding knowledge bases:

- Oracle supplies knowledge bases only in English and French. You must provide your own thesaurus for any other language.
- You can add knowledge bases only for languages with single-byte character sets. You cannot create a knowledge base for languages that can be expressed only in multibyte character sets. If the database is a multibyte universal character set, such as UTF-8, you must still set the `NLS_LANG` parameter to a compatible single-byte character set when you compile the thesaurus.
- Adding a knowledge base works best for whitespace-delimited languages.
- Only one knowledge base is allowed for each `NLS_LANG` language.
- Obtaining hierarchical query feedback information (for example, 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

15
Using Faceted Navigation

Become familiar with the faceted navigation feature.

15.1 About Faceted Navigation

This feature implements group counts, also known as facets, which are frequently used in e-commerce or catalog applications. In various applications, it is preferable not only to display the list of hits returned by a query, but also to categorize the results.

For example, an e-commerce application wants to display all products matching a query for the term *management* along with faceting information. The facets include ‘type of product’ (books or DVDs), ‘author’, and ‘date’. For each facet, the application displays the unique values (books or DVDs) and their counts. You can quickly assess that most of the product offerings of interest fall under the ‘books’ category. You can further refine the search by selecting the ‘books’ value under ‘type of product’.

A group count is defined as the number of documents that have a certain value. If a value is repeated within the same document, the document contributes a count of 1 to the total group count for the value. Group counts or facets are supported for `SDATA` sections that use *`optimized_for search`* `SDATA`. To request a computation of facets for a query, use the Result Set Interface.

15.2 Defining Sections As Facets

`SDATA` refers to structured data. Group counts or facets are supported for `SDATA` sections that you create with the *`optimized_for`* attribute set to either ‘search’ or ‘sort and search’. In the `MULTI_COLUMN_DATASTORE` preference, when data appears between tags or columns that are specified as *`optimized_for search`* `SDATA`, the data is automatically indexed as the facet data. Any data that does not match its declared type is handled according to the same framework that currently handles indexing errors for a specific row.

Examples

In the following statements, some tagged data is inserted into a `VARCHAR2` column of a table. You can later define `SDATA` sections to collect the data based on the tags used here.

- Binary float or binary double with tag **price**:

  ```
  insert into mytab values (1, 'red marble' <price>1.23</price>');
  ```

- Time stamp with tag **T**:

  ```
  insert into mytab values (1,'blue marbles <T>2012-12-05T05:20:00</T>');
  ```

In the following statements, a section group is created and various SDATA section groups are added. The section definition includes the section group to which it belongs, the name of the section, the tag to be looked for, and the data type.

```
exec ctx_ddl.create_section_group('sg','BASIC_SECTION_GROUP')
exec ctx_ddl.add_SDATA_section('sg','sec01','name', 'varchar2')
exec ctx_ddl.add_SDATA_section('sg','sec02','count', 'number')
exec ctx_ddl.add_SDATA_section('sg','sec03','date', 'date')
exec ctx_ddl.add_SDATA_section('sg','sec04','timestamp', 'timestamp')
exec ctx_ddl.add_SDATA_section('sg','sec05','new price', 'binary_double')
exec ctx_ddl.add_SDATA_section('sg','sec06','old price','binary_float')
exec ctx_ddl.add_SDATA_section('sg','sec07','timestamp','timestamp with time
zone')
```

The name given to the facet is 'sec01' and the 'name' tag is the actual tag name that occurs inside the document that is to be indexed. The 'date', 'timestamp', and 'timestamp with time zone' data types require the input data to be in the standard ISO format.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for more information about the standard ISO formats

Example 15-1 Using Faceted Navigation

The following statements create a table named products:

```
drop table products;

create table products(name varchar2(60), vendor varchar2(60), rating number,
price number, mydate date);
```

The following statement inserts values into products:

```
insert all
 into products values ('cherry red shoes', 'first vendor', 5, 129, sysdate)
 into products values ('bright red shoes', 'first vendor', 4, 109, sysdate)
 into products values ('more red shoes', 'second vendor', 5, 129, sysdate)
 into products values ('shoes', 'third vendor', 5, 109, sysdate)
select * from dual;
```

The following statements create a MULTI_COLUMN_DATASTORE preference named ds to bring various other columns into the index (name) to be used as facets:

```
/*A MULTI_COLUMN_DATASTORE automatically adds tags by default so that the
text to be indexed looks like
'<name>cherry red shoes</name><vendor>first vendor</vendor><rating> .... '*/

exec ctx_ddl.drop_preference  ('ds')
exec ctx_ddl.create_preference('ds', 'MULTI_COLUMN_DATASTORE')
```

```
exec ctx_ddl.set_attribute    ('ds', 'COLUMNS', 'name, vendor, rating, price,
mydate')
```

> **Note:**
>
> Oracle does not allow table columns with `binary_float`, `binary_double`, `timestamp`, and `timestamp with timezone` data types. It is therefore difficult to use such data types with `MULTI_COLUMN_DATASTORE`. You can still create facets if the document contains tagged data for these data types. Alternatively, you can convert 'timestamp' columns to 'date' and you can store `binary_float` and `binary_double` as 'number'.

The following statements create a section group named `sg` and enable the `optimized_for search` attribute for each column to be treated as a facet:

```
/* A Section Group is created to specify the data type of each column
(varchar2 is the default) and
how each column that is brought into the index should be used.*/


exec ctx_ddl.drop_section_group   ('sg')
exec ctx_ddl.create_section_group ('sg', 'BASIC_SECTION_GROUP')

exec ctx_ddl.add_sdata_section    ('sg', 'vendor', 'vendor', 'VARCHAR2')
exec ctx_ddl.add_sdata_section    ('sg', 'rating', 'rating', 'NUMBER')
exec ctx_ddl.add_sdata_section    ('sg', 'price', 'price', 'NUMBER')
exec ctx_ddl.add_sdata_section    ('sg', 'mydate', 'mydate', 'DATE')

exec ctx_ddl.set_section_attribute('sg', 'vendor', 'optimized_for', 'SEARCH')
exec ctx_ddl.set_section_attribute('sg', 'rating', 'optimized_for', 'SEARCH')
exec ctx_ddl.set_section_attribute('sg', 'price',  'optimized_for', 'SEARCH')
exec ctx_ddl.set_section_attribute('sg', 'mydate', 'optimized_for', 'SEARCH')
```

The following statement creates an index on `name` and specifies the preferences by using the `PARAMETERS` clause:

```
CREATE INDEX product_index ON products (name)
INDEXTYPE IS ctxsys.context
PARAMETERS ('datastore ds section group sg');
```

The following statements query for a product name, ‘red shoes’ and the facets for computation can be specified. The `count` attribute shows the total number of items that match the query for the product. The Result Set Interface specifies various requirements, such as the top vendors that have the largest number of matching items, the lowest available prices, and the latest arrivals:

```
set long 500000
set pagesize 0

variable displayrs clob;
```

```
declare
  rs clob;
begin
   ctx_query.result_set('product_index', 'red shoes',
'<ctx_result_set_descriptor>
         <count/>
         <group sdata="vendor" topn="5" sortby="count" order="desc">
         <count exact="true"/>
         </group>
         <group sdata="price" topn="3" sortby="value" order="asc">
         <count exact="true"/>
         </group>
         <group sdata="mydate" topn="3" sortby="value" order="desc">
         <count exact="true"/>
         </group>
         </ctx_result_set_descriptor>',
         rs);

/* Pretty-print the result set (rs) for display purposes.
It is not required if you are going to manipulate it in XML.*/

   select xmlserialize(Document XMLType(rs) as clob indent size=2)
into :displayrs from dual;
   dbms_lob.freetemporary(rs);
end;
/
select :displayrs from dual;
```

The following is output:

```
<ctx_result_set>

  <count>3</
count>

  <groups
sdata="VENDOR">
    <group value="first
vendor">
      <count>2</
count>
    </
group>
    <group value="second
vendor">
      <count>1</
count>
    </
group>
  </
groups>

  <groups
sdata="PRICE">
    <group
```

```
value="109">
      <count>1</
count>
    </
group>
    <group
value="129">
      <count>2</
count>
    </
group>
  </
groups>

  <groups
sdata="MYDATE">
    <group value="2017-12-06
05:44:54">
      <count>3</
count>
    </
group>
  </
groups>
</ctx_result_set>
```

15.3 Querying Facets by Using the Result Set Interface

Starting with Oracle Database Release 18c, the group-counting operation for a specified list of facets is provided. You can obtain the group counts for each single value by using the `bucketby` attribute with its value set to `single`. The `topn`, `sortby`, and `order` attributes are also supported. Starting with Oracle Database Release 21c, you can obtain the group counts for a range of numeric and variable character facet values by using the `range` element, which is a child element of the `group` element.

bucketby Attribute

Valid attributes are `single` and `custom`.

- The 'single' mode produces a list of all unique values for the facet and a document count for each value.
- The 'custom' mode produces document counts for a range of numeric values.

count Element (Single Count)

In the following example, a few rows are inserted into the `mytab` table. Some rows have two values for the facet `<B>`, and some rows have a single value.

```
begin
     insert into mytab values (1, '<B>1.234</B><B>5</B>');
     insert into mytab values (2, '<B>1.432</B>');
     insert into mytab values (3, '<B>2.432</B><B>6</B>');
     insert into mytab values (4, '<B>2.432</B>');
end;
```

Single counts show each unique value and the number of documents that have this value:

```
<ctx_result_set>
    <groups sdata="SEC01">
        <group value="2.432"><count>2</count></group>
        <group value="1.234"><count>1</count></group>
        <group value="5"><count>1</count></group>
        <group value="6"><count>1</count></group>
        <group value="1.432"><count>1</count></group>
    </groups>
</ctx_result_set>
```

If document 1 is deleted, you see the following result:

```
<ctx_result_set>
    <groups sdata="SEC01">
        <group value="2.432"><count>2</count></group>
        <group value="6"><count>1</count></group>
        <group value="1.432"><count>1</count></group>
    </groups>
</ctx_result_set>
```

range Element

The `range` element supports `start`, `greaterthan`, `end`, and `lessthan` attributes. The `start` and `greaterthan` attributes specify the beginning value for the range. The `end` and `lessthan` attributes specify the ending value for the range.

Ranges can overlap each other. For example, `<range start="1" end="2"/>` and `<range start="2" end="3"/>`. Ranges can also be open ended. For example, you can specify only the start value or the end value. If you do not specify the attributes of the `range` element, all results are returned.

Example 15-2 Obtaining Group Counts for a Range of Facets

Create a table named `products` and populate it:

```
drop table products;

create table products(name varchar2(60), vendor varchar2(60), rating number,
price number);

insert all
 into products values ('cherry red shoes', 'first vendor', 5, 129)
 into products values ('bright red shoes', 'first vendor', 4, 109)
 into products values ('more red shoes', 'second vendor', 5, 129)
 into products values ('shoes', 'third vendor', 5, 109)
 into products values ('dark red shoes', 'fourth vendor', 3, 98)
 into products values ('light red shoes', 'fifth vendor', 2, 49)
select * from dual;
```

Create a `MULTI_COLUMN_DATASTORE` preference named `ds` to bring various other columns into the index `(name)` to be used as facets:

```
exec ctx_ddl.drop_preference  ('ds')
exec ctx_ddl.create_preference('ds', 'MULTI_COLUMN_DATASTORE')
exec ctx_ddl.set_attribute    ('ds', 'COLUMNS', 'name, vendor, rating, price')
```

Create a section group named `sg` and enable the `optimized_for search` attribute for each column to be treated as a facet:

```
exec ctx_ddl.drop_section_group   ('sg')
exec ctx_ddl.create_section_group ('sg', 'BASIC_SECTION_GROUP')

exec ctx_ddl.add_sdata_section    ('sg', 'rating', 'rating', 'NUMBER')
exec ctx_ddl.add_sdata_section    ('sg', 'price', 'price', 'NUMBER')
exec ctx_ddl.add_sdata_section    ('sg', 'vendor', 'vendor', 'VARCHAR2')

exec ctx_ddl.set_section_attribute('sg', 'rating', 'optimized_for', 'SEARCH')
exec ctx_ddl.set_section_attribute('sg', 'price',  'optimized_for', 'SEARCH')
exec ctx_ddl.set_section_attribute('sg', 'vendor',  'optimized_for', 'SEARCH')
```

Create an index on `name` and specify the preferences by using the `parameters` clause:

```
create index mytab_idx on products (name)
indextype is ctxsys.context
parameters ('datastore ds section group sg');
```

Query for a product name, ‘red shoes’ by setting the `bucketby` attribute to `custom` and provide the values for the `range` element:

```
set long 500000
set pagesize 0

variable displayrs clob;

declare
  rs clob;
begin
     ctx_query.result_set('mytab_idx', 'red shoes',
'<ctx_result_set_descriptor>
         <group sdata="rating" bucketby="custom">
          <range start="1" lessthan="10"/>
          <range start="10" lessthan="20"/>
          <range start="20"/>
        </group>
        <group sdata="price" bucketby="custom">
          <range end="1"/>
          <range greaterthan="1" end="10"/>
          <range greaterthan="10" end="100"/>
          <range greaterthan="100"/>
        </group>
        <group sdata="vendor" bucketby="custom">
          <range greaterthan="a"/>
          <range start="s"/>
```

```
          <range end="f"/>
        </group>
    </ctx_result_set_descriptor>',
         rs);

   select xmlserialize(Document XMLType(rs) as clob indent size=2)
into :displayrs from dual;
   dbms_lob.freetemporary(rs);
end;
/
select :displayrs from dual;
```

The following is output:

```
<ctx_result_set>
  <groups sdata="RATING">
    <group value="range" start="1" lessthan="10">
      <count>5</count>
    </group>
    <group value="range" start="10" lessthan="20">
      <count>0</count>
    </group>
    <group value="range" start="20" end="5">
      <count>0</count>
    </group>
  </groups>
  <groups sdata="PRICE">
    <group value="range" start="49" end="1">
      <count>0</count>
    </group>
    <group value="range" greaterthan="1" end="10">
      <count>0</count>
    </group>
    <group value="range" greaterthan="10" end="100">
      <count>2</count>
    </group>
    <group value="range" greaterthan="100" end="129">
      <count>3</count>
    </group>
  </groups>
  <groups sdata="VENDOR">
    <group value="range" greaterthan="a" end="second vendor">
      <count>5</count>
    </group>
    <group value="range" start="s" end="second vendor">
      <count>1</count>
    </group>
    <group value="range" start="fifth vendor" end="f">
      <count>0</count>
    </group>
  </groups>
</ctx_result_set>
```

topn Attribute

- Valid attribute values are non-negative numbers greater than zero.
- This attribute specifies that only top `n` facet values and their counts are returned.
- Group count determines the top `n` values to return unless the `sortby` attribute is set to `value`. In that case, the values are sorted according to the data type and the top `n` results of the sort are returned. The order attribute is respected for the sort.
- By default, the results are sorted by the group count in descending order.
- If a tie occurs in the count, the ordering of the facet values within this tie is not guaranteed.

sortby and order Attributes

`sortby` supports `count` and `value` attributes.

- `count` sorts by group counts (numbers). This is the default.
- `value` sorts by value depending on the data type.

`order` supports `ASC` (ascending order) and `DESC` (descending order), which is the default.

If there is no selection, the default is `count DESC`.

This example shows the grouping of a number facet if `bucketby` is set to `single`, where `mytab_idx` is the name of the index, `text` is the query, and group `SDATA` requests the facets:

```
begin
     ctx_query.result_set('mytab_idx', 'text',
                          '<ctx_result_set_descriptor>
     <group sdata="sec01" topn = "4" sortby = "value" order="asc"
bucketby="single">
     <count/>
     </group>
     </ctx_result_set_descriptor>'
     :rs);
end;
```

The following is a sample output showing that the values are listed in alphabetical order because the `sortby` attribute is set to `value` instead of `count`. The values are also displayed in ascending order (ABC to XYZ) because the `order` attribute is set to `asc`. Only four values are displayed because the `topn` attribute is set to `4`.

```
<ctx_result_set>
     <group SDATA="SEC01">
          <group value="ABC"><count>2</count>
     </group>
          <group value="DEF"><count>1</count>
     </group>
          <group value="GHI"><count>10</count>
     </group>
          <group value="XYZ"><count>1</count>
     </group>
</ctx_result_set>
```

15.4 Refining Queries by Using Facets As Filters

The facet implementation now supports `CONTAINS` queries with the standard set of database comparison operators available for SDATA. The following example is based on the 'name' varchar2 section. When you use it with numbers, do not use quotation marks around the numeric term to be searched.

```
contains (text, 'SDATA(sec01 = "run")', 1) > 0
contains (text, 'SDATA(sec01 > "run")', 1) > 0
contains (text, 'SDATA(sec01 >= "run")', 1) > 0
contains (text, 'SDATA(sec01 < "run")', 1) > 0
contains (text, 'SDATA(sec01 <= "run")', 1) > 0
contains (text, 'SDATA(sec01 <> "run")', 1) > 0
contains (text, 'SDATA(sec01 != "run")', 1) > 0
contains (text, 'SDATA(sec01 between "run1" and "run2")', 1) > 0
contains (text, 'SDATA(sec01 not between "run1" and "run2")', 1) > 0
contains (text, 'SDATA(sec01 is null)', 1) > 0
contains (text, 'SDATA(sec01 is not null)', 1) > 0
contains (text, 'SDATA(sec01 like "%run")', 1) > 0
contains (text, 'SDATA(sec01 like "run%")', 1) > 0
contains (text, 'SDATA(sec01 like "%run%")', 1) > 0
contains (text, 'SDATA(sec01 not like "%run")', 1) > 0
contains (text, 'SDATA(sec01 not like "run%")', 1) > 0
contains (text, 'SDATA(sec01 not like "%run%")', 1) > 0

contains (text, 'SDATA(sec02 = 9)', 1) > 0
contains (text, 'SDATA(sec02 < 10)', 1) > 0
contains (text, 'SDATA(sec02 between 2 and  20)', 1) > 0
```

The comparison operators behave according to the current *optimized_for search* `SDATA` behavior for the various data types.

15.5 Multivalued Facets

If multiple values are in an *optimized for search* `SDATA` section within the same document, then each value is indexed if the value is enclosed in its own tag corresponding to the SDATA section. Values that are not enclosed within separate section tags, but that appear together within the same section tag, are treated as a single value.

For example, in a document, `<car>First Car, Second Car</car>` is treated as a single string of value 'First Car, Second Car'. However, `<car>First Car</car><car>Second Car</car>` is treated as two separate values for the document.

16

Using Result Set Interface

The CTX_QUERY.RESULT_SET procedure executes an XML or JSON query and generates a result set in XML or JSON.

> **Note:**
>
> The Oracle Text Result Set Interface queries are not supported on shard catalog instances.

16.1 Overview of the XML Query Result Set Interface

The XML Query Result Set Interface (RSI) enables you to perform queries in XML and return results as XML, avoiding the SQL layer and requirement to work within SELECT semantics. The RSI uses a simple Oracle Text query and an XML result set descriptor, where the hitlist is returned in XML according to the result set descriptor. The XML Query RSI uses SDATA sections for grouping and counting.

In applications, a page of search results can consist of many disparate elements, such as metadata of the first few documents, total hit counts, and per-word hit counts. Each extra call takes time to reparse the query and look up index metadata. Additionally, some search operations, such as iterative query refinement, 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 XML Query RSI 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 RSI can also return data views that are difficult to express in SQL.

16.2 Using the XML Query Result Set Interface

The CTX_QUERY.RESULT_SET() and CTX_QUERY.RESULT_SET_CLOB_QUERY() APIs enable you to obtain query results with a single query, rather than running multiple CONTAINS() queries to achieve the same result. The two APIs are identical except that one uses a VARCHAR2 query parameter, and the other uses a CLOB query parameter to allow for longer queries.

For example, to display a search result page, you must first get the following information:

- Top 20 hit list sorted by date and relevancy
- Total number of hits for the given Oracle 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 issue four SQL statements to obtain the four pieces of information needed for displaying the search result page:

```
-- 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 Oracle 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, because you run the same query four times. However, if you use `CTX_QUERY.RESULT_SET()`, then you can enter all of the 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 is XML that as the 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 on `CTX_QUERY.RESULT_SET`

16.3 Creating XML-Only Applications with Oracle Text

Although it is common to create applications by using SQL `SELECT` statements with the `CONTAINS` clause, it is not the most efficient method. An alternative method is to use the XML-based RSI. With this method, you can obtain summary information (such as the total number of hits) without fetching all results of the query.
To use the RSI, you specify a Result Set Descriptor (RSD). The RSD declares the information to be returned, which can consist of:

- Total result count of the query
- Hitlist
- Summary information over `SDATA` fields

In turn, the hitlist consists of repeating elements, each of which may contain:

- Rowid of the hit
- SDATA fields from the hit

Related Topics

- Example of a Result Set Descriptor

16.4 Example of a Result Set Descriptor

This example shows how to use an RSD. The following example requests a hitlist with the top 10 hits (ordered by score) and the count of the total number of results.

```
<ctx_result_set_descriptor>
  <hitlist start_hit_num="1" end_hit_num="10" order="SCORE DESC">
    <rowid />
    <sdata name="title" />
    <sdata name="author" />
    <sdata name="articledate" />
    <snippet radius="20" max_length="160" starttag="&lt;b&gt;" endtag="&lt;/b&gt;" />
  </hitlist>
  <count />
</ctx_result_set_descriptor>
```

For each hit, you are requesting the `rowid` (which you could use to fetch further information about the row, if necessary), the contents of the `SDATA` fields or the `title`, `author`, and `articledate` columns, and a snippet (which is a short summary with keywords highlighted, in this case by `<b>...</b>`).

16.5 Identifying Collocates

Collocates are a group of words that frequently co-occur in a document. They provide a quick summary of other keywords or concepts that are related to a specified keyword. You can then use the other keywords in queries to fetch more relevant results.

You identify collocates based on a search query. For each document that is returned by the query, snippets of text around the search keyword are automatically extracted. Next, the words in these snippets are correlated to the query keyword by using statistical measures and, depending on how frequently the extracted words occur in the overall document set, a score is assigned to each returned co-occurring word.

Use the RSI to identify collocates. You can specify the number of co-occurring words that must be returned by the query. You can also specify whether to identify collocates that are common nouns or collocates that emphasize uniqueness. Synonyms of the specified search keyword can also be returned.

> **Note:**
>
> Collocates are supported only for `BASIC_LEXER`.

To identify collocates:

1. Create the document set table for the query.
2. Create an Oracle Text index on the document set table.
3. Use the XML Query RSI to define and input a query that identifies collocates. Include the `collocates` element with the required attributes.

Example 16-1 Identifying Collocates Within a Document Set

In this example, the keyword used to query documents in a data set is ‘Nobel.’ Oracle Text searches for occurrences of this keyword in the document set. In addition to the result set, use collocates to search for five common words that co-occur with ‘Nobel.’ Use the `max_words` attribute to identify the number of collocates to be generated. Set the `use_tscore` attribute to `TRUE` to specify that common words must be identified for the collocates. The number of words to pick on either side of the keyword in order to identify collocates is 10.

The following is the input RSI descriptor that is used to determine collocates:

```
declare
rsd varchar2(32767);
 begin
  ctx_query.result_set('tdrbnbsan01idx', 'nobel',
  <ctx_result_set_descriptor>
  <collocates radius = "10" max_words="5" use_tscore="TRUE"/>
  </ctx_result_set_descriptor>',
  :rs);
```

```
  end;
/
```

Here is the output result set for the query:

```
<ctx_result_set>
<collocates>
    <collocation>
       <word>PRIZE</word>
       <score>82</score>
    </collocation>
    <collocation>
       <word>LAUREATE</word>
       <score>70</score>
    </collocation>
    <collocation>
       <word>NOBELPRIZE</word>
       <score>44</score>
    </collocation>
    <collocation>
       <word>AWARD</word>
       <score>42</score>
    </collocation>
    <collocation>
       <word>ORG</word>
       <score>41</score>
    </collocation
</collocates>
</ctx_result_set>
```

For ‘Nobel,’ the top five common collocates, in order, are Prize, Laureate, Nobelprize, award, and org. Each word is assigned a score that indicates the frequency of occurrence. Collocates are always returned after any `hitlist` elements are returned.

If you set `use_tscore` to `FALSE` in the same example, then less common (unique) words are identified. Here is the output result set:

```
<ctx_result_set>
<collocates>
    <collocation>
       <word>MOLA</word>
       <score>110</score>
    </collocation>
    <collocation>
       <word>BISMARCK</word>
       <score>89</score>
    </collocation>
    <collocation>
       <word>COLONNA</word>
       <score>67</score>
    </collocation>
    <collocation>
       <word>LYNEN</word>
       <score>55</score>
    </collocation>
```

```
    <collocation>
       <word>TIMBERGEN</word>
       <score>25</score>
    </collocation>
    </collocates>
</ctx_result_set>
```

> **See Also:**
>
> *Oracle Text Reference* for information about attributes used with collocates

16.6 Overview of the JSON Result Set Interface

The JSON Result Set Interface (RSI) enables you to perform queries in JSON and return results as JSON, avoiding the SQL layer and requirement to work within `SELECT` semantics.

The RSI uses a simple Oracle Text query or facets and a JSON result set descriptor, where the hitlist is returned in one single CLOB of JSON according to the result set descriptor. The JSON RSI uses `SDATA` sections for grouping and counting.

In applications, a page of search results can consist of many disparate elements, such as metadata of the first few documents, total hit counts, and per-word hit counts. Each extra call takes time to reparse the query and look up index metadata. Additionally, some search operations, such as iterative query refinement, 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 JSON RSI 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 RSI can also return data views that are difficult to express in SQL.

The JSON RSI supports queries based on `CONTEXT` and JSON search indexes. You can also perform other aggregations in facets like `COUNT`, `MIN`, and `MAX` apart from the supported group counts. `AVG` and `SUM` are supported for numeric facets.

16.7 Using the JSON Result Set Interface

The `CTX_QUERY.RESULT_SET()` and `CTX_QUERY.RESULT_SET_CLOB_QUERY()` APIs enable you to obtain query results with a single query, rather than running multiple `CONTAINS()` queries to achieve the same result. The two APIs are identical except that one uses a `VARCHAR2` query parameter, and the other uses a `CLOB` query parameter to allow for longer queries.

Usage

The input Result Set Descriptor (RSD) query consists of the following parts:

- **$query** - Use `$query` to specify a search query, the path constraints, and additional path based filter conditions. The `$query` part is supported only when a JSON search index exists on the column.
- **$search** - Use `$search` to display the score ranked search results and their count. For a non-JSON Oracle Text full-text index, you can also specify the `SDATA` sections to project for the search results.

- **$facet** - Use `$facet` to specify the facets for various paths of a JSON document or `SDATA` sections of a context indexed document. Facets bucketed by a single unique value and facets per user specified range buckets are supported. The facts can also be one of the aggregations like `COUNT`, `MIN`, etc.

The result set output is of the following format:

```
{
  "$count" : number ,
  "$hit" :
  [
    {
      "score" : <search_score>,
      "rowid" : <rowid>,
      "project" : {"<sdata_name>" : <sdata_value>, … }
    },
    …
  ],
"$facets" :
  [
    {"<field>" : [ ..., { "value" : <value_i>, "$uniqueCount" :
<group_count_i>}, ...  ]},
    {"<field>" : [ ..., { "bucket" : <bucket_object_i>, "<op>" :
<group_count_i>}, ...  ]},
    {"<field>" : { "<op>" : <actual_value of the aggregation> } },
    …
  ]
}
```

> **See Also:**
>
> *Oracle Text Reference* for more information about `CTX_QUERY.RESULT_SET` procedure

17

Performing Sentiment Analysis Using Oracle Text

Sentiment analysis enables you to identify a positive or negative sentiment in a search topic.

This chapter contains the following topics:

17.1 Overview of Sentiment Analysis

Sentiment analysis uses trained sentiment classifiers to provide sentiment information for documents or topics within documents.

This section contains the following topics:

17.1.1 About Sentiment Analysis

Oracle Text enables you to perform sentiment analysis for a topic or document by using sentiment classifiers that are trained to identify sentiment metadata.

With growing amounts of data, organizations must gain more insights about their data rather than just obtaining hits in response to a search query. The insight could be in the form of answering certain basic types of queries (such as weather queries or queries about recent events) or providing opinions about user-specified topics. Keyword searches provide a list of results containing the search term. However, to identify a sentiment or opinion about the search term, must browse through the results and then manually locate the required sentiment information. Sentiment analysis provides a one-step process to identify sentiment information within a set of documents.

Sentiment analysis is the process of identifying and extracting sentiment metadata about a specified topic or entity from a set of documents. Trained sentiment classifiers identify the sentiment. When you run a query with sentiment analysis, in addition to the search results, sentiment metadata is also identified and displayed. Sentiment analysis provides answers to questions such as “Is a product review positive or negative?” or “Is the customer satisfied or dissatisfied?” For example, from a document set consisting of multiple reviews for a particular product, you can determine an overall sentiment that indicates if the product is good or bad.

17.1.2 About Sentiment Classifiers

A sentiment classifier is a type of document classifier that is used to extract sentiment metadata about a topic or document.

To perform sentiment analysis by using a sentiment classifier, you must first associate a sentiment classifier preference with the sentiment classifier and then train the sentiment classifier.

You can associate user-defined sentiment classifiers with a sentiment classifier preference of type `SENTIMENT_CLASSIFIER`. A sentiment classifier preference specifies the parameters that are used to train a sentiment classifier. These parameters are defined as attributes of the sentiment classifier preference. You can either create a sentiment classifier preference or use the default `CTXSYS.DEFAULT_SENTIMENT_CLASSIFIER`. To create a user-defined sentiment classifier preference, use the `CTX_DDL.CREATE_PREFERENCE` procedure to define a sentiment classifier preference and the `CTX_DDL.SET_ATTRIBUTE` procedure to define its parameters.

To train a sentiment classifier, you need to provide an associated sentiment classifier preference, a training set of documents, and the sentiment categories. If you do not specify a classifier preference, then Oracle Text uses default values for the training parameters. You train the sentiment classifier by using the set of sample documents and the specified preference. You assign each sample document to a category. Oracle Text uses this sentiment classifier to deduce a set of classification rules that define how sentiment analysis must be performed. Use the `CTX_CLS.SA_TRAIN` procedure to train a sentiment classifier.

Typically, you define and train separate sentiment classifiers for different categories of documents, such as finance, product reviews, and music. If you do not want to create your own sentiment classifier or if suitable training data is not available to train your classifier, you can use the default sentiment classifier provided by Oracle Text. The default sentiment classifier is unsupervised.

> **Note:**
>
> The default sentiment classifier works only with `AUTO_LEXER`. Do not use `AUTO_LEXER` with user-defined sentiment classifiers.

> **See Also:**
>
> - Creating a Sentiment Classifier Preference
> - Training Sentiment Classifiers

17.1.3 About Performing Sentiment Analysis

To perform sentiment analysis, you run a sentiment query that includes the sentiment classifier which must be used to identify sentiment information. The classifier can be the default or a user-defined sentiment classifier.

You can perform sentiment analysis only as part of a search operation. Oracle Text searches for the specified keywords and generates a result set. Then, sentiment analysis is performed

on the result set to identify a sentiment score for each result. If you do not explicitly specify a sentiment classifier in your query, the default classifier is used.

You can either identify one single sentiment for the entire document or separate sentiments for each topic within a document. Most often, a document contains multiple topics and the author's sentiment toward each topic may be different. In such cases, document-level sentiment scores may not be useful because they cannot identify sentiment scores associated with different topics in the document. Identifying topic-level sentiment scores provides the required answers. For example, when searching through a set of documents containing reviews for a camera, a document-level sentiment tells you whether the camera is good or not. Assume that you want the general opinion about the picture quality of a camera. Performing a topic-level sentiment analysis, with "picture quality" as one of the topics provides the required information.

Note:

If you do not specify a topic of interest for sentiment analysis, then Oracle Text returns the overall sentiment for the entire document.

See Also:

- Performing Sentiment Analysis with the CTX_DOC Package
- Performing Sentiment Analysis with the RSI

17.1.4 Sentiment Analysis Interfaces

Oracle Text supports multiple interfaces for performing sentiment analysis.

Use one of the following interfaces to run a sentiment query:

- Procedures in the `CTX_DOC` package
- XML Query Result Set Interface (RSI)

See Also:

- Performing Sentiment Analysis with the CTX_DOC Package
- Performing Sentiment Analysis with the RSI

17.2 Creating a Sentiment Classifier Preference

Use the `CTX_DDL.CREATE_PREFERENCE` procedure to create a sentiment classifier preference and the `CTX_DDL.SET_ATTRIBUTE` procedure to define its attributes. The classifier type associated with a user-defined sentiment classifier preference is `SENTIMENT_CLASSIFIER`.

To create a sentiment classifier preference:

1. To define a sentiment classifier preference, use the `CTX_DDL.CREATE_PREFERENCE` procedure. The classifier must be of type `SENTIMENT_CLASSIFIER`.
2. To define attributes for the sentiment classifier preference, use the `CTX_DDL.SET_ATTRIBUTE` procedure. The attributes define the parameters that are used to train the sentiment classifier.

Example 17-1 Creating a Sentiment Classifier Preference

The following example creates a sentiment classifier preference named `clsfier_camera`. This preference is used to classify a set of documents that contain reviews for SLR cameras.

1. Define a sentiment classifier preference named `clsfier_camera` with type `SENTIMENT_CLASSIFIER`.

```
exec ctx_ddl.create_preference('clsfier_camera','SENTIMENT_CLASSIFIER');
```

2. Define the attributes of the `clsfier_camera` sentiment classifier preference. Set 1000 for the maximum number of features to be extracted. Set 600 for the number of iterations for which the classifier runs.

```
exec ctx_ddl.set_attribute('clsfier_camera','MAX_FEATURES','1000');
exec ctx_ddl.set_attribute('clsfier_camera','NUM_ITERATIONS','600');
```

For attributes that are not explicitly defined, the default values are used.

> **See Also:**
>
> - *Oracle Text Reference*
> - About Sentiment Classifiers

17.3 Training Sentiment Classifiers

Training a sentiment classifier generates the classification rules that are used to provide a positive or negative sentiment for a search keyword.

The following example trains a sentiment classifier that can perform sentiment analysis on user reviews of cameras:

1. Create and populate the training document table. This table contains the actual text of the training set documents or the file names (if the documents are present externally).

   Ensure that the training documents are randomly selected to avoid any possible bias in the trained sentiment classifier. The distribution of positive and negative documents must not be skewed. Oracle Text checks for the distribution while training the sentiment classifier.

```
create table training_camera (review_id number primary key, text
varchar2(2000));
insert into training_camera values( 1,'/sa/reviews/cameras/review1.txt');
insert into training_camera values( 2,'/sa/reviews/cameras/review2.txt');
insert into training_camera values( 3,'/sa/reviews/cameras/review3.txt');
insert into training_camera values( 4,'/sa/reviews/cameras/review4.txt');
```

2. Create and populate the category table.

This table specifies training labels for the documents present in the document table. It tells the classifier the true sentiment of the training set documents.

The primary key of the document table must have a foreign key relationship with the unique key of the category table. The names of these columns must be passed to the `CTX_CLS.SA_TRAIN` procedure so that the sentiment label can be associated with the corresponding document.

Oracle Text validates the parameters specified for the classifier preference and the `category` values. The `category` values are restricted to 1 for positive, 2 for negative, and 0 for neutral sentiment. Documents with a `category` of 0 (neutral documents) are not used while training the classifier. Additional columns in the category table, other than document ID and category, are also not used by the classifier.

```
create table train_category (doc_id number, category number, category_desc
varchar2(100));

insert into train_category values (1,0,'neutral');
insert into train_category values (2,1,'positive');
insert into train_category values (3,2,'negative');
insert into train_category values (4,2,'negative');
```

3. Create the context index on the training document table. This index is used to extract metadata for training documents while training the sentiment classifier.

   In this example, create an index without populating it.

   ```
   exec ctx_ddl.create_preference('fds','DIRECTORY_DATASTORE');
   create index docx on training_camera(text) indextype is ctxsys.context
   parameters ('datastore fds nopopulate');
   ```

4. (Optional) Create a `clsfier_camera` sentiment classifier preference that performs sentiment analysis on a document set consisting of camera reviews.

5. Train the sentiment classifier `clsfier_camera`.

   During training, Oracle Text determines the ratio of positive to negative documents. If this ratio is not in the range of 0.4 to 0.6, then a warning written to the CTX log indicates that the sentiment classifier is skewed. After the sentiment classifier is trained, it is ready to be used in sentiment queries to perform sentiment analysis.

   In the following example, `clsfier_camera` is the name of the sentiment classifier that is being trained, `review_id` is the name of the document ID column in the document training set, `train_category` is the name of the category table that contains the labels for the training set documents, `doc_id` is the document ID column in the category table, `category` is the category column in the category table, and `clsfier` is the name of the sentiment classifier preference that is used to train the classifier.

   ```
   exec
   ctx_cls.sa_train_model('clsfier_camera','docx','review_id','train_category'
   ,'doc_id','category','clsfier');
   ```

> **Note:**
>
> If you do not specify a sentiment classifier preference when running the `CTX_CLS.SA_TRAIN_MODEL` procedure, then Oracle Text uses the default preference `CTXSYS.DEFAULT_SENTIMENT_CLASSIFIER`.

Related Topics

- About Sentiment Classifiers
  A sentiment classifier is a type of document classifier that is used to extract sentiment metadata about a topic or document.
- Creating a Sentiment Classifier Preference
  Use the `CTX_DDL.CREATE_PREFERENCE` procedure to create a sentiment classifier preference and the `CTX_DDL.SET_ATTRIBUTE` procedure to define its attributes. The classifier type associated with a user-defined sentiment classifier preference is `SENTIMENT_CLASSIFIER`.
- *Oracle Text Reference*

17.4 Performing Sentiment Analysis with the CTX_DOC Package

Use the procedures in the `CTX_DOC` package to perform sentiment analysis on a single document within a document set. For each document, you can either determine a single sentiment score for the entire document or individual sentiment scores for each topic within the document.

Before you perform sentiment analysis, you must create a context index on the document set. The following command creates a `camera_revidx` context index on the document set in the `camera_reviews` table:

```
create index camera_revidx on camera_reviews(review_text) indextype is
ctxsys.context parameters ('lexer mylexer stoplist ctxsys.default_stoplist');
```

To perform sentiment analysis with the `CTX_DOC` package, use one of the following methods:

- Run the `CTX_DOC.SENTIMENT_AGGREGATE` procedure with the required parameters.

  This procedure provides a single consolidated sentiment score for the entire document.

  The sentiment score is a value in the range of -100 to 100, and it indicates the strength of the sentiment. A negative score represents a negative sentiment and a positive score represents a positive sentiment. Based on the sentiment scores, you can group scores into labels such as Strongly Negative (–80 to –100), Negative (–80 to –50), Neutral (-50 to +50), Positive (+50 to +80), and Strongly Positive (+80 to +100).
- Run the `CTX_DOC.SENTIMENT` procedure with the required parameters.

  This procedure returns the individual segments within the document that contain the search term, and provides an associated sentiment score for each segment.

Example 17-2 Obtaining a Single Sentiment Score for a Document

The following example uses the `clsfier_camera` sentiment classifier to provide a single aggregate sentiment score for the entire document. The sentiment classifier was created and trained. The table containing the document set has a `camera_revidx` context index. The

doc_id of the document within the document table for which sentiment analysis must be performed is 49. The topic for which a sentiment score is being generated is 'Nikon.'

```
select
ctx_doc.sentiment_aggregate('camera_revidx','49','Nikon','clsfier_camera')
from dual;

CTX_DOC.SENTIMENT_AGGREGATE('CAMERA_REVIDX','49','NIKON','CLSFIER_CAMERA')
--------------------------------------------------------------------------
                            74
1 row selected.
```

Example 17-3 Obtaining a Single Sentiment Score with the Default Classifier

The following example uses the default sentiment classifier to provide an aggregate sentiment score for the entire document. The table containing the document set has a camera_revidx context index. The doc_id of the document within the document table for which sentiment analysis must be performed is 1.

```
select ctx_doc.sentiment_aggregate('camera_revidx','1') from dual;

CTX_DOC.SENTIMENT_AGGREGATE('CAMERA_REVIDX','1')
------------------------------------------------
                                              2

1 row selected.
```

Example 17-4 Obtaining Sentiment Scores for Each Topic Within a Document

The following example uses the clsfier_camera sentiment classifier to generate sentiment scores for each segment within the document. The sentiment classifier was created and trained. The table containing the document set has a camera_revidx context index . The doc_id of the document within the document table for which sentiment analysis must be performed is 49. The topic for which a sentiment score is being generated is 'Nikon.' The restab result table, which will be populated with the analysis results, was created with the columns snippet (CLOB) and score (NUMBER).

```
exec
ctx_doc.sentiment('camera_revidx','49','Nikon','restab','clsfier_camera',
starttag=>'<<', endtag=>'>>');

SQL> select * from restab;
SNIPPET
--------------------------------------------------------------------------------
--
     SCORE
----------
It took <<Nikon>> a while to produce a superb compact 85mm lens, but this
time they finally got it right.
        65

Without a doubt, this is a fine portrait lens for photographing head-and-
shoulder portraits (The only lens which is optically better is
<<Nikon>>'s legendary 10
5mm f2.5 Nikkor lens, and its close optical twin, the 105mm f2.8 Micro Nikkor.
```

```
        75

Since the 105mm f2.5 Nikkor lens doesn't have an autofocus version, then this
might be the perfect moderate telephoto lens for owners of
<<Nikon>> autofocus
SLR cameras.
        84
3 rows selected.
```

Example 17-5 Obtaining a Sentiment Score for a Topic Within a Document

The following example uses the `tdrbrtsent03_cl` sentiment classifier to generate a sentiment score for each segment within the document. The sentiment classifier was created and trained. The table containing the document set has a `tdrbrtsent03_idx` context index. The `doc_id` of the document within the document table for which sentiment analysis must be performed is 1. The topic for which a sentiment score is being generated is 'movie.' The `tdrbrtsent03_rtab` result table, which will be populated with the analysis results was created with the columns snippet and score.

```
SQL> exec
ctx_doc.sentiment('tdrbrtsent03_idx','1','movie','tdrbrtsent03_rtab','tdrbrtse
nt03_cl');
PL/SQL procedure successfully completed.

SQL> select * from tdrbrtsent03_rtab;
SNIPPET
--------------------------------------------------------------------------------
--
SCORE
----------
the <b>movie</b> is a bit overlong , but nicholson is such good fun that the
running time passes by pretty quickly
 -62

1 row selected.
```

> **See Also:**
>
> - `CTX_DOC.SENTIMENT_AGGREGATE` in the *Oracle Text Reference*
> - `CTX_DOC.SENTIMENT` in the *Oracle Text Reference*

17.5 Performing Sentiment Analysis with the RSI

The XML Query Result Set Interface (RSI) enables you to perform sentiment analysis on a set of documents by using either the default sentiment classifier or a user-defined sentiment classifier. The documents on which sentiment analysis must be performed are stored in a document table.

Use the `sentiment` element in the input RSI to indicate that sentiment analysis, in addition to other operations specified in the Result Set Descriptor (RSD), must be performed at query time. If you specify a value for the `classifier` attribute of the `sentiment` element, then the

specified sentiment classifier is used to perform the sentiment analysis. If the `classifier` attribute is omitted, then Oracle Text performs sentiment analysis by using the default sentiment classifier. The `sentiment` element contains a child element called `item` that specifies the topic or concept about which a sentiment must be generated during sentiment analysis.

You can generate either a single sentiment score for each document or separate sentiment scores for each topic within the document. Use the `agg` attribute of the `item` element to generate a single aggregated sentiment score for each document.

You can perform sentiment classification by using a keyword query or the `ABOUT` operator. When you use the `ABOUT` operator, the result set includes synonyms of the keyword that are identified by using the thesaurus.

To perform sentiment analysis by using RSI:

1. Create and train the sentiment classifier you will use to perform sentiment analysis.
2. Create the document table that contains the documents to be analyzed and a context index on the document table.
3. Use the required elements and attributes within a query to perform sentiment analysis.

   The RSI must contain the `sentiment` element.

Example 17-6 Input the RSD to Perform Sentiment Analysis

The following example performs sentiment analysis and generates a sentiment for the 'lens' topic. The driving query is a keyword query for 'camera.' The `sentiment` element specifies that sentiment analysis must be performed by using the `clsfier_camera` sentiment classifier. This classifier was previously created and trained by using the `CTX_CLS.SA_TRAIN_MODEL` procedure. The `camera_revidx` context index is on the document set table.

The sentiment score ranges from -100 to 100. A positive score indicates positive sentiment, whereas a negative score indicates negative sentiment. The absolute value of the score is indicative of the magnitude of positive and negative sentiment.

To perform sentiment analysis and obtain a sentiment score for each topic within the document:

1. Create the `rs` result set table that will store the results of the search operation.

```
SQL> var rs clob;
SQL> exec dbms_lob.createtemporary(:rs, TRUE, DBMS_LOB.SESSION);
```

2. Perform sentiment analysis as part of a search query.

   The keyword being searched for is 'camera.' The topic for which sentiment analysis is performed is 'lens.'

```
begin
ctx_query.result_set('camera_revidx','camera','
    <ctx_result_set_descriptor>
        <hitlist start_hit_num="1" end_hit_num="10" order="score desc">
        <sentiment classifier="clsfier_camera">
           <item topic="lens" />
           <item topic="picture quality" agg="true" />
       </sentiment> </hitlist>
   </ctx_result_set_descriptor>',:rs);
end;
```

```
/
```

3. View the results stored in the result table.

   Other applications can use the XML result set for further processing. For brevity, some output was removed. For each segment within the document, a score represents the sentiment score for the segment.

```
SQL> select xmltype(:rs) from dual;
XMLTYPE(:RS)
--------------------------------------------------------------------------------
-----
<ctx_result_set>
  <hitlist>
    <hit>
      <sentiment>
         <item topic="lens">
            <segment>
               <segment_text>The first time it was sent in was because the
<b>lens </b> door failed to turn on the camera
and it was almost to come off of its track . Eight months later, the flash
quit working in all modes AND the door was
failing AGAIN!</segment_text>
                <segment_score>-81</segment_score>
             </segment>
          </item>
           <item topic="picture quality"> <score> -75 </score>
           </item>
      </sentiment>
    </hit>
    <hit>
       <sentiment>
          <item topic="lens">
             <segment>
                 <segment_text>I was actually quite impressed with it.
Powerful zoom , sharp <b>lens</b>, decent picture
quality. I also played with some other Panasonic models in various stores
just to get a better feel for them, as well as
spent a few hours on </segment_text>
                  <segment_score> 67 </segment_score>
            </segment>
          </item>
             <item topic="picture quality">  <score>-1</score>    </item>
       </sentiment>
    </hit>
    . . .
  . . .
  </hitlist>
</ctx_result_set>
```

See Also:

Oracle Text Reference

18
Working with Sharded Databases

Learn how to run Oracle Text PL/SQL procedures in Oracle Globally Distributed Database (sharded database).

18.1 Running Oracle Text PL/SQL APIs in a Sharded Database

Oracle Globally Distributed Database enables you to horizontally partition data across multiple, independent Oracle databases. Each physical database in such a configuration is called a shard. You can use the sharding-specific PL/SQL procedure `SYS.EXEC_SHARD_PLSQL` to propagate certain Oracle Text `CTXSYS` procedures across all shards.
You use the `SYS.EXEC_SHARD_PLSQL` wrapper to run a target procedure on all shards in the same way as DDL statements are run in a sharded database configuration. These procedures are propagated to all shards, tracked by the catalog, and run whenever a new shard is added to a configuration.

If you run a target procedure without using the `SYS.EXEC_SHARD_PLSQL` wrapper, then the procedure runs only on the shard catalog and is not propagated to all shards.

The `SYS.EXEC_SHARD_PLSQL` procedure takes a single CLOB argument, which is a character string specifying a fully qualified procedure name and its arguments.

For example, to run `CTXSYS.CTX_DDL.CREATE_PREFERENCE` on all shards:

```
exec sys.exec_shard_plsql('ctxsys.ctx_ddl.create_preference(
    preference_name => "mylexer",
    object_name => "BASIC_LEXER")');
```

Note the following:

- Certain PL/SQL procedures need a wrapper and others do not. For a complete list of allowed PL/SQL procedures, see Supported APIs in a Sharded Database.
- You must use double quotation marks (") inside a target procedure call specification, because the call specification itself is a string parameter to `SYS.EXEC_SHARD_PLSQL`.
- Running the `SYS.EXEC_SHARD_PLSQL` procedure without specifying a fully-qualified name (for example, `CTXSYS.CTX_DDL.CREATE_PREFERENCE`) results in an error.

Related Topics

- *Oracle Globally Distributed Database Guide*

18.2 Supported APIs in a Sharded Database

This is a list of all the Oracle Text `CTXSYS` procedures that you can run with or without using the sharding-specific PL/SQL procedure `SYS.EXEC_SHARD_PLSQL`.

Supported With the SYS.EXEC_SHARD_PLSQL Wrapper

You can run the following procedures from the `CTXSYS` package using the `SYS.EXEC_SHARD_PLSQL` wrapper:

- `CTX_DDL.ADD_ATTR_SECTION`
- `CTX_DDL.ADD_FIELD_SECTION`
- `CTX_DDL.ADD_MDATA_COLUMN`
- `CTX_DDL.ADD_MDATA_SECTION`
- `CTX_DDL.ADD_NDATA_SECTION`
- `CTX_DDL.ADD_SUB_LEXER`
- `CTX_DDL.ADD_STOPWORD`
- `CTX_DDL.ADD_STOPCLASS`
- `CTX_DDL.ADD_SDATA_SECTION`
- `CTX_DDL.ADD_SDATA_COLUMN`
- `CTX_DDL.ADD_STOP_SECTION`
- `CTX_DDL.ADD_STOPTHEME`
- `CTX_DDL.ADD_SPECIAL_SECTION`
- `CTX_DDL.ADD_ZONE_SECTION`
- `CTX_DDL.CREATE_PREFERENCE`
- `CTX_DDL.DROP_PREFERENCE`
- `CTX_DDL.CREATE_POLICY`
- `CTX_DDL.CREATE_SECTION_GROUP`
- `CTX_DDL.CREATE_SHADOW_INDEX`
- `CTX_DDL.CREATE_STOPLIST`
- `CTX_DDL.DROP_STOPLIST`
- `CTX_DDL.DROP_SECTION_GROUP`
- `CTX_DDL.DROP_POLICY`
- `CTX_DDL.DROP_SHADOW_INDEX`
- `CTX_DDL.EXCHANGE_SHADOW_INDEX`
- `CTX_DDL.OPTIMIZE_INDEX`
- `CTX_DDL.REMOVE_STOPWORD`
- `CTX_DDL.REMOVE_SECTION`
- `CTX_DDL.RECREATE_INDEX_ONLINE`

- CTX_DDL.SET_ATTRIBUTE
- CTX_DDL.SET_SECTION_ATTRIBUTE
- CTX_DDL.SET_SEC_GRP_ATTR
- CTX_DDL.SYNC_INDEX
- CTX_DDL.UPDATE_POLICY
- CTX_QUERY.REMOVE_SQE
- CTX_QUERY.STORE_SQE

Supported Without the SYS.EXEC_SHARD_PLSQL Wrapper

You can run the following CTXSYS procedures on the catalog without using the SYS.EXEC_SHARD_PLSQL wrapper. These APIs work directly on the catalog because they only access the metadata stored in the catalog and do not need to be propagated to shards for accurate results.

- CTX_ANL.ADD_DICTIONARY
- CTX_DOC.POLICY_SNIPPET
- CTX_ENTITY.EXTRACT
- CTX_REPORT.CREATE_INDEX_SCRIPT
- CTX_REPORT.DESCRIBE_INDEX
- CTX_REPORT.CREATE_POLICY_SCRIPT
- CTX_REPORT.INDEX_SIZE
- CTX_REPORT.DESCRIBE_POLICY
- CTX_QUERY.HFEEDBACK
- CTX_QUERY.EXPLAIN

Not Supported With the SYS.EXEC_SHARD_PLSQL Wrapper

The following CTXSYS procedures are not supported on a sharded database:

- CTX_ANL APIS
- CTX_CLS APIS
- CTX_DDL.ADD_INDEX
- CTX_DDL.CREATE_INDEX_SET
- CTX_DDL.DROP_INDEX_SET
- CTX_DDL.POPULATE_PENDING
- CTX_DDL.REMOVE_INDEX
- CTX_DDL.REMOVE_MDATA
- CTX_DDL.UPDATE_SDATA
- CTX_DOC.HIGHLIGHT
- CTX_DOC.MARKUP
- CTX_DOC.FILTER

- CTX_DOC.GIST
- CTX_DOC.POLICY_MARKUP
- CTX_DOC.POLICY_SNIPPET
- CTX_DOC.SNIPPET
- CTX_ENTITY.EXTRACT
- CTX_QUERY.COUNT_HITS
- CTX_QUERY.EXPLAIN
- CTX_QUERY.HFEEDBACK
- CTX_QUERY.RESULT_SET
- CTX_REPORT.DESCRIBE_POLICY
- CTX_REPORT.TOKEN_INFO
- CTX_REPORT.INDEX_STATS
- CTX_THES APIS

19
Administering Oracle Text

Become familiar with Oracle Text administration.

This chapter contains the following topics:

19.1 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.

This section contains the following sections:

19.1.1 CTXSYS User

The `CTXSYS` user is created during installation and can:

- View all indexes
- Sync all indexes
- Run `ctxkbtc`, the knowledge base extension compiler
- Query all system-defined views
- Perform all tasks of a user with the `CTXAPP` role

> **Note:**
>
> In earlier releases of Oracle Text, CTXSYS had SYSDBA privileges, and only CTXSYS users could perform certain functions, such as modifying system-defined preferences or setting system parameters.

Starting with Oracle Database Release 19c, the CTXSYS user is a schema only user. To use the CTXSYS schema, run the following statements:

```
connect / as sysdba;

alter session set CURRENT_SCHEMA=CTXSYS;
```

19.1.2 CTXAPP Role

The CTXAPP role is a system-defined role that enables users to:

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

19.1.3 Granting Roles and Privileges to Users

The system uses the standard SQL model for granting roles to users. To grant an Oracle 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 EXECUTE privileges for the Oracle Text package to each user.

> **See Also:**
>
> "Creating an Oracle Text User"

19.2 DML Queue

When you make inserts, updates, or deletes to documents in your base table, the data manipulation language (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.

You can query pending insert, update, and delete operations using the DR$*index_name*$C table.

You can query insert, update, and delete errors with the CTX_INDEX_ERRORS or CTX_USER_INDEX_ERRORS view.

Related Topics

- Viewing Pending DML Operations
  When you insert, update, or delete documents in the base table, their ROWIDs are held in a DML queue until you synchronize the index.

19.3 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

19.4 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 and 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, the settings of the attributes of the objects, and (for `CTX_REPORT.DESCRIBE_INDEX`) the 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
   stopword:                      teststopword
storage:                        BASIC_STORAGE
   r_table_clause:                 lob (data) store as (cache)
   i_index_clause:                 compress 2
```

CTX_REPORT.CREATE_INDEX_SCRIPT and CTX_REPORT.CREATE_POLICY_SCRIPT

`CREATE_INDEX_SCRIPT` creates a SQL*Plus script that can create a duplicate of a given Oracle Text index. Use this when you have an index but you do not have the original script (if any) that was used to create this index, and you 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 of 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 in the index, average size of its tokens, and fragmentation information for the index. Optimizing stoplists is an example of a use for `INDEX_STATS`.

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.

CTX_REPORT.TOKEN_INFO

`TOKEN_INFO` helps you diagnose query problems. For example, use it to check that index data is not corrupted and to find out which documents are producing unexpected or bad tokens.

CTX_REPORT.TOKEN_TYPE

`TOKEN_TYPE` is a lookup function that is used mainly as input to other functions (`CTX_DDL.OPTIMIZE_INDEX`, `CTX_REPORT.TOKEN_INFO`, and so on).

> **See Also:**
>
> - *Oracle Text Reference* for an example of the output of `CTX_REPORT.INDEX_STATS` procedure
> - *Oracle Text Reference* for an example of the output of `CTX_REPORT.QUERY_LOG_SUMMARY` procedure

19.5 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 Oracle 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 you need to take.

The Text Indexes page shows the jobs that are in progress, that are scheduled within the last seven days, or that 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.

This section contains the following sections:

- Using Text Manager
- Viewing General Information for an Oracle Text Index
- Checking Oracle Text Index Health

> **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 Indexing with Oracle Text under Creating Oracle Text Indexes.

19.5.1 Using Text Manager

You can access Text Manager to manage Oracle Text indexes or schedule jobs for a specific index.

On 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

1. Sign in to the database with a user account that is authorized to access Cloud Control. For example, use `SYS` or `SYSTEM` and the password that you specified during database installation.
2. On the **Database Home** page, click the **Schema** tab.
3. In the **Text Manager** group, select **Text Indexes**.

   The **Text Indexes** page displays a list of Oracle Text indexes for this database instance.

When you select an Oracle Text index from the Text Indexes page, edit and action options become available for that index. For example, to configure attributes for searching, click **Edit** for the selected index. On the Edit Text Index page, you can set such attributes as Wild Card Maximum Term, Fuzzy Score, and Number of Fuzzy Expansions. You can also change index and partition names, and specify settings for `NETWORK_DATASTORE`.

19.5.2 Viewing General Information for an Oracle Text Index

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

To view general information for an Oracle Text index, on the **Text Indexes** page, in the list of indexes, click the name of the index. The **View Text Index** page opens and the **General** tab is selected. From here, you can select actions to perform maintenance tasks.

19.5.3 Checking Oracle Text Index Health

In Text Manager, the Text Indexes page displays the Oracle Text indexes for the database instance. Use that page to help you understand the critical actions that are necessary to make sure that the entire application is performing properly.

Use the Text Indexes page to see:

- The status of the indexes and jobs submitted during the last seven days.
- The number of Oracle Text indexes that contain invalid partitions, and which are, therefore, invalid. The number of partitions that are invalid, if any, for all Oracle Text indexes is also shown.
- The number of indexes and partitions that are in an in-progress state.

- The number of indexes where all partitions are valid, and no activity is in progress.
- The sum total of the Oracle Text indexes found for this database instance.
- The index type for each Oracle Text index, the owner, the number of documents that are not synchronized, total number of documents, and percentage of fragmentation.

After you select an Oracle Text index from the list, options become available for editing or performing actions.

19.6 Servers and Indexing

You index documents and enter queries with standard SQL. No server is needed for performing batch insert, update, and delete operations. You can synchronize the `CONTEXT` index with the `CTX_DDL.SYNC_INDEX` procedure, or from Text Manager in Oracle Enterprise Manager.

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

19.7 Tracking Database Feature Usage in Oracle Enterprise Manager

In Oracle Enterprise Manager, Database Feature Usage statistics provide an approximation of how often various database features are used. Tracking this information is useful for application development and for auditing.

To access Database Feature Usage, in Oracle Enterprise Manager, click the **Server** tab, and then select **Database Feature Usage** under **Database Configuration.**

Database Feature Usage captures the following information for Oracle Text:

- **Index Usage Statistics:** The number of existing indexes in the database for the `CONTEXT`, `CTXCAT`, and `CTXRULE` index types
- **SQL Operator Usage Statistics:** Whether the user has ever used the `CONTAINS`, `CATSEARCH`, and `MATCHES` operators
- **Package Usage Statistics:** How often, if ever, and when the following packages were used:
  - `CTX_ADM`
  - `CTX_CLS`
  - `CTX_DDL`
  - `CTX_DOC`
  - `CTX_OUTPUT`
  - `CTX_QUERY`
  - `CTX_REPORT`
  - `CTX_THES`

- CTX_ULEXER

> **Note:**
>
> The feature usage tracking statistics might not be 100 percent accurate.

19.8 Oracle Text on Oracle Real Application Clusters

For maximum throughput and performance of applications, you can parallelize Oracle Text queries across Oracle Real Application Clusters (Oracle RAC) nodes.

You can manage Oracle Text indexes on Oracle RAC nodes with Text Manager in Oracle Enterprise Manager, as described in Text Manager in Oracle Enterprise Manager.

Related Topics

- Parallelizing Queries Across Oracle RAC Nodes
  Oracle Real Application Clusters (Oracle RAC) enables you to improve query throughput and scalability as the query load increases.

19.9 Configuring Oracle Text in Oracle Database Vault Environment

In an Oracle Database Vault environment, you can create a `CTXSYS` user if you have the `DV_ACCTMGR` role.

To create a `CTXSYS` user, run the `@$ORACLE_HOME/ctx/admin/catctx_user.sql` SQL script. Then, connect as `SYS` user and run the `@$ORACLE_HOME/ctx/admin/catctx_schema.sql` SQL script.

> **Note:**
>
> If the `SYS` user also has the `DV_ACCTMGR` role, then you can run the `@$ORACLE_HOME/ctx/admin/catctx.sql` SQL script which installs both, `catctx_user.sql` and `catctx_schema.sql` scripts.

19.10 Unsupported Oracle Text Operations in Oracle Database Vault Realm

Oracle Database Vault realms place restrictions on DDL operations within a realm. For this reason, once you are added to a realm but if you are not authorized in the realm, then you cannot create, alter, or drop an Oracle Text index. You also cannot use any DDL operations contained in the `CTX_DDL` package.

The DDL error messages and query error messages on indexes that could not be created within the realm might indicate insufficient privileges as the cause. The insufficient privilege message is specific to DDL operations not being allowed within the realm.

19.11 Export and Import of Schemas Containing Oracle Text Settings

Before Oracle Database Release 21c, schema objects like preferences, section groups, stoplists, and other Oracle Text preferences were not exported or imported. Starting with Oracle Database Release 21c, they are copied when you export and import the schema by using Data Pump Export and Import utilities (invoked with the `expdp` and `impdp` commands, respectively).

20

Migrating Oracle Text Applications

You can migrate Oracle Text applications into a new Oracle Database release.

When you upgrade to a new release of Oracle Database, you may have difficulty migrating your applications from earlier releases of Oracle Text. Where applicable, Oracle provides information about the migration steps to move Oracle Text applications into the new release.

This chapter contains the following topics:

- Oracle Text and Rolling Upgrade with Logical Standby
- Identifying and Copying Oracle Text Files to a New Oracle Home

> **See Also:**
>
> *Oracle Database Upgrade Guide* for information on upgrading Oracle Database and topics about migrating applications

20.1 Performing a Rolling Upgrade with a Logical Standby Database

You can use a logical standby database to perform a rolling upgrade of Oracle Database. To incur minimal downtime on the primary database, you can run different releases of Oracle Database on the primary and logical standby databases while you upgrade your databases, one at a time. Oracle Text takes full advantage of upgrading Oracle Text indexes.

All CTX PL/SQL procedures are fully replicated to the standby database and are upgraded, except with certain limitations for these procedures:

- CTX_DDL PL/SQL Procedures
- CTX_OUTPUT PL/SQL Procedures
- CTX_DOC PL/SQL Procedures

> **See Also:**
>
> *Oracle Data Guard Concepts and Administration* for information on creating a logical standby database to perform rolling upgrades

20.1.1 CTX_DDL PL/SQL Procedures

Oracle Database uses rowids internally for the construction of indexes. The following `CTX_DDL` procedures are not fully replicated to the standby:

- ADD_MDATA
- REMOVE_MDATA

20.1.2 CTX_OUTPUT PL/SQL Procedures

Only CTX_OUTPUT.ENABLE_QUERY_STATS and CTX_OUTPUT.DISABLE_QUERY_STATS are replicated. If you enable Oracle Text logging on the primary database before you run an operation that causes logging, then the operation runs with logging on the primary database and without logging on the secondary database.

20.1.3 CTX_DOC PL/SQL Procedures

When you use the following CTX_DOC procedures with Oracle Text Result Tables, the data stored in the tables is replicated. When these procedures are used without Result Tables, they are not replicated.

- CTX_DOC.SET_KEY_TYPE
- CTX_DOC.FILTER
- CTX_DOC.GIST
- CTX_DOC.MARKUP
- CTX_DOC.TOKENS
- CTX_DOC.THEMES
- CTX_DOC.HIGHLIGHT
- CTX_DOC.FILTER_CLOB_QUERY
- CTX_DOC.MARKUP_CLOB_QUERY
- CTX_DOC.HIGHLIGHT_CLOB_QUERY

> **See Also:**
>
> *Oracle Data Guard Concepts and Administration* for information on performing a rolling upgrade for minimal downtime on the primary database

20.2 Identifying and Copying Oracle Text Files to a New Oracle Home

To upgrade Oracle Text, use this procedure to identify and copy required files from your existing Oracle home to the new release Oracle home. Complete this task after you upgrade Oracle Database.

Certain Oracle Text features rely on files under the Oracle home that you have configured. After manually upgrading to a new Oracle Database release, or after any process that changes the Oracle home, you must identify and move these files manually. These files include user filters, mail filter configuration files, and all knowledge base extension files. After you identify the files, copy the files from your existing Oracle home to the new Oracle home.

To identify and copy required files from your existing Oracle home to the new release Oracle home:

1. Log in with the `SYS`, `SYSTEM`, or `CTXSYS` system privileges for the upgraded database.
2. Under the Oracle home of the upgraded database, run the `$ORACLE_HOME/ctx/admin/ctx_oh_files.sql` SQL script.

   For example:

   ```
   sqlplus / as sysdba
   connected
   SQL> @?/ctx/admin/ctx_oh_files
   ```

3. Review the output of the `ctx_oh_files.sql` command, and copy the files to the new Oracle home.

A
CONTEXT Query Application

This appendix describes how to build a simple web search application by using the `CONTEXT` index type.

This appendix contains the following topics:

- Web Query Application Overview
- The PL/SQL Server Pages (PSP) Web Application
- The Java Server Pages (JSP) Web Application

A.1 Web Query Application Overview

A common use of Oracle Text is to index HTML files on websites and provide search capabilities to users. The sample application in this appendix indexes a set of HTML files stored in the database. It also uses a web server connected to Oracle Database to provide the search service.

This appendix describes two versions of the Web query application:

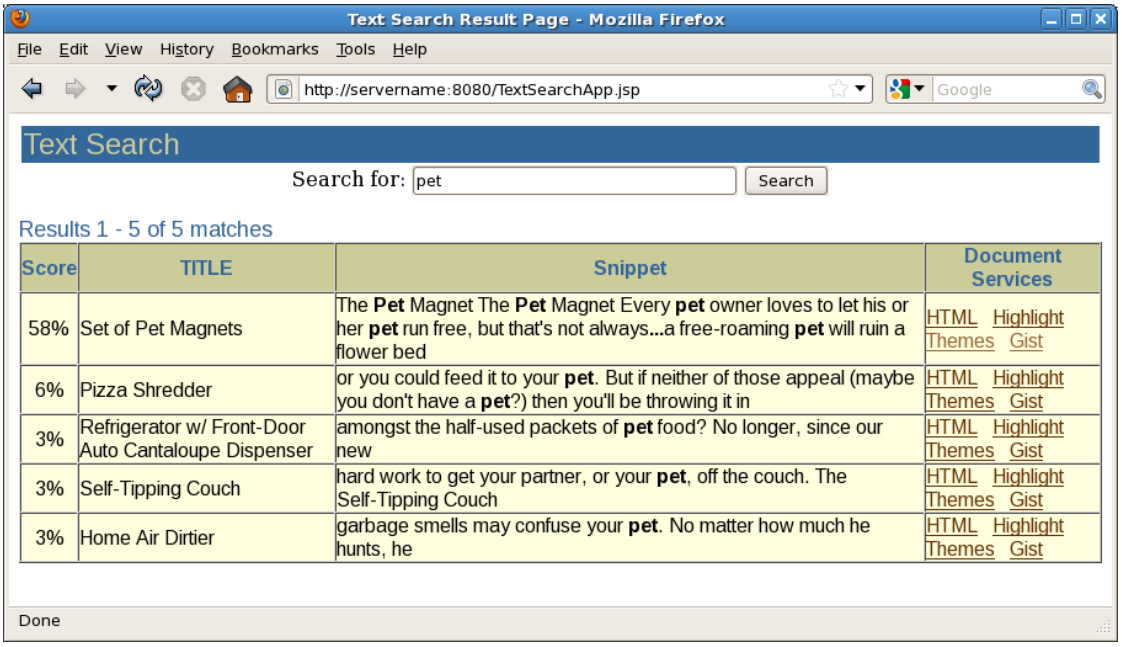
- One using PL/SQL Server Pages (**PSP**)
- One using Java Server Pages (**JSP**)

Figure A-1 shows the JSP version of the text query application.

Figure A-1 The Text Query Application

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

Figure A-2 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.
- The **Highlight** link displays the document with the search term highlighted.
- The **Theme** link shows the top 50 themes associated with the document.

  The **Gist** link displays a short summary of the document.

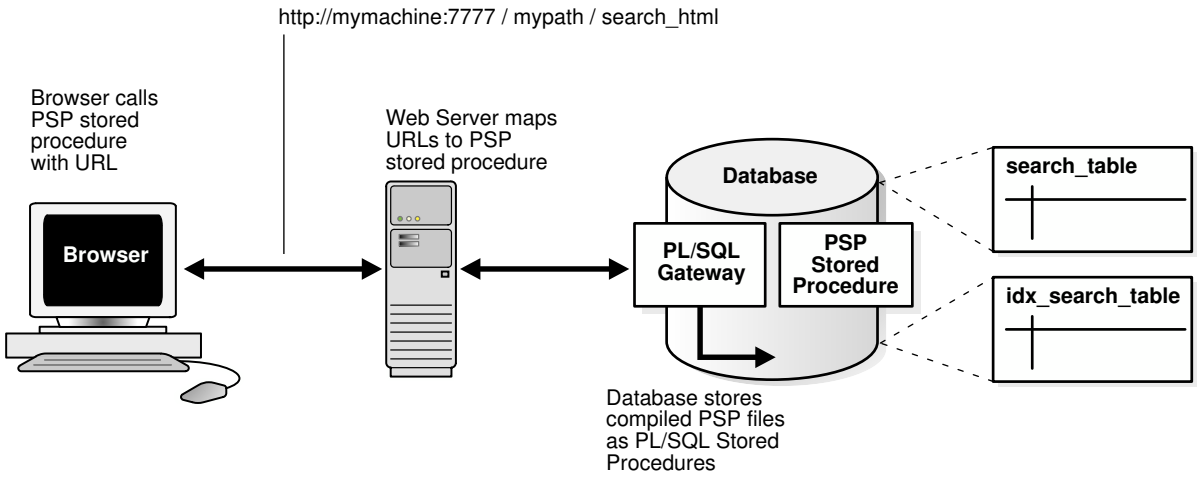
A.2 The PL/SQL Server Pages (PSP) Web Application

The PSP web application is based on PL/SQL server pages. Figure A-3 illustrates how the browser calls the PSP-stored procedure on Oracle Database through a web server.

This section contains the following topics:

- PSP Web Application Prerequisites
- Building the PSP Web Application
- PSP Web Application Sample Code

Figure A-3 The PSP Web Application



A.2.1 PSP Web Application Prerequisites

The PSP web application has the following requirements.

- Your Oracle Database must be up and running.

  For a connection example, see *Oracle Database JDBC Developer's Guide*.

- The SCOTT account is unlocked with its password, and the account has `CREATE`, `RESOURCE`, and `CTXAPP` privileges.

- The Oracle PL/SQL gateway must be running.

  For complete information about setting up the PL/SQL gateway and developing PL/SQL web applications, see *Oracle Database Development Guide*.

- A web server such as Apache is up and running and is correctly configured to send requests to Oracle Database.

  For information about installing Apache HTTP Server, see *Oracle Database 2 Day + PHP Developer's Guide*.

A.2.2 Building the PSP Web Application

To create PSP web application:

1. Create your text tables.

   You must create text tables with the `CREATE TABLE` command to store your HTML files. These examples create the `output_table`, `gist_table`, and `theme_table` tables:

   ```
   CREATE TABLE output_table  (query_id NUMBER, document CLOB);
   CREATE TABLE gist_table  (query_id NUMBER, pov VARCHAR2(80), gist CLOB);
   CREATE TABLE theme_table  (query_id NUMBER, theme VARCHAR2(2000), weight NUMBER);
   ```

2. Load HTML documents into the table by using SQL*Loader.

   You must load the text tables with the HTML files. This example uses the loader.ctl control file to load the files named in loader.dat. The SQL*Loader statement is as follows:

   ```
   % sqlldr userid=scott/password control=loader.ctl
   ```

3. Create the `CONTEXT` index.

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

   ```
   create index idx_search_table on search_table(text)
     indextype is ctxsys.context parameters
     ('filter ctxsys.null_filter section group CTXSYS.HTML_SECTION_GROUP');
   ```

4. Compile the `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 character large object `(CLOB)` in `search_table` and output the result for viewing. To do that, call procedures in the `search_htmlservices` package. Compile the file search_htmlservices.sql file at the SQL*Plus prompt as follows:

   ```
   SQL> @search_htmlservices.sql

   Package created.
   ```

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 as follows:

   ```
   % loadpsp -replace -user scott/password search_html.psp
   ```

   The output appears as:

   ```
   "search_html.psp": procedure "search_html" created.
   ```

   > **See Also:**
   >
   > Oracle Database 11g Release 2 (11.2) of Oracle Database Development Guide for more information about using PSP

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.

   You can use the Oracle WebDB web listener or Oracle Application Server, which includes the Apache web server.

7. Enter the query from a browser.

   You can access the query application from a browser by using a URL. You configure the URL with your web server. An example URL might look like the following:

   ```
   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 and Figure A-2.

A.2.3 PSP Web Application 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

A.2.3.1 loader.ctl

This example shows a sample `loader.ctl` file. It is used by `sqlldr` to load the `loader.dat` data file.

```
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)
```

A.2.3.2 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
```

A.2.3.3 HTML Files for loader.dat Example

The HTML files that are named and loaded into `loader.dat` are included here for your reference.

- Pizza.html
- Cantaloupe.html
- Couch.html
- Mess.html
- Pet.html
- Snooze.html

Pizza.html

```
<html>
<header>
<title>The Pizza Shredder</title>
</header>
<body>

<h2>The Pizza Shredder</h2>
<h4>Keeping your pizza preferences secure</h4>
```

```
So it's the end of a long evening. Beer has been drunk, pizza has been eaten.
<p>
But there's leftover pizza - what are you going to do with it?

<p>
You could save it for the morning, or you could feed it to your pet.  But if neither of
those appeal (maybe you don't have a pet?) then
you'll be throwing it in the trash.

<p>
But wait a minute - anybody could look through your trash, and figure out what kind of
pizza you've been eating! "No big deal," I hear you
say. But it is! After they've figured out that your favorite pizza is pepperoni, then
it's only a short step to figuring out that
your top-secret online banking password is "pepperoni_pizza."

<p>
Get one over the dumpster-divers with our new patent-pending "Mk III Pizza Shredder."
Cross-cut blades ensure that your pizza will be rendered
unreadable, and nobody will be able to identify the original toppings. Also doubles as a
lettuce-shredder and may also be used for removing
unwanted fingertips.

<h2>Model Comparison</h2>

<table border="1">
  <tr><th>Model</th><th>Blades0</th><th>Pizza Thickness</th><th>Price</th></tr>
  <tr><td>Mk I</td><td>Plastic</td><td>1/2 inch (Thin Crust)</td><td>$69.99</td></tr>
  <tr><td>Mk II</td><td>Brass</td><td>1 inch (Deep Pan)</td><td>$99.99</td></tr>
  <tr><td>Mk III</td><td>Carbon Steel</td><td>2 inch (Calzoni)</td><td>$129.99</td></tr>
</table>

</body>
</html>
```

Cantaloupe.html

```
<html>
<header>
<title>The Fridge with a Cantaloupe Dispenser</title>
</header>
<body>
<h2>The Fridge with a Cantaloupe Dispenser</h2>
<h4>A nice cold melon at the touch of a button</h4>

Does your refrigerator only have a boring water dispenser in the door?

<p>
When you're hungry for a cantaloupe, do you have to expend valuable energy opening the
fridge door and fishing around amongst the half-used
 packets of pet food?

<p>
Do your friends complain that they wish there was an effortless way to get cantaloupes
from your fridge? Do you overhear them saying they're
tired of always having to rummage through your moldy leftovers and seal-a-meals to get
to the cold melons?

<p>
What you need is the convenience of a built-in cantaloupe dispenser.
```

```
<p>
Impress your friends. Win praise from your neighbors. Become a legendary host!

<p>
<b>Try our new <i>Melonic 2000</i> model!</b>

<p>
Works with honeydews and small crenshaws too.

<p>
Let the <i>Melonic 2000</i> go to work for you. Order one now at your local store.

</body>
</html>
```

Couch.html

```
<html>
<header>
<title>The Self-Tipping Couch</title>
</header>
<body>
<h2>The Self-Tipping Couch</h2>

<h4>Sometimes it's hard work to get off the couch</h4>

<p>
Sometimes it's hard work to get your partner, or your pet, off the couch.

<p>
The <b>Self-Tipping Couch</b> solves these problems for you. At the touch of a button it
will deposit the contents of the couch onto the
 floor in front of it.

<p>
The <b>Self-Tipping Couch</b> has been proven to boost communication with stubborn
spouses, children, and relatives.

<p>
You will never again need to yell, "Get off the couch!" Simply press a button and all
those couch hoggers are gently
dumped onto your carpet.

<p>
Get your own <b>Self-Tipping Couch</b> TODAY!

</body>
</html>
```

Mess.html

```
<html>
<header>
<title>Home Air Dirtier</title>
</header>
<body>
<h2>Home Air Dirtier</h2>
<h4>Missing your home in the middle of the city?</h4>

<p>
Like many ex-city-dwellers, you might be finding that the air in the countryside is just
too clean.
```

```
<p>
You can remedy this right now with the <i>UltraAppliance</i> <b>Home Air Dirtier</b>.

<p>
Simply insert our patented <i>CityFilth</i> cartridge,
and soon you'll be enjoying the aromas of vehicle fumes and decaying garbage that you're
used to from home.

<p>
<b>Please note:</b> Decaying garbage smells may confuse your pet.
We recommend adding genuine garbage to your environment if this is a concern.

</body>
</html>
```

Pet.html

```
<html>
<header>
<title>The Pet Magnet</title>
</header>
<body>
<h2>The Pet Magnet</h2>

<h4>Every pet owner loves to let the pet run free, but that's not always possible</h4>

<p>
Sometimes local laws require pets to be on leashes. Sometimes a free-roaming pet will
ruin a flower bed, leave a "calling card" on the
sidewalk, or chew through another pet. In the case of extremely smart pets, like
chimpanzees or dolphins, the unattended pet may get
away and run up hundreds of dollars of long-distance charges on your phone.

<p>
But leashes aren't always a practical answer. They can be too confining, or too big, or
can tug uncomfortably at the pet's neck. They
may get tangled, or wrapped around poles or passersby. Pets may chew through the leash,
or, again, in the case of extremely smart pets,
burn through it with an acetylene torch. In the case of cats, leashes simply look
ridiculous, as though the pet owner really wanted to
own a dog but got confused at the pet store.

<p>
The <b>Hold 'Em 2000 Pet Magnet</b> from <i>UltraAppliance</i> is the answer. Instead of
old-fashioned leashes, the
<b>Hold 'Em 2000 Pet Magnet</b> keeps your pet under control in a simple way.

<p>
Here's how it works. Dozens of small magnets are placed underneath the coat of your pet,
where they remain painlessly invisible. Any time
you need to recall your animal, you merely activate the handy, massive Hold 'Em 2000 Pet
Magnet electromagnet (fits inside any extremely
oversized purse) and your pet is gently and painlessly dragged to you from up to 100
yards. It's a must-have for any pet owner!

<p>

<blockquote>
<i>
"The <b>Hold 'Em 2000 Pet Magnet</b> not only keeps my dog from running away, but the
electromagnet also comes in very handy if I need to
```

```
find a needle in a haystack"</i>
-- Anonymous Celebrity
</blockquote>
</body>
</html>
```

Snooze.html

```
<html>
<header>
<title>Esteem-building Talking Pillow</title>
</header>
<body>
<h2>Esteem-building Talking Pillow</h2>
<h4>Do you feel less than your true potential when you wake up in the morning?</h4>

<p>
We searched for a way to capture the wasted time spent sleeping and to use this precious
time to build motivation, character, and self-esteem.

<p>
We are proud to announce the <b>Esteem-building Talking Pillow</b>. Our pride in this
wonderful invention glows even more because:
<i>We use our own invention every night!</i>

<p>
Only you will know that you are sleeping with the <b>Esteem-building Talking Pillow</b>
because only you can hear the soothing
affirmations that gently enter your brain through the discreet speaker.

<p>
You will wake up refreshed and raring to go with a new sense of pride and enthusiasm for
any task the day may bring.

<p>
Be the first to own the <b>Esteem-building Talking Pillow</b>! Your friends and fellow
workers will be amazed when you no longer
cower in the corner. Now you will join in every conversation.

<p>
<b>Disclaimer:</b> Not responsible for narcissism and hyberbolic statements. May cause
extreme behavior with overuse.

</body>
</html>
```

A.2.3.4 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 search_htmlServices;
/
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 search_htmlServices;
```

```
/
show errors

set define on
```

A.2.3.5 search_html.psp

```
<%@ plsql procedure="search_html" %>
<%@ plsql parameter="query" default="null" %>
<%! v_results number := 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>
```

A.3 The Java Server Pages (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 PSP Web Application" for more information. 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`

This section contains the following topics:

- JSP Web Application Prerequisites
- JSP Web Application Sample Code

A.3.1 JSP Web Application Prerequisites

The JSP web application has the following requirements:

- Your Oracle Database must be up and running.

- You have a web server such as Apache Tomcat, which can run JavaServer Pages (JSP) scripts that connect to the Oracle Database by using Java Database Connectivity (JDBC).

> **See Also:**
>
> *Oracle Database 2 Day + PHP Developer's Guide* for information about installing Apache HTTP Server

A.3.2 JSP Web Application Sample Code

This section lists the Java code used to build the example web application, as shown in the `TextSearchApp.jsp` file.

```
<%@page language="java" pageEncoding="utf-8" contentType="text/html; charset=utf-8" %>
<%@ page import="java.sql.*, java.util.*, java.net.*,
   oracle.jdbc.*, oracle.sql.*, oracle.jsp.dbutil.*" %>

<%
// Change these details to suit your database and user details

String connStr = "jdbc:oracle:thin:@//servername:1521/pdb1";
String dbUser  = "scott";
String dbPass  = "tiger";

// The table we're running queries against is called SEARCH_TABLE.
// It must have columns:
//  tk     number  primary key,     (primary key is important for document services)
//  title  varchar2(2000),
//  text   clob
// There must be a CONTEXT index called IDX_SEARCH_TABLE on the text column

request.setCharacterEncoding("UTF-8");

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;
String action    = null;
String theTk     = null;
URLEncoder myEncoder;
int count=0;
int loopNum=0;
int startNum=0;

userQuery     =   request.getParameter("query");
action        =   request.getParameter("action");
theTk         =   request.getParameter("tk");

if (action == null)  action = "";

// Connect to database

try {
  DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver() );
  info.put ("user",     dbUser);
  info.put ("password", dbPass);
  conn      = DriverManager.getConnection(connStr,info);
}
  catch (SQLException e) {
%>    <b>Error: </b> <%= e %><p>  <%
```

```
  }

if ( action.equals("doHTML") ) {
  // Directly display the text of the document
  try {

    // not attempting to share the output table for this example, we'll truncate it each time
    conn.createStatement().execute("truncate table OUTPUT_TABLE");

    String sql = "{ call ctx_doc.filter( index_name=>'IDX_SEARCH_TABLE', textkey=> '" + theTk + "',
restab=>'OUTPUT_TABLE',
                  plaintext=>false ) }";
    PreparedStatement s = conn.prepareCall( sql );
    s.execute();

    sql = "select document from output_table where rownum = 1";
    stmt = conn.createStatement();
    rset = stmt.executeQuery(sql);

    rset.next();
    oracle.sql.CLOB res = (oracle.sql.CLOB) rset.getClob(1);
    // should fetch from clob piecewise, but to keep it simple we'll just fetch 32K to a string
    String txt = res.getSubString(1, 32767);
    out.println(txt);
  }
  catch (SQLException e) {
%>    <b>Error: </b> <%= e %><p> <%
  }
}
else if ( action.equals("doHighlight") ) {
  // Display the text of the document with highlighting from the "markup" function
  try {

    // not attempting to share the output table for this example, we'll truncate it each time
    conn.createStatement().execute("truncate table OUTPUT_TABLE");

    String sql = "{ call ctx_doc.markup( index_name=>'IDX_SEARCH_TABLE', textkey=> '" + theTk + "',
text_query => '" + userQuery + "',
                  restab=>'OUTPUT_TABLE', plaintext=>false, starttag => '<i><font color=\"red\">',
endtag => '</font></i>' ) }";
    PreparedStatement s = conn.prepareCall( sql );
    s.execute();

    sql = "select document from output_table where rownum = 1";
    stmt = conn.createStatement();
    rset = stmt.executeQuery(sql);

    rset.next();
    oracle.sql.CLOB res = (oracle.sql.CLOB) rset.getClob(1);
    // should fetch from clob piecewise, but to keep it simple we'll just fetch 32K to a string
    String txt = res.getSubString(1, 32767);
    out.println(txt);
  }
  catch (SQLException e) {
%>    <b>Error: </b> <%= e %><p> <%
  }
}

else if ( action.equals("doThemes") ) {
  // Display the text of the document with highlighting from the "markup" function
  try {

    // not attempting to share the output table for this example, we'll truncate it each time
    conn.createStatement().execute("truncate table THEME_TABLE");

    String sql = "{ call ctx_doc.themes( index_name=>'IDX_SEARCH_TABLE', textkey=> '" + theTk + "',
restab=>'THEME_TABLE') }";
    PreparedStatement s = conn.prepareCall( sql );
    s.execute();
```

```
    sql = "select * from ( select theme, weight from theme_table order by weight desc ) where
rownum <= 20";
    stmt = conn.createStatement();
    rset = stmt.executeQuery(sql);
    int    weight = 0;
    String theme  = "";
%>
    <h2>The top 20 themes of the document</h2>
    <table BORDER=1 CELLSPACING=0 CELLPADDING=0"
       <tr bgcolor="#CCCC99">
       <th><font face="arial" color="#336699">Theme</th>
       <th><font face="arial" color="#336699">Weight</th>
       </tr>
<%
    while ( rset.next() ) {

      theme  = rset.getString(1);
      weight = (int)rset.getInt(2);
%>
       <tr bgcolor="ffffe0">
         <td align="center"><font face="arial"><b> <%= theme  %> </b></font></td>
         <td align="center"><font face="arial"> <%= weight %></font></td>
       </tr>
<%
    }

%>
</table>
<%
  }
  catch (SQLException e) {
%>    <b>Error: </b> <%= e %><p> <%
  }
}
else if ( action.equals("doGists") ) {
  // Display the text of the document with highlighting from the "markup" function
  try {

    // not attempting to share the output table for this example, we'll truncate it each time
    conn.createStatement().execute("truncate table GIST_TABLE");

    String sql = "{ call ctx_doc.gist( index_name=>'IDX_SEARCH_TABLE', textkey=> '" + theTk + "',
restab=>'GIST_TABLE', query_id=>1) }";
    PreparedStatement s = conn.prepareCall( sql );
    s.execute();

    sql = "select pov, gist from gist_table where pov = 'GENERIC' and query_id = 1";
    stmt = conn.createStatement();
    rset = stmt.executeQuery(sql);
    String pov   = "";
    String gist  = "";

    while ( rset.next() ) {

      pov   = rset.getString(1);
      oracle.sql.CLOB gistClob = (oracle.sql.CLOB) rset.getClob(2);

      out.println("<h3>Document Gist for Point of View: " + pov + "</h3>");
      gist = gistClob.getSubString(1, 32767);
      out.println(gist);

    }

%>
</table>
<%
  }
  catch (SQLException e) {
%>    <b>Error: </b> <%= e %><p> <%
  }
```

```
}

if ( (action.equals("")) && ( (userQuery == null) || (userQuery.length() == 0) ) ) {
%>
  <html>
    <title>Text Search</title>
    <body>
      <table width="100%">
        <tr bgcolor="#336699">
          <td><font face="arial" 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 if (action.equals("") ) {
%>
  <html>
    <title>Text Search Result Page</title>
    <body text="#000000" bgcolor="#FFFFFF" link="#663300"
          vlink="#996633" alink="#ff6600">
      <table width="100%">
        <tr bgcolor="#336699">
          <td><font face="arial" 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="<%= userQuery %>" size = 30>
      <input type=submit value="Search">
      </form>
    </center>
<%
  myQuery   =   URLEncoder.encode(userQuery);
  try {

    stmt      = conn.createStatement();

    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=>'IDX_SEARCH_TABLE', "+
         "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" SIZE=+1>Results
          <%=startNum+1%> - <%=startNum+20%> of <%=count%> matches
<%
    }
    else if(count>0){
```

```
%>
    <font color="#336699" FACE="Arial" SIZE=+1>Results
           <%=startNum+1%> - <%=count%> of <%=count%> matches
<%
    }
    else {
%>
    <font color="#336699" FACE="Arial" SIZE=+1>No match found
<%
    }
%>
  <table width="100%">
  <TR ALIGN="RIGHT">
<%
  if((startNum>0)&(count<=startNum+20))
  {
%>
    <TD ALIGN="RIGHT">
    <a href="TextSearchApp.jsp?sn=<%=startNum-20 %>&query=
            <%=myQuery %>">previous20</a>
    </TD>
<%
  }
  else if((count>startNum+20)&(startNum==0))
  {
%>
    <TD ALIGN="RIGHT">
    <a href="TextSearchApp.jsp?sn=<%=startNum+20
          %>&query=<%=myQuery %>">next20</a>
    </TD>
<%
  }
  else if((count>startNum+20)&(startNum>0))
  {
%>
    <TD ALIGN="RIGHT">
    <a href="TextSearchApp.jsp?sn=<%=startNum-20 %>&query=
              <%=myQuery %>">previous20</a>
    <a href="TextSearchApp.jsp?sn=<%=startNum+20 %>&query=
              <%=myQuery %>">next20</a>
    </TD>
<%
  }
%>
  </TR>
  </table>
<%
    String ctxQuery =
        " select /*+ FIRST_ROWS */ " +
        "   tk, TITLE, score(1) scr, " +
        "   ctx_doc.snippet ('IDX_SEARCH_TABLE', tk, '" + theQuery + "') " +
        " from search_table " +
        " where contains(TEXT, '"+theQuery+"',1 ) > 0 " +
        " order by score(1) desc";
    rset = stmt.executeQuery(ctxQuery);
    String   tk           = null;
    String[] colToDisplay = new String[1];
    int      myScore      = 0;
    String   snippet      = "";
    int      items        = 0;
    while (rset.next()&&items< 20) {
      if(loopNum>=startNum)
      {
        tk = rset.getString(1);
        colToDisplay[0] = rset.getString(2);
        myScore         = (int)rset.getInt(3);
        snippet         = rset.getString(4);
        items++;
        if (items == 1) {
%>
```

```
        <center>
          <table BORDER=1 CELLSPACING=0 CELLPADDING=0 width="100%"
            <tr bgcolor="#CCCC99">
              <th><font face="arial" color="#336699">Score</th>
              <th><font face="arial" color="#336699">TITLE</th>
              <th><font face="arial" color="#336699">Snippet</th>
              <th> <font face="arial"
                        color="#336699">Document Services</th>
            </tr>
<%   } %>
       <tr bgcolor="#FFFFE0">
         <td ALIGN="CENTER"> <%= myScore %>%</td>
         <td> <%= colToDisplay[0] %> </td>
         <td> <%= snippet %> </td>
         <td>
           <a href="TextSearchApp.jsp?action=doHTML&tk=<%= tk %>">HTML</a>  
           <a href="TextSearchApp.jsp?action=doHighlight&tk=<%= tk %>&query=<%= theQuery
%>">Highlight</a>  
           <a href="TextSearchApp.jsp?action=doThemes&tk=<%= tk %>&query=<%= theQuery %>">Themes</a>
 
           <a href="TextSearchApp.jsp?action=doGists&tk=<%= tk %>">Gist</a>  
         </td>
       </tr>
<%
       }
       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 by using the `CATSEARCH` index type.

This appendix contains the following topics:

- CATSEARCH Web Query Application Overview
- The JSP Web Application

> **Note:**
>
> The Oracle Text indextype `CTXCAT` is deprecated with Oracle Database 23ai. The indextype itself, and it's operator `CTXCAT`, can be removed in a future release. Both `CTXCAT` and the use of `CTXCAT` grammar as an alternative grammar for `CONTEXT` queries is deprecated. Instead, Oracle recommends that you use the `CONTEXT` indextype, which can provide all the same functionality, except that it is not transactional. Near-transactional behavior in `CONTEXT` can be achieved by using `SYNC(ON COMMIT)` or, preferably, `SYNC(EVERY [time-period])` with a short time period.
>
> `CTXCAT` was introduced when indexes were typically a few megabytes in size. Modern, large indexes, can be difficult to manage with `CTXCAT`. The addition of index sets to `CTXCAT` can be achieved more effectively by the use of `FILTER BY` and `ORDER BY` columns, or `SDATA`, or both, in the `CONTEXT` indextype. `CTXCAT` is therefore rarely an appropriate choice. Oracle recommends that you choose the more efficient `CONTEXT` indextype.

B.1 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 JavaServer Pages (JSP).

B.2 The JSP Web Application

This application is based on JavaServer pages (JSP) and has the following requirements:

- Your Oracle Database must be up and running.
- A web server such as Apache Tomcat, which is can run JSP scripts that connect to the Oracle Database by using Java Database Connectivity (JDBC).

> **See Also:**
>
> *Oracle Database 2 Day + PHP Developer's Guide* for information about installing Apache HTTP Server

This section contains the following topics:

- Building the JSP Web Application
- JSP Web Application Sample Code

B.2.1 Building the JSP Web Application

This application models an online bookstore, where you can look up book titles and prices.

To create the JavaServer Pages (JSP) web application:

1. Create your table.

   You must create the table to store such book information as title, publisher, and price. From SQL*Plus:

   ```
   sqlplus>create table book_catalog (
            id        numeric,
            title     varchar2(80),
            publisher varchar2(25),
            price     numeric )
   ```

2. Load data by using SQL*Loader.

   Load the book data from the operating system command line with SQL*Loader:

   ```
   % sqlldr userid=ctxdemo/ctxdemo control=loader.ctl
   ```

3. Create the 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;
   /
   ```

4. Create the `CTXCAT` index.

   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');
   ```

5. Try a simple search by using `CATSEARCH`.

   You can test the newly created index in SQL*Plus as follows:

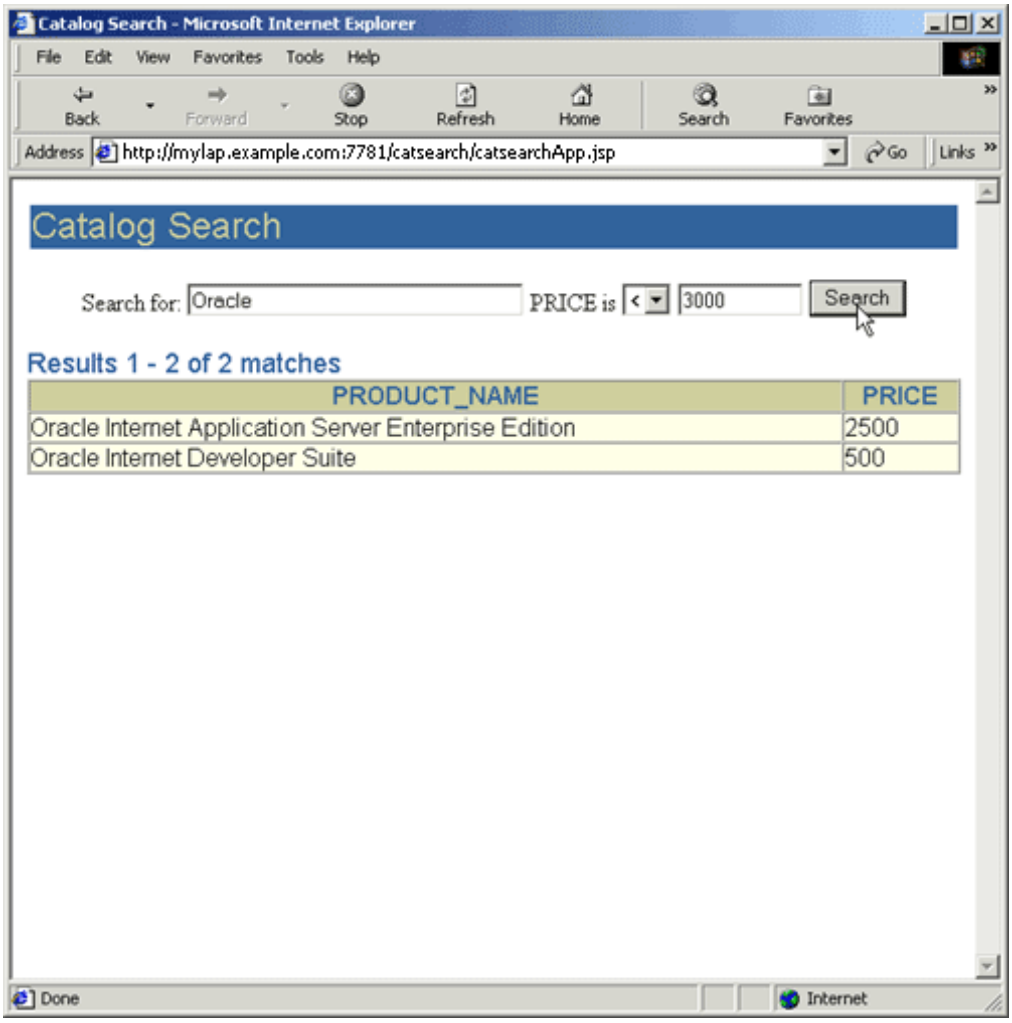
   ```
   sqlplus>select id, title from book_catalog
          where catsearch(title,'Java','price > 10 order by price') > 0
   ```

6. Copy the `catalogSearch.jsp` file to your JSP directory.

When you do so, you can access the application from a browser. The URL is `http://`*localhost*`:`*port*`/`*path*`/catalogSearch.jsp`.

The application displays a query field in your browser and returns the query results as a list of HTML links. See Figure B-1.

Figure B-1 Screenshot of the Web Query Application



B.2.2 JSP Web Application 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/technetwork/indexes/downloads/index.html

B.2.2.1 loader.ctl

```
LOAD DATA
  INFILE 'loader.dat'
  INTO TABLE book_catalog
  REPLACE
  FIELDS TERMINATED BY ';'
  (id, title, publisher, price)
```

B.2.2.2 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; BIG LITTLE BOOKS; 10
10; Emergency Surgery Tips and Tricks; SPOT-ON PUBLISHING; 10
11; Programming with Your Eyes Shut; KLONDIKE BOOKS; 10
12; English in Twelve Minutes; WRENCH BOOKS 11
13; Spanish in Twelve Minutes; WRENCH BOOKS 11
14; C++ Programming for Woodland Animals; 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; Python Programming for Snakes;  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
```

B.2.2.3 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:dev";

  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="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="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>
<%
 }
%>
```

C

Custom Index Preference Examples

This appendix describes a few custom index preference examples.

This appendix contains the following topics:

C.1 Datastore Examples

You can use datastore preferences to specify how your text is stored. These are the examples for setting some of the datastore preference types.

Specifying DIRECT_DATASTORE

This example creates a table with a `CLOB` column to store text data. It then populates two rows with text data and indexes the table by using the system-defined `CTXSYS.DEFAULT_DATASTORE` preference, 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

This example creates a `MULTI_COLUMN_DATASTORE` 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 FILE_DATASTORE

This example creates a data storage preference by using `FILE_DATASTORE` to specify that the files to be indexed are stored in the operating system. The example uses `CTX_DDL.SET_ATTRIBUTE` to set the `PATH` attribute to the `/docs` directory.

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

> **Note:**
>
> Starting with Oracle Database 19c, the Oracle Text type `FILE_DATASTORE` is deprecated. Use `DIRECTORY_DATASTORE` instead.

Specifying DIRECTORY_DATASTORE

This example creates a `DIRECTORY_DATASTORE` preference called `MYDS`. The example uses `CTX_DDL.SET_ATTRIBUTE` to set the `DIRECTORY` attribute to `myhome`, which is the Oracle directory object.

```
exec ctx_ddl.create_preference('MYDS','DIRECTORY_DATASTORE')
exec ctx_ddl.set_attribute('MYDS','DIRECTORY','myhome')
```

Specifying URL_DATASTORE

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.example.com');
 ctx_ddl.set_attribute('my_url','NO_PROXY','us.example.com');
 ctx_ddl.set_attribute('my_url','TIMEOUT','300');
end;
```

> **Note:**
>
> Starting with Oracle Database 19c, the Oracle Text type `URL_DATASTORE` is deprecated. Use `NETWORK_DATASTORE` instead.

Specifying NETWORK_DATASTORE

This example creates a `NETWORK_DATASTORE` preference called `NETWORK_PREF` 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('NETWORK_PREF','NETWORK_DATASTORE');
 ctx_ddl.set_attribute('NETWORK_PREF','HTTP_PROXY','www-
proxy.us.example.com');
 ctx_ddl.set_attribute('NETWORK_PREF','NO_PROXY','us.example.com');
 ctx_ddl.set_attribute('NETWORK_PREF','TIMEOUT','300');
end;
/
```

Related Topics

- *Oracle Text Reference*

C.2 NULL_FILTER Example: Indexing HTML Documents

If your document set is entirely in HTML, then Oracle recommends that you use `NULL_FILTER` in your filter preference because it does no filtering.

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

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

C.3 PROCEDURE_FILTER Example

Consider a `CTXSYS.NORMALIZE` filter procedure 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, set up your filter preference:

```
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;
```

C.4 BASIC_LEXER Example: Setting Printjoin Characters

Printjoin characters are nonalphanumeric characters that are to be included in index tokens, so that words such as *vice-president* are indexed as *vice-president*.

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

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

Create the index with printjoins characters set as previously shown:

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

C.5 MULTI_LEXER Example: Indexing a Multilanguage Table

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

The first step is to create the multilanguage table with a primary key, a text column, and a language column:

```
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 by using `CTX_DDL.ADD_SUB_LEXER`:

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

Add the German and Japanese lexers in their respective languages with the `CTX_DDL.ADD_SUB_LEXER` procedure. Also assume that the language column is expressed in the standard ISO 639-2 language codes, and add those codes 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');
```

Create the `globalx` index, specifying the multi-lexer preference and the language column in the parameter clause:

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

C.6 BASIC_WORDLIST Example: Enabling Substring and Prefix Indexing

This example improves performance for wildcard queries by setting the wordlist preference for prefix and substring indexing. For prefix indexing, the example creates 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;
```

C.7 BASIC_WORDLIST Example: Enabling Wildcard Index

Wildcard indexing supports fast and efficient wildcard search for all wildcard expressions.

This example creates a wordlist preference and enables the wildcard (“K-gram”) index. By default, the K-grams have a K value of `3`:

```
begin
     ctx_ddl.create_preference('mywordlist','BASIC_WORDLIST');
     ctx_ddl.set_attribute('mywordlist','WILDCARD_INDEX','TRUE');
end;
```

> **See Also:**
>
> *Oracle Text Reference* for more information about the `BASIC_WORDLIST` attributes table and the `WILDCARD_INDEX` and `WILDCARD_INDEX_K` attributes