WITHIN YOU AND ABOUT YOU: GETTING STARTED WITH *INTER*MEDIA TEXT

Carol Brennan, Comedy Central
Douglas Scherer, Core Paradigm

Introduction

Oracle8i's *inter*Media Text provides a set of methods and extensions to standard SQL that can be used for searching text and documents. Unlike Oracle ConText, an add-on product available with Oracle7, *inter*Media Text fully integrates most search-related tasks into the familiar database environment. With it, effective searches through SQL queries become possible, and index creation and management becomes more intuitive.

This document explores, with the aid of examples, all of the steps involved in building an *inter*Media Text search-enabled database. It will provide general considerations for table creation, methods for loading data into tables, tasks involved with creating text indexes on tables, examples of search queries, and issues surrounding text index maintenance. Special attention will be paid to the use of large objects (LOBs) as it relates to document management for the purpose of searching.

Creating and Loading Database Tables

When preparing to use *inter*Media Text, table creation and loading is done no differently than it would be for any other table; it follows that implementing *inter*Media Text searches on existing data is a straightforward process. The processes of table creation and population will not be examined fully here, although the following notes are relevant to our discussion of *inter*Media Text:

- In order to build an *inter*Media Text Index (which will be discussed in the next section), the table must have a primary key.
- Any valid table loading method may be used to insert data into a table which is to be the basis of a text search. This would generally consist of a standard INSERT statement; if the table contains a LOB column, then components of the DBMS_LOB package would also be necessary to the process. SQL*Loader can also be used to bulk-load data, and the Oracle EXPORT/IMPORT utilities can be used for migrating the data from another schema.

Example

Suppose we wish to create a searchable table of cooking recipes. One may build this table with the following statement:

```
CREATE TABLE recipes (
  id                NUMBER         NOT NULL PRIMARY KEY,
  name              VARCHAR2(100)  NOT NULL,
  prep_time_minutes NUMBER,
  servings          NUMBER,
  description       VARCHAR2(1000),
  html_page         CLOB DEFAULT EMPTY_CLOB());
```

For simplicity, we will assume in subsequent examples that this table has been created and contains data.

Creating Text Indexes

Given that a database table exists and has a primary key defined, it can be made *inter*Media Text-searchable through the creation of a special index. This type of index will be referred to as a *text index* or an inter*Media Text index* for the remainder of this document, in order to avoid confusing it with standard Oracle indexes.

The syntax for creating a text index is as follows:

```
CREATE INDEX <index_name>
ON <table_name> (<column_name>)
INDEXTYPE IS CTXSYS.CONTEXT;
```

The resultant index exists for the exclusive use of *inter*Media Text in performing search queries; it does not eliminate any need for standard indexing. It differs fundamentally from a standard index in the fact that it provides a means of locating documents based on searches of their contents, unlike standard indexes which provide fast access to record contents via a pointer to the record itself. Because of this distinction, *inter*Media Text indexes are often referred to as "inverted" indexes.

The following rules govern the creation of text indexes:

- A text index can only be defined on one table column; composite text indexes are not permitted. However, text indexes on multiple columns of one table are allowed.
- Unlike a standard index, if an error occurs during the creation of a text index, the index is still created and must, in most cases, be dropped before making another creation attempt.

Stages of Index Creation

The process of index creation actually consists of four classes of activities, which occur in sequential order and which are all customizable.

Stage 1: Datastore

In this stage, the storage specifics for the text being indexed are specified. For example, the physical location of the data (i.e. external files or within the database itself) is determined.

Stage 2: Filter

Next, the output of the previous step (the data itself) is filtered as specified. For example, if the data exists externally in a Microsoft Word document, it can be parsed into plain text, HTML, or XML in this step.

Stage 3: Section

Here, the output of the previous step (the filtered data) is broken down into groups according to the definition of section groups and sections. This will be useful later when we are running queries against the indexed data, as it will allow us to restrict returned records that contain certain words or phrases within specific sections only, through the use of the WITHIN clause. (Implicitly, all documents have "paragraph" and "sentence" sections.)

Stage 4: Lexer

Finally, the text returned from the previous step is split into words, based upon the lexical standards of the language being used.

Each of these classes have parameters (objects) which can be defined explicitly. Defaults exist in the CTX_PARAMETERS view.

Example

To continue the example we began above: the following statements will create indexes on several useful fields within the RECIPES table:

```
CREATE INDEX recipe_name_index
```

```
ON recipes (name)
INDEXTYPE IS CTXSYS.CONTEXT;

CREATE INDEX recipe_desc_index
ON recipes (description)
INDEXTYPE IS CTXSYS.CONTEXT;

CREATE INDEX recipe_html_index
ON recipes (html_page)
INDEXTYPE IS CTXSYS.CONTEXT;
```

Note that, in the example, we have accepted the defaults for all index creation classes. Text index creation is customizable by using preferences. More information on preferences can be found in the document "Oracle8i interMedia Text 8.1.5 – Technical Overview" avaliable at
http://technet.oracle.com/sample_code/products/intermedia//htdocs/text_samples/imt_815_techover.html

Performing Queries Against Indexed Text

With an *inter*Media Text index in place, the data is prepared for text queries. These queries are reasonably simple to construct, as they are exactly like standard SQL queries, with several new functions used mainly within WHERE clauses.

In the following discussion, the CONTAINS clause, which enables the most basic type of text query, will be explored. Two other clauses used in more sophisticated queries, WITHIN and ABOUT, will also be discussed.

CONTAINS Clause

The CONTAINS clause is used to match an exact word or phrase, or a combination of exact words or phrases, to the contents of a text field. The following are several basic variations on the CONTAINS clause; there are, however, many others available.

Single-Word Match

The simplest form of the CONTAINS clause matches one word to text, as in this example:

```
SELECT id
FROM   recipes
WHERE  CONTAINS(description, 'bean') > 0;
```

This query will return rows for which the DESCRIPTION field contains the word "bean". The second argument to the CONTAINS function can be no more than 2000 characters long. It returns a number corresponding to the strength of the match for each record ("0" indicates no match). Note that text searches are not case sensitive.

Phrase Match

Using the same syntax, the CONTAINS clause can also be used to search for a phrase:

```
SELECT id
FROM   recipes
WHERE  CONTAINS(description, 'black bean soup') > 0;
```

Note that this query will only return records for which the complete phrase "black bean soup" is present in the DESCRIPTION field.

Match Containing Boolean Operators

The Boolean operators AND, OR, and NOT can be used to combine words and/or phrases within a CONTAINS clause. Standard operator precedence applies.

```
SELECT id
FROM   recipes
WHERE  CONTAINS(description, '(bean AND soup) OR rice') > 0;

SELECT id
FROM   recipes
WHERE  CONTAINS(description, 'bean NOT soup') > 0;
```

As illustrated in the second example above, in this case, the NOT operator is a not considered a unary operator as it often is; in this example, it corresponds to the English phrase “bean but not soup”.

Weighted Match

Within a CONTAINS clause, search terms can be assigned different weights:

```
SELECT id
FROM   recipes
WHERE  CONTAINS(description, '(bean*2) AND rice') > 0;
```

In this example, the result set will include records for which the DESCRIPTION field contains both “bean” and “rice”. However, documents containing “bean” will weighed twice as heavily as those containing “rice”. As a result, documents containing “bean” will be more likely to have a higher score. This is relevant to our discussion of sorting by score, which appears as the next topic.

Scoring and Sorting

As eluded to earlier, the CONTAINS clause returns a value that corresponds to the strength of the match. This value, or score, can be used to sort query results, as illustrated in the following example:

```
SELECT id
FROM   recipes
WHERE  CONTAINS(description, 'bean', 1) > 0
ORDER BY SCORE(1) DESC;
```

As illustrated, the CONTAINS clause must contain a new numerical argument that corresponds to the argument for SCORE in the ORDER BY clause. Any number can be used, but it must be the same both inside and outside of the CONTAINS argument list. (Note that, here, SCORE(1) could also have been included in the list of SELECTed values.)

Complex Queries

All of the examples thus far have consisted of CONTAINS used within very simple queries. However, CONTAINS can also be used in more complex situations, as illustrated in the following examples:

```
SELECT id
FROM   recipes
WHERE  CONTAINS(name, 'vegetarian') > 0
AND    id NOT IN (SELECT id
                  FROM   recipes
                  WHERE  CONTAINS(description, 'microwave') > 0);
```

Here, CONTAI NS is used within a subquery. It can also be called from within PL/SQL, and from within certain DML statements (e.g. “INSERT AS SELECT…”).

```
CREATE OR REPLACE VIEW quick_rice_recipes AS
  SELECT *
```

```
FROM   recipes
WHERE  CONTAINS(description, 'rice') > 0
AND    prep_time_minutes <= 20;
```

As illustrated, CONTAINS can also be called from within a view definition.

WITHIN Clause

If documents' sections are properly defined, as discussed earlier in this document under index creation, then searches can be restricted to certain sections within documents using the WITHIN clause.

As a simple example, suppose that, for each of our recipe records, the HTML_PAGE field consists of an HTML document instead of plain text. As an HTML document, it contains logical sections as defined by pairs of tags, such as title (defined by "<TITLE>…</TITLE>"), body text (defined by <BODY>…</BODY>), and so forth. Using these sections, we can run queries such as:

```
SELECT id
FROM   recipes
WHERE  CONTAINS(html_page, 'stew WITHIN title') > 0;
```

The power of WITHIN can be more fully exploited if XML is used; in this case, the document will contain specialized tags in addition to the formatting tags provided with plain HTML. Each set of tags can be defined as a section, and targeted WITHIN queries can then be executed against the data. (The reader is referred to reference information on defining sections – specifically, specifying section groups and sections.)

For more information about XML, see any of the excellent XML resources on the Web. For a concise introduction to XML and its use of tagged fields, see "Get Up to Speed with XML" by Boris Feldman, available at http://www.xmlmag.com/upload/free/features/xml/1999/01win99/bfwin99/bfwin99.asp

ABOUT Clause

ABOUT queries are used to search for documents containing words with a similar meaning, or having a similar *theme*, as the search term. The syntax is illustrated in the following example:

```
SELECT id
FROM   recipes
WHERE  CONTAINS(description, 'about(bean)') > 0;
```

This query would return records with the word "bean" in the DESCRIPTION field, as well as those containing other recognized forms of the word "bean". By default, this is based on the built-in thesaurus that comes with *inter*Media Text.

The standard thesaurus can be expanded, however, for use with a particular database, as illustrated in the following example:

Example

Suppose that our recipe database is run by a company marketing bean and soup products. It would be desirable for searches to return records for which the text contains the company's products identified by name, in addition to those containing the search word. Perhaps the company also wishes to make their search facility usable to persons searching on foreign-language terms. In this case, a custom thesaurus could be built.

First, a text file similar in format to the following would be created:

```
bean
  syn garbanzo
  syn legume
  syn acme hearty has-beens
soup
```

```
  syn stew
  syn broth
  syn acme liquefied legumes
  syn acme pulverized has-beens
  italian: zuppa
  spanish: sopa
…
```

Notice that, in this text file, we have defined several synonyms for the words “bean” and “soup”, and that we have identified some foreign-language equivalents for the word “soup”.

After this file is created (assume, for this example, that it is named mythes.txt), it must be loaded into the database using the CTXLOAD facility:

From the OS command line:

```
ctxload –user ctxsys/ctxsys –thes –name mythes –file mythes.txt
```

Once it is loaded, the synonyms and foreign-language equivalents are available for use in ABOUT queries.

Maintaining Text Indexes

When indexed data changes, the corresponding text indexes are not updated automatically. Therefore, the index must periodically be reset to correspond to the current state of its underlying data through a process called *synchronization*. Several synchronization options exist, and one should be chosen based on the specific needs the database and its applications.

Immediate vs. Delayed Effects of DML

When a data manipulation (DML) statement is performed on indexed data, data is invalidated and/or added. *inter*Media Text indexes will immediately reflect invalidations by not including then in any subsequent search result sets. However, addition of new data is not immediately reflected in the appropriate text indexes, and will not be reflected until the next synchronization. Therefore, the effect of each type of DML statement is as follows:

INSERT: The inserted document will not be included in any search results until after the next synchronization.

UPDATE: The old version of the document is immediately excluded from search results, and the new version of the document will not be included in any search results until after the next synchronization.

DELETE: The document is immediately excluded from search results.

Manual Text Index Synchronization

To synchronize a text index and its underlying data, the index can be manually synchronized using the following command:

```
ALTER INDEX <index_name> REBUILD [PARAMETERS ('SYNC')];
```

The “PARAMETERS(‘SYNC’)” clause causes synchronization of only the indexed records that have changed since the last rebuild. Without this clause, the entire index will be rebuilt, which will take much longer; in this case, the “REBUILD” syntax is the same as for a standard Oracle index.

Example

```
ALTER INDEX recipe_desc_index REBUILD PARAMETERS ('SYNC');

ALTER INDEX recipe_desc_index REBUILD;
```

Automatic Text Index Synchronization

If a text-indexed table experiences a significant amount of DML, and business requirements dictate that changes must be reflected in search results on a timely basis, then manual text index synchronization may not be a viable option. Therefore, several methods exist for automatically synchronizing *inter*Media Text indexes.

CTXSRV

CTXSRV, a utility that is provided as part of interMedia Text, can be set to run as a background process to repeatedly synchronize text indexes. Depending on its configuration, it can synchronize indexes as often as once every few seconds. It should be noted that, because of the way it operates, CTXSRV causes significant fragmentation within text indexes.

DBMS_JOB or Cron

DBMS_JOB can be used to schedule the automatic execution of an "ALTER INDEX REBUILD PARAMETERS ('SYNC')" statement. (On UNIX, a script performing a similar function can be set to execute via cron.)Oracle Enterprise Manager

Index synchronization can be run within the Oracle Enterprise Manager Job Queue.

Summary

Oracle8i's *inter*Media Text makes it possible to perform text queries on database fields. It provides methods for building indexes on text data as well as an extensive set of SQL extensions for querying the indexed data. It includes utilities for keeping text indexes synchronized with their underlying data sets. Finally, interMedia Text allows for customized extension of its standard search capabilities, including definable document sections and extendable thesauri.

About the Authors

Carol Brennan is a Database Administrator and Senior Technical Analyst at Comedy Central. She is a Technical Editor of the *Oracle PL/SQL Interactive Workbook* and the *Oracle DBA Interactive Workbook*, both published by Prentice Hall. She is an instructor for the Computer Technology and Applications Program at Columbia University and an Oracle8 Certified Database Administrator.
carol.brennan@comedycentral.com

Douglas Scherer (Oracle7 and Oracle8 Certified – Chauncey and OCP) is president of Core Paradigm - providing guidance, consulting, and formal training solutions. He is a frequent speaker at international conferences, corporations and user-group meetings and has appeared in *Visions of the New Millennium,* a series seen on PBS and its affiliates. He is lead author of *Oracle 8i Tips & Techniques* published by Osborne McGraw-Hill (Oracle Press), and co-author of the *Oracle DBA Interactive Workbook* and *Complete Video Course* both published by Prentice Hall. He chaired the database track at Columbia University's Computer Technology and Applications (CTA) program for three years and is currently the database track's curriculum advisor. His column on Oracle's interMedia services appears in *Oracle Magazine*.
dscherer@corparadigm.com