Oracle*8i* *inter*Media Text 8.1.5 - Technical Overview

*inter*Media integrates all of the features and functions of the former Oracle ConText product with image, audio and video, and geographic location services for Web Content Management applications built with the Oracle8*i* Internet Platform.
*inter*Media Text has been completely re-architected, enhanced and tightly integrated with Oracle8*i* to provide up to an order of magnitude better search performance, greater ease of use and new capabilites like thematic searches. Please refer to the documentation for detailed information.

Table of Contents

1. Introduction to *inter*Media text
2. Installation and Setup
3. Indexing
4. The Preference System
5. Datastores
6. Filters
7. Section Groups
8. Lexers
9. Other Indexing Classes
10. DML
11. Optimization
12. Querying
13. Query Operators (Part 1)
14. Query Operators (Part 2)
15. Thesaurus
16. Query Tuning
17. Query Feedback
18. Highlighting
19. Linguistic Extraction

Introduction to *inter*Media Text

What does it do?

*inter*Media Text extends Oracle8*i* by indexing any text or documents stored in Oracle8i, in operating system flat files or URLs. It also enables content-based queries, (e.g., find text and documents which contain this word) using familiar, standard SQL. It enables file-based text applications to use Oracle8*i* to manage text and documents in an integrated fashion with traditional relational information.
The easiest way to grasp what it does is to see it in action...

How do you use it?

Here is a quick example. We'll start by creating a table to hold our documents. For now, we'll use a small varchar2 column to hold the document text. We'll also need a primary key to identify each document.

```
    create table docs (id number primary key, text varchar2(80));
```

Nothing special about that. Let's populate it with two example documents:

```
    insert into docs values (1, 'first document');
    insert into docs values (2, 'second document');
    commit;
```

Now comes the fun part -- we build a Text index:

```
    create index doc_index on docs(text)
    indextype is ctxsys.context;
```

This is, for the most part, familiar DDL -- the difference is the indextype clause, which instructs Oracle to build a Text index instead of a regular, b-tree index. Once the index has been created, we can run content-based queries on our huge database of two documents using the contains function:

```
    select id from docs
     where contains(text, 'first') > 0;
```

This will find all rows in docs where the text column contains the word "first", (which is document 1, for those not following closely). The > 0 part is necessary to make it legal Oracle SQL, which does not support boolean return values for functions (at this time).

Integration Architecture

*inter*Media Text is more tightly integrated with Oracle8*i*. For those of you familiar with previous versions of ConText, notice that-- in the previous example, no ctxsrv was required. The result is a specialized "domain index" which works like any b-tree index. It's all transparent to the user -- Oracle8*i* is simply extended to support text. Unlike previous versions of Oracle ConText, there are no servers to start up, there is no query rewrite, and index creation is done through familiar SQL rather than through a custom PL/SQL interface.

Installation and Setup

Migrating from Previous Versions

Due to significant architectural changes, *inter*Media Text is not backward-compatible with previous versions of Oracle ConText. Existing installations and applications need to be migrated to work with Oracle8*i*. There is a manual which details the steps of this process. Please remember that some of those steps need to be completed on the existing system **before** you install Oracle8*i*.

Installation

By default, the Oracle *inter*Media installation procedure installs *inter*Media Text. *inter*Media Text can also be installed separately, if you choose a custom install.

Important Files

?/bin/ctxsrv	This may be renamed ctxsrv8 or some such. Oracle8i still supports the ctxsrv server. However, the only valid personality mask is M. You need to run it only when you want background DML. Batch DML (sync) does not require a running server. We'll go into DML in more depth later.
?/ctx/admin	Data dictionary scripts. See below for details.
?/ctx/admin/defaults	Language-specific default preferences. See below, and an upcoming issue on the preference system for more details.
?/ctx/data	This directory has data files needed by *inter*Media Text, which includes the linguistic lexicon files, the Xerox stemming files, and the Korean lexer dictionary. If you get strange internal errors in linguistics, stemming, etc. check this directory.
?/ctx/bin	This directory is for user-defined filters, which we'll talk about later. This directory should also contain the ctxhx program which is used by the INSO filter.
?/ctx/lib	This directory has platform-specific shared libraries and template files used by the INSO filter. It also contains the safe callout used by indexing and document services -- libctxx8.so (name will change from platform to platform -- look for "ctxx")

Data Dictionary Installation

*inter*Media Text is integrated with the Oracle Database Creation Assistant (DBCA) so the ctxsys data dictionary should be installed with this tool. If a manual installation is required please follow these steps.

1. Before starting, make sure that:
   - The *inter*Media Text software tree is installed
   - The database does not have a ctxsys user
   - Current directory is ?/ctx/admin
   - You can sqlplus internal
2. Create the ctxsys user. Pass it the ctxsys password, default tablespace, and temporary tablespace as arguments.
3.
```
        sqlplus internal @dr0csys
```
4. Install the data dictionary:
5.
```
        sqlplus ctxsys/ @dr0inst
```

   The argument is the full path to the ctxx library, for example:

```
        sqlplus ctxsys/ @dr0inst
/some_directory/app/oracle/product/8.1.5/ctx/lib/libctxx8.so
```

   On Unix, you must not use the environment symbol $ORACLE_HOME. You have to use the actual physical path.

6. Install appropriate language-specific default preferences. There are forty-odd scripts in ?/ctx/admin/defaults which create language- specific default preferences. They are named in the form drdefXX.sql, where XX is the language code (from the Server Reference Manual).

   To install the US defaults, for instance:

```
sqlplus ctxsys/ @defaults/drdefus.sql
```

We'll talk more about this in the upcoming preference system issue.

After these steps, *inter*Media Text should be installed and working.

Post-Installation Setup

If this database was an existing ConText site, make sure to remove text_enable from the init.ora. It is no longer used in Oracle8i, and will actually prevent Oracle8i from operating properly -- you'll get errors like "cannot find package DR_REWRITE".
Finally, ensure that the Net8 listener is running and is configured to invoke external procedures. A brief description of the process is below, with complete details are in the Oracle8i Server Administrator's Guide.

1. Add an entry to the tnsnames.ora:
2.   extproc_connection_data =
3.   (DESCRIPTION =
4.   (ADDRESS = (PROTOCOL = ipc)
5.   (KEY = DBSID))
6.   (CONNECT_DATA = (SID = ep_agt1)))

   DBSID is the database SID. ep_agt1 can be named anything. extproc_connection_data should not be changed.
7. Add the following to the listener SID_LIST:
8.   (SID_DESC = (SID_NAME = ep_agt1)
9.   (ORACLE_HOME = /oracle)
10.   (ENVS = LD_LIBRARY_PATH=/oracle/ctx/lib)
11.   (PROGRAM = extproc))

   ep_agt1 matches the CONNECT_DATA SID for extproc_connection_data in the tnsnames.ora.
   The PROGRAM section tells the Net8 listener to start the external procedure process.
   The ENVS section, which is shown here for UNIX, will ensure that the environment includes ?/ctx/lib in LD_LIBRARY_PATH. This is needed so that indexing can use the INSO filters.
   On NT, you may need to have ORACLE_HOME set in this section as well.

12. Since the extproc_connection_data ADDRESS section specifies ipc, make sure that the ADDRESS_LIST of listener.ora accepts ipc connections.

    A quick way to test the Net8 configuration is to do:

    ```
    exec ctx_output.start_log('log')
    ```

    from SQL*Plus. If you get the error:

    ```
    DRG-50704: Net8 listener is not running or cannot start external procedures
    ```

    then things aren't set up correctly. Some of the things to check:

    - listener is not running
    - listener.ora is not configured for extproc

- need to reload the listener
- tnsnames.ora is not configured for extproc

Indexing

Creating Indexes

The first step in using *inter*Media Text is to create a Text index. Without a b-tree index, value queries are slower; without a Text index, contains queries are simply not possible. As we've seen, the index is created using the create index command:

```
  create index INDEXNAME on TABLE(COLUMN)
  indextype is ctxsys.context
```

Unlike previous versions of ConText, there is no separate policy creation step. We'll talk more about this when we discuss the preference system. View indexing is not allowed in Oracle8*i*, consistent with regular b-tree indexes. Parallel index creation is also not supported in this first Oracle8*i* version.
Composite indexes are not supported -- only one column is allowed in the column list. This column must be one of the following types: CHAR, VARCHAR, VARCHAR2, LONG, LONG RAW, BLOB, CLOB, BFILE. Date, number, and nested table columns cannot be indexed. Object columns also cannot be indexed, but their attributes can be, provided they are atomic datatypes.
The table must also have a primary key constraint. This is needed mainly for identifying the documents for document services, and may be used in the future for other purposes. Composite primary keys are supported, up to 16 columns.
The issuing user does not need the ctxapp role to create an index. If the user has Oracle grants to create a b-tree index on the column, then they have sufficient permission to create a Text index.
Unlike previous versions of ConText, the issuing owner, table owner, and index owner can all be different users, just like regular b-tree indexes.
IMPORTANT: If a syntax error occurs in the create index statement, the index is still created. This is different from regular b-tree indexes; before you reissue the corrected statement, you must drop the failed index first.
If an error occurs during actual indexing (e.g. you run out of tablespace) then you can pick up where you left off (after correcting the problem, of course) using alter index:

```
  alter index INDEXNAME rebuild parameters ('resume')
```

Storage clauses to the create index statement are not used. Index storage parameters are provided using the STORAGE object, which we will discuss later. Partitioning clauses are also not supported at this time, although you can manually partition the index tables if you wish, again using the STORAGE object.
Once the index is created, any export will include the index definition. At import time, imp will re-create the index by issuing the create index statement.

The Indexing Pipeline

Oracle8i detects that this is a Text index and performs the text indexing. The index is created using a pipeline of steps - "The Indexing Pipeline".

The diagram below shows the indexing pipeline:

```
 +-----------+           +----------+ filtered   +-----------+
 |           | doc data  |          | doc text   |           |
 | datastore |---------->|  filter  |----------->| sectioner |---------+
 |           |           |          |            |           |--+      |
 +-----------+           +----------+            +-----------+  |      |
      ^                                                         |      |
      |                        section offsets                  |      |
      |column data            +---------------------------------+      |
      |                  |                                  |
      |                  V                                  |
 +==========+           +----------+           +-----------+          |
 |          |           |          |           |           |
 | database |<-----------|  engine  |<-----------|   lexer   |<--------+
 |          |  index data|          |   tokens   |           |   plain
 +==========+           +----------+           +-----------+   text
```

Let's step through this pipeline, stage-by-stage. Each stage will be covered in depth later, so they'll be described here briefly.

Start with the datastore. This stage loops over the rows of the table and reads the data out of the column. Unlike previous versions, the rows are not read in any particular order. The datastore passes out document data. Usually, this will simply be the column data, but some datastores use the column data as a pointer to the document data. The URL_DATASTORE, for instance, uses the column data as a URL, does a GET, and passes out the returned data.

The filter takes the document data and transforms it to some kind of text representation. This is needed when storing binary documents such as Word or Acrobat files. Unlike previous versions, the output of the filter does not have to be plain text -- it can be a text format such as XML or HTML.

The sectioner, new for Oracle8i, takes the output from the filter, and converts it to plain text. There are different sectioners for different text formats, including XML and HTML. Conversion to plain text includes detecting important section tags, removing "invisible" information, and reformatting the text.

The lexer takes the plain text from the sectioner and splits it into discrete tokens. We have lexers for whitespace-delimited languages, and specialized lexers for Asian languages, where segmentation is quite a bit more complex. The basic lexer also includes theme functionality to build unified text/theme indexes.

Finally, Oacle8i takes all the tokens from the lexer, the section offsets from the sectioner, and a list of low-information words called the stoplist, and builds an inverted index. An inverted index stores tokens, and the documents in which these tokens occur. For instance, our example from issue #1:

```
  insert into docs values (1, 'first document');
  insert into docs values (2, 'second document');
```

would produce an inverted index like this:

```
  DOCUMENT  ---> doc 1 position 2, doc 2 position 2
  FIRST     ---> doc 1 position 1
  SECOND    ---> doc 2 position 1
```

Each document is assigned an identifier called a docid, which is used in the inverted index. The document primary key isn't stored.

Logging

The time needed to create an index is a function of the amount of text being indexed. Oracle8*i* can create a log file which can be monitored during indexing. start_log starts the logging, and should be called before issuing the create index, in the same session:

```
  ctx_output.start_log('mylog');
```

This will create a mylog file in ?/ctx/log which can be used to monitor indexing progress. The directory can be changed by ctxsys, using the system parameter LOG_DIRECTORY. See documentation for details.
IMPORTANT: In this first release of *inter*Media ?/ctx/log is not created at install time. You may get errors like "unable to open file". Simply have the DBA create this directory.
Logging is halted when the session exits or when end_log is called:

```
  ctx_output.end_log;
```

Errors During Indexing

Processing errors encountered during indexing -- filter errors, file cannot be read, etc. -- do not bring indexing to a halt. Instead, the error is logged and indexing continues to the next file.
You can see these logged errors in the ctx_user_index_errors view. It will tell you the rowid of the failed document, and the error which was encountered. You need to manually empty this table when you are finished reviewing the errors:

```
    delete from ctx_user_index_errors;
```

The Preference System

Classes, Objects, Preferences

The preference system allows you to customize Text indexing. Each stage of the indexing pipeline is really an interface into which customized objects can be plugged in. These stages are called "classes", and they can be found in the ctx_classes view:

DATASTORE	Reads column data and returns document data
FILTER	Reads document data and returns formatted document text
SECTION_GROUP	Reads formatted document text and returns section offsets and plain document text
LEXER	Reads plain document text and breaks it into tokens
WORDLIST	Contains query expansion defaults
STOPLIST	Contains a list of low-information words
STORAGE	Index table storage parameters

Each class has one or more "objects" which are like behavioral templates. The DATASTORE class, for instance, has the following objects:

DIRECT_DATASTORE	Passes column data directly as document data
DETAIL_DATASTORE	Concatenates multiple rows of a detail table to construct document data
FILE_DATASTORE	Interprets column data as a filename. Reads file and passes file contents as document data

URL_DATASTORE	Interprets column data as a URL. Performs a GET and passes return as document data
USER_DATASTORE	Invokes a stored procedure to synthesize document data

Objects can be found in the ctx_objects view.
But objects are only templates, and cannot be used directly in an index. Instead, you create a "preference" from one of these object templates, customize it by setting "attributes", then use the preferences to create the index. Let's take a closer look:

Creating Preferences

You create a preference using ctx_ddl.create_preference, specifying an object to use as template:

```
    ctx_ddl.create_preference('mypref', 'FILE_DATASTORE');
```

This creates the preference mypref, using the FILE_DATASTORE object. If you are familiar with previous versions, the object names have been changed, and there is no comment argument.
This is done in PL/SQL, so surround it with "begin" and "end", or use "exec" if you are in SQL*Plus. Also, in order to create preferences, the issuing owner must have the ctxapp role.
Once the preference is created, we can customize it using set_attribute: For our example, let's say all our files were in /docs. We can simply set the PATH attribute to communicate this to *inter*Media:

```
    ctx_ddl.set_attribute('mypref', 'PATH', '/docs');
```

If you are familiar with previous versions, set_attribute now comes *after* create_preference, which allows us to do attribute-level validation, and necessitates the first argument, which is the preference name.
This attribute is a text attribute, so it is set to a text value. There are also number attributes and boolean attributes, which should be set to 'TRUE' or 'FALSE' (or 'YES'/'NO'). The attributes for each object and their types can be found in view ctx_object_attributes or in the documentation.
Some attributes have a list of allowed values. The allowed values can be found in the view ctx_object_attribute_lov. Unlike previous versions, LOV attributes can be set using a descriptive "tag" rather than a numeric value.

Non-preference Classes

Preferences are used for all classes except the SECTION_GROUP and STOPLIST classes. These classes use specialized objects and have their own API. We'll discuss this in the respective upcoming issues.

Using Preferences

Once you've created your preferences, you can build all kinds of customized indexes by mixing and matching them. Because all objects of a class adhere to a common interface, any filter can be hooked up to any datastore, etc. Got a bunch of Korean Word 97 files? No problem -- hook up a FILE_DATASTORE preference, an INSO_FILTER preference, and a KOREAN_LEXER preference.
You plug in your preferences using the parameters clause of create index:

```
    create index doc_index on docs(text)
      indextype is ctxsys.context
    parameters ('datastore mypref')
```

This tells create index to use mypref as the datastore. Each of the classes has a parameters keyword to name a preference. The keyword is the same as the class name, except for SECTION_GROUP, whose keyword is the phrase "section group". You can specify multiple classes by simply tacking the keyword-preference pairs on the end of the string:

```
   parameters('datastore mypref filter myfilter')
```

Any user can use any preference. To specify a preference in another user's schema, simply add the owner name in the parameters string:

```
   parameters('datastore kenny.mypref')
```

Unlike previous versions, the index makes a copy of the preference settings, which means that after use in an index, the preference can be modified or dropped without affecting the index. Note that only the preference owner can modify or drop it.

The Default System

In the example above, only the datastore class is set. Since preferences were not specified for the other classes in the parameters string, they get their values from the default system.

The default system uses system parameters. System parameters are name-value pairs which apply to the *inter*Media Text installation as a whole -- kind of our version of init.ora. They can be seen in the view ctx_parameters.

The values for the parameters used by the default system are preference names. For instance, in our example we have not specified a lexer preference. *inter*Media gets the value of the system parameter DEFAULT_LEXER, and uses the preference named there. This allows the administrator to set the default behavior for the system.

The set of parameters used by the default system are:

DEFAULT_DATASTORE	The default datastore preference. At install time (the default default) it is set to CTXSYS.DEFAULT_DATASTORE, a preference which uses the DIRECT_DATASTORE object.
DEFAULT_FILTER_BINARY	This is the filter preference to use when indexing binary datatypes such as RAW, LONG RAW, or BLOB. At install time it is set to CTXSYS.INSO_FILTER, which uses the INSO_FILTER object.
DEFAULT_FILTER_FILE	This is the filter preference to use when indexing files, either BFILE or the FILE_DATASTORE. At install time it is set to CTXSYS.INSO_FILTER.
DEFAULT_FILTER_TEXT	This is the filter preference to use when indexing text datatypes such as CHAR, VARCHAR2, or CLOB. At install time it is set to CTXSYS.NULL_FILTER, which does no filtering.
DEFAULT_SECTION_HTML	This is the section group to use when the filter class uses the INSO filter, or the datastore uses the URL_DATASTORE. The INSO filter filters binary files to HTML (see filters). At install time it is set to CTXSYS.HTML_SECTION_GROUP, which merely converts the HTML to plain text.
DEFAULT_SECTION_TEXT	This is the section group to use in all other cases. At install time it is set to CTXSYS.NULL_SECTION_GROUP, which does no sectioning.
DEFAULT_LEXER	This is the default lexer preference. At install time it is set to

	CTXSYS.DEFAULT_LEXER. The definition of that preference is language-specific. See below for more details.
DEFAULT_WORDLIST	This is the default wordlist preference. At install time it is set to CTXSYS.DEFAULT_WORDLIST. The definition of that preference is language-specific. See below for more details.
DEFAULT_STOPLIST	This is the default stoplist. At install time it is set to CTXSYS.DEFAULT_STOPLIST. The contents of that stoplist are language-specific. See below for more details.
DEFAULT_STORAGE	This is the default storage preference. At install time it is set to CTXSYS.DEFAULT_STORAGE, which has no settings.

Language-Specific Defaults

The LEXER, WORDLIST, and STOPLIST classes are the most affected by the language of the documents. Every installation gets DEFAULT_LEXER, DEFAULT_WORDLIST, and DEFAULT_STOPLIST preferences, but the definition of these preferences will depend on the language chosen at install time.
?/ctx/admin/defaults holds language-specific scripts for each supported language. Based on the language chosen, DBCA runs the matching language-specific script, which creates the default preferences most appropriate for that language.
The result is that a German instance gets a DEFAULT_LEXER which is case-sensitive and does German decompounding, a DEFAULT_STOPLIST which has common German words, and a DEFAULT_WORDLIST with German stemming. A Japanese instance gets different defaults, more tailored to Japanese.

Browsing the Data Dictionary

Here is a list of views useful for browsing the *inter*Media Text data dictionary:
ctx_classes Available classes ctx_objects Available objects for each class
ctx_object_attributes Attribute names and types for each object
ctx_object_attribute_lov Allowed values for each LOV object attribute
ctx_preferences Preferences ctx_preference_values Attribute values of each preference ctx_user_index_objects Objects used by the user's indexes
ctx_user_index_values Attribute values for the user's indexes

Datastores

The DATASTORE class

The datastore class is responsible for reading the column data from the database, getting the document data based on the column data, and passing that up to the filter.
It is important to keep in mind that some form of the document data must be in a table. Datastore objects which access external resources such as files and web pages still require a table of pointers to those resources.
Datastore preferences are plugged into indexes using the datastore keyword:

```
create index
...
```

```
    parameters ('datastore mydstore');
```
The datastore class has five objects.

DIRECT_DATASTORE

The direct datastore is the simplest case -- it assumes that document data is stored in the indexed column. It is so simple it has no attributes to customize. There should be no reason to create a preference based on the DIRECT_DATASTORE object -- CTXSYS.DEFAULT_DATASTORE can be used for any index.
IMPORTANT: If you are indexing BFILEs, make sure the user ctxsys has READ permission on the BFILE directory.

FILE_DATASTORE

The file datastore reads the column data as a filename. It opens the file and returns the contents as the document data. The indexed column cannot be LONG or LOB types. The files must be accessible from the database host machine.
This object has one attribute: PATH. PATH is optional, and specifies the directory where the documents are stored.
IMPORTANT: If PATH is specified, the column data must be simple file names; if not specified, the column data must be full path names. You cannot mix and match -- for instance, with PATH of /A, and a column of B/C.TXT it will NOT find /A/B/C.TXT.
On some platforms, PATH can specify multiple directories. If you do this, make sure that your documents have unique names -- if a document C.TXT is in directories /A and /B, and /A is in the PATH before /B, only /A/C.TXT will be indexed.
There is not much of a difference between this object and BFILEs. It's a matter of choice. Like BFILE's, when the file changes, the row will not be automatically reindexed. You should update the column value to itself to trigger a reindex.

URL_DATASTORE

The URL datastore reads the column data as a URL. It performs a GET and returns the contents as the document data. The URLs for the documents must in the table -- the URL datastore does not crawl. The indexed column cannot be LONG or LOB types.
http, ftp, and file protocols are supported. The URL datastore also has multi-threaded read (on most platforms) to maximize throughput. It is very customizable, and has several attributes:

TIMEOUT	Read request timeout in seconds
MAXTHREADS	Maximum number of threads to use for multithreaded read
URLSIZE	Maximum length of an URL
MAXURLS	Maximum number of URL's to read at one time
MAXDOCSIZE	Maximum length of document
HTTP_PROXY	HTTP proxy

FTP_PROXY	FTP proxy
NO_PROXY	Domain for proxy exclusion

Like the file datastore, if the web page changes, then you should manually update the column value to itself to trigger the reindex.

DETAIL_DATASTORE

Many legacy applications store text line-by-line, in multiple rows of a detail table. The detail datastore constructs documents by concatenating the text of these lines into a single, coherent document.
The detail datastore must be told the specifics of the detail table:

DETAIL_TABLE	Name of the detail table
DETAIL_KEY	The foreign key column(s) in the detail table
DETAIL_LINENO	The sequence column in the detail table
DETAIL_TEXT	The text column in the detail table

For instance, let's say the master and detail tables look like this:

```
   the_master                 my_detail

   ID     TITLE                M_ID   SEQ  LINETEXT
   1      Grate Expectations    1      1    It was the best of times
                                1      2    it was the worst of times.
```

"That's G-R-A-T-E Expectations, also by Edmund Wells."
The attributes of the preference should be set like this:

DETAIL_TABLE	my_detail
DETAIL_KEY	m_id
DETAIL_LINENO	seq
DETAIL_TEXT	linetext

There is one final attribute: BINARY. By default, the detail datastore treats each row as a separate line, and, when concatenating the contents, will automatically stick a newline at the end of the text for each row. BINARY set to TRUE suppresses this. In our example, we should set BINARY to FALSE or simply leave it unset.
The index itself should to be built on the master table:

```
   create index myidx on master(somecolumn)
     indextype is ctxsys.context
     parameters ('datastore my_detail')
```

The master table must have a primary key column, just like any other table indexed by *inter*Media Text. This primary key column is used to find the corresponding detail rows, where detail.fk = master.pk.
The indexed column can be any column of allowable type -- the contents are not used by *inter*Media. If you can afford the space, you can add a dummy char(1) column called "text" or "body", to make the queries more readable:

```
   select author, title from master
    where contains(text, 'best & worst') > 0;
```

If detail rows are added, removed, or changed without changing the master text column, you should manually update the text column to itself to trigger a reindex.

For those of you familiar with previous versions of ConText, this is similar to the master detail new datastore. The old master detail datastore is no longer supported

USER_DATASTORE

The user datastore is new for Oracle8i. You write a stored procedure which, given a rowid, synthesizes a document. *inter*Media calls this procedure whenever document data is required. Let's look at an example. Say we have a table like this:

```
 articles
   id      number
   author   varchar2(80)
   title    varchar2(120)
   text     clob
```

and we want to automatically have author and title be part of the indexed document text. We can write a stored procedure following the user datastore interface:

```
   create procedure myproc(rid in rowid, tlob in out clob) is
     offset number := 1;
   begin
     for c1 in (select author, title, text from articles
                where rowid = rid)
     loop
       append_varchar_to_lob(tlob, c1.title, offset);
       append_varchar_to_lob(tlob, 'by '||c1.author, offset);
       dbms_lob.append(tlob, c1.text);
     end loop;
   end;
```

This procedure takes in a rowid and a temporary clob locator, and concatenates all the articles columns into the temp clob, This assumes a helper procedure append_varchar_to_lob has been written.

To ensure that the DBA has signed off on the code, only ctxsys-owned stored procedures are allowed for the user datastore. Thus, we need to do something like this as ctxsys:

```
   create procedure s_myproc(rid in rowid, tlob in out clob) is
   begin
     appowner.myproc(rid, tlob);
   end;
```

And, we need to make sure that the index owner can execute the stub procedure, so:

```
   grant execute on s_myproc to appowner
```

Now, back as the app owner, we create the preference, setting the PROCEDURE attribute to the name of the ctxsys stub procedure:

```
 ctx_ddl.create_preference('myud', 'user_datastore');
 ctx_ddl.set_attribute('myud', 'procedure', 's_myproc');
```

When we then create the index on articles(text) using this preference, *inter*Media indexing will see author and title in the document text.

The user datastore can be used for any on-the-fly document synthesis, including more complex master-detail relationships, nested tables, text preprocessing, or multicolumn concatenation, like the example.

There are four constraints on the procedures used in user datastores:

1. They must be owned by ctxsys
2. They must be executable by the index owner
3. They cannot issue DDL or transaction control statements like "commit"
4. They cannot be safe callouts or call safe callouts

If you change the stored procedure, indexes based upon it will not be notified, so you should manually recreate such indexes. *inter*Media cannot tell what you are doing in the stored procedure, so if the stored procedure makes use of other columns, and those column values change, the row will not be reindexed. The row is only reindexed when the indexed column changes.

Filters

The FILTER Class

The filter class takes the document data provided by the datastore class, and filters it to readable text, passing it to the sectioner. With Oracle8i, this does not have to be plain text -- it can be a text format such as HTML.
Filter preferences are plugged into indexes using the filter keyword:

```
   create index
   ...
   parameters ('filter myfilter');
```

The filter class has three objects.

NULL_FILTER

The null filter is used when the document contents are not in a binary format. It simply passes text from the datastore to the sectioner. The default CTXSYS.NULL_FILTER preference can be used in any index to employ the null filter.

CHARSET_FILTER

The charset filter is new for Oracle8i. It converts documents from a foreign character set to the database character set. This is most useful for Japanese customers, who have to deal with two widely-used and incompatible character sets (and one infrequently-used incompatible character set).
The charset filter has one attribute, CHARSET, which takes the NLS name of the source character set. A list of NLS charsets can be found in the Oracle National Language Support Guide. The destination character set is always the database character set, so does not need to be specified.
Additionally, you can specify JAAUTO, which is a custom setting for Japanese character set auto-detection. Oracle8*i* will automatically detect Shift-JIS, JIS7, or EUC for each document and convert it to the database charset if needed.

USER_FILTER

The user filter is a method for plugging in your own filters. You write a filtering program, place it in ?/ctx/bin, and the indexing engine will invoke it to filter each document. Here's an example -- we'll create an uppercase filter which will uppercase every word.
We start by creating a program to do the filtering -- in this case, we'll write a perl script:

```
  #!/usr/local/bin/perl

  open(IN, $ARGV[0]);
  open(OUT, ">".$ARGV[1]);
  while ()
  {
   tr/a-z/A-Z/;
   print OUT;
  }

  close(IN);
  close(OUT);
```

This perl script converts a file to uppercase.
User filter programs like this take two arguments. The first argument is the name of the input file. The second argument is the name of the output file. Our filter reads the contents of the input file, filters it, and writes the output to the output file.
the program (called upcase.pl) is placed in ?/ctx/bin. Ensure that it's executable by the oracle operating-system user. Now the preference is created:

```
    ctx_ddl.create_preference('mypref', 'USER_FILTER');
    ctx_ddl.set_attribute('mypref', 'EXECUTABLE', 'upcase.pl');
```

When this preference is used in an index, the indexing engine will invoke the user filter program for each document when the text is required.

INSO_FILTER

The INSO filter automatically recognizes and filters over a hundred different formats, including Word 97 and Acrobat. The full list can be found in the documentation.
The INSO filter does not have any attributes at this time, so the CTXSYS.INSO_FILTER preference can be used in any index needing filtering.
IMPORTANT: This filter outputs HTML, not plain text. Make sure you employ the HTML section group or all these tags will be indexed. The default system will employ the HTML section group when the INSO filter is detected.
The INSO filter uses code from Inso, "the world leader in filtering and viewer technology," and Adobe. This code is not integrated code directly, it instead follows the user filter architecture. The user filter executable for the INSO filter is ctxhx.
ctxhx requires shared libraries and other files (.flt and .tpt files) located in ?/ctx/lib. The installer should copy the correct platform- specific files to this directory. Additionally, ensure that the external procedure agent has ?/ctx/lib in the LD_LIBARY_PATH environment variable (PATH on NT). This can be done using the ENVS section in the listener.ora. On NT you may need to have ORACLE_HOME set in the ENVS section, as well.See Installation section above for details.
If you are encountering problems using this filter, it can be run by hand. First get a formatted binary file. Let's call it testfile.doc. You do:

```
    ctxhx testfile.doc testfile.txt
```

This should create the file testfile.txt with the HTML representation. The error messages you get, if any, should help you determine what's wrong.
The INSO filter is supported only on Solaris, HP-UX, AIX, and NT. Other platforms are not able to use this filter at this time.

Where's the HTML filter ?

Some of you familiar with previous versions of ConText may be wondering about the HTML filter. In 8i, this code has been moved to the new sectioner class, which allows any filter to spit out HTML if it so desires.

Section Groups

The SECTION_GROUP Class

The section group class is new for Oracle8i, and incorporates functionality that was part of the wordlist and filter classes in ConText. It takes a text format, such as XML or HMTL, as input. It is unique in having two outputs -- the section boundaries, which go to the engine, and the plaintext, which goes to the lexer.

Creating Section Groups

Section groups are not created using create_preference -- they have their own API in ctx_ddl:

```
    ctx_ddl.create_section_group('mygroup','html_section_group');
```

The first argument is the name of the new section group. The second argument is the section group type. This specifies the input text format, and tells the sectioner the rules for detecting section boundaries and transforming to plaintext. Each section group type is covered briefly below.

An empty section group can transform formatted test to plaintext, but will not index any section boundaries. You must first tell it which sections to index by adding sections to the group.

Sections have three important attributes: TAG, NAME, and TYPE. TAG tells the section group how to recognize the section. Because the section group already knows the format, you do not need to specify start and end tags; a section in an HTML section group with TAG of B knows that the section starts with <b> and ends with </b>, for instance. Tags are unique across the sections of a section group.

NAME is how you refer to the section in queries. You may want to name the B tag BOLD, for readability. Multiple tags can be mapped to the same name -- they are then treated as instances of the same section. H1, H2, H3 can all be treated as instances of the HEADLINE section. We recommend avoiding non-alphanumeric characters -- such as underscore -- in section names. Using these characters will force you to escape the names in queries.

TYPE is totally new for 8i. There are three different types of sections: ZONE, SPECIAL, and FIELD.

Section Types

ZONE sections are like sections from previous versions. *inter*Media records where in the document the section start and end tags occur. WITHIN queries check that the hit words occur between the start and end word offset.

If a ZONE section repeats, each instance is treated separately in query semantics. See Examples, below. ZONE sections can enclose other sections, including themselves, and can be enclosed by other sections. ZONE sections are added like this:

```
    ctx_ddl.add_zone_section('groupname', 'sectionname', 'tag');
```

SPECIAL sections are so named because they are not recognized by tags. There are two special sections in Oracle8*i* -- SENTENCE and PARAGRAPH -- both recognized by punctuation in the lexer. They index the start and end of sentences and paragraphs, respectively. You add special sections like this:

```
ctx_ddl.add_special_section('groupname', 'sectionname');
```

No tag argument is needed. There are only two allowed values for section name: SENTENCE, and PARAGRAPH.

ZONE and SPECIAL sections index section start and end word offsets, but do nothing to the words in the section. FIELD sections, on the other hand, extract their contents and index them separately from other words in the document. WITHIN queries are run on this separate, smaller index. This makes field section query faster than zone section query -- up to three times as fast in tests we've conducted -- especially when the section tags occur in every document. This speed comes at the cost of flexibility. FIELD sections are meant for non-repeating, non-overlapping sections. If a field section repeats, it is treated as a continuation of the section, not a distinct instance. If a field section is overlapped by itself or by another field section, it is implicitly closed at the point where the other section starts. Also, there is a maximum of 64 field sections in any section group. This is 64 section names, not tags. Remember that you can map multiple tags to the same section name. You add field sections like this:

```
ctx_ddl.add_field_section('groupname', 'sectionname', 'tag');
```

To illustrate, let's work through a couple of examples to illustrate the different types of sections and the impact on query semantics. We'll use the following document as an example:

```
<A>rat</A><A>ox</A>
<B>tiger rabbit</B>
<C>dragon<C>snake</C></C>
```

This is an XML-style markup, but without a DTD, so we will use the basic section group type:

```
ctx_ddl.create_section_group('mygroup','basic_section_group');
```

Let's start with ZONE sections:

```
ctx_ddl.add_zone_section('mygroup', 'asec', 'a');
ctx_ddl.add_zone_section('mygroup', 'bsec', 'b');
ctx_ddl.add_zone_section('mygroup', 'csec', 'c');
```

This tells the section group to recognize A tags as the section ASEC, B tags as the section BSEC, etc. To do section queries, we use the WITHIN operator. Remember to use the section name and not the tag:

```
contains(text, 'rat within asec') > 0
```

This finds the document.

```
contains(text, 'tiger within asec') > 0
```

This does not find the document. Although it has "tiger", it does not occur in the ASEC. If instead of the original setup we had mapped A and B to the same section name:

```
ctx_ddl.add_section('mygroup', 'asec', 'a');
ctx_ddl.add_section('mygroup', 'asec', 'b');
```

Then both:

```
contains(text, 'rat within asec') > 0
contains(text, 'tiger within asec') > 0
```

would find the document, because both A and B are treated as ASEC.

An important facet of ZONE section query semantics is that each instance is treated distinctly. That is, a query like this:

```
contains(text, '(tiger and rabbit) within bsec') > 0
```

finds the document, but a query like this:

```
contains(text, '(rat and ox) within asec') > 0
```

does not find the document. Although the document has "rat" and has "ox", and "rat" is in ASEC, and "ox" is in ASEC, "rat" and "ox" are not within the SAME ASEC. Note that

```
    contains(text, '(dragon and snake) within csec') > 0
```
finds the document, since they are both in the outer CSEC, even though the inner CSEC contains only "snake".

Special sections follow the same semantics, so a query like this:
```
    contains(text, '(jumbo and shrimp) within sentence') > 0
```
is looking for documents with "jumbo" and "shrimp" in the SAME sentence.

Now let's look at field sections. What if in setup we made ASEC a field section instead of a zone section:
```
    ctx_ddl.add_field_section('mygroup', 'asec', 'a');
```
Unlike zone sections, each instance of a field section is considered a continuation of the section. The query
```
    contains(text, '(rat and ox) within asec') > 0
```
finds the document, although it didn't when ASEC was a zone section. The field section simply stitches the two instances together. Field sections work best on sections like BSEC, which does not repeat like ASEC nor nest like CSEC.

One last subtlety with field sections. If ASEC is a zone section, then
```
    contains(text, 'rat') > 0
```
finds the document -- it contains the word rat. However, remember that field sections work by extracting the document contents and indexing them separately. Thus, if ASEC were a field section, this query would not find the document. "rat" is in ASEC, which is separate from the document contents.

You can, however, change this by making the field section visible. There is an optional boolean fourth argument to add_field_section. If this boolean argument is set to true:
```
    ctx_ddl.add_field_section('mygroup', 'asec', 'a', TRUE);
```
then the field section is visible, and the section contents are visible to non-within queries, just like zone sections. This is accomplished by double- indexing the word -- once as part of the extracted section, and once as part of the body, so this option has index space cost.

Using Section Groups

Section groups are plugged into indexes using the section group keyword:
```
    create index
    ...
    parameters ('section group mygroup');
```
Unlike previous versions, you do not need to set an attribute in the wordlist object. You also no longer need to set STARTJOINS and ENDJOINS in the lexer. However, the WHITESPACE, NEWLINE, and PUNCTUATIONS settings in the basic lexer affect sentence and paragraph boundary recognition, which then impact special sections. There are five section group types.

NULL_SECTION_GROUP

The null section group is used when there is no section information to be extracted -- it simply passes the text through to the lexer. Only special sections can be added to the null section group.

BASIC_SECTION_GROUP

The basic section group is for simple XML-style markup without going full- bore on the DTD stuff. It can be used in custom situations where full XML is unnecessary. Tags are in the form .... Entities and tag attributes are not supported. The only processing it does for plaintext is removal of the markup tags.

HTML_SECTION_GROUP

The HTML section group is for HTML. Good name, huh? It knows the rules for HTML, including ISO Latin-1 entity translation, HTML to plaintext conversion, and
omission. It knows HTML 4.0 tags, and has code to deal with unknown tags.
SCRIPT and STYLE contents, and comments are removed. Contents of the TITLE section are preserved. META tag information indexing is a known customer requirement for a future version.
For those familiar with previous versions, there is no need for KEEP_TAG anymore -- this is automatic when you add the tag to the section group.

XML_SECTION_GROUP

The XML section group is for XML. It does no processing to plaintext except for entity translation and tag removal. It can handle non-system, non-parameter entities in the internal DTD. It cannot access external DTD's in this version.

NEWS_SECTION_GROUP

The news section group handles newsgroup-style postings -- RFC-1036 in particular. This format is used in newsgroups and is similar to the one used for e-mail. Note that while RFC-1036 defines the set of allowable header fields, Oracle does not require this -- any header fields can be used.
Messages in this format consist of a header section, a blank line, and a body section. The news section group recognizes <beginning of line>TAG: as the start of sections, and <end of line> as the end of the section. It stops looking for this pattern when the blank line is encountered.
Translation to plaintext consists of removing all header lines which are not important to section searching. A message like this:

```
  From:  me
  To:    you
  X-ref: 5!oowj

  hello!  How are you!
```

with only the from and to tags in the section group, will look like this to the indexing engine:

```
  <from section start>me<from section end>
  <to section start>you<to section end>
  hello!  How are you!
```

Lexers

The LEXER Class

The lexer class gets plaintext from the sectioner and splits it into words.
Lexer preferences are plugged into indexes using the lexer keyword:

```
    create index
    ...
    parameters ('lexer mylex');
```
The lexer class has five objects.

BASIC_LEXER

The basic lexer can be used for most European languages. It is programmed with default rules for splitting whitespace-delimited text into words. You can modify its parsing behavior to some extent with the lexical attributes.
Ordinarily, non-alphanumeric characters end a word. SQL*Plus is lexed as "SQL" and "Plus". The JOINS attributes allow you to modify this behavior by declaring sets of non-alphabetic characters to treat as valid word letters.
PRINTJOINS are non-alphanumeric characters which are treated as valid characters, and become part of the word. Declaring PRINTJOINS of "*" lets SQL*Plus be lexed as "SQL*Plus".
SKIPJOINS are like PRINTJOINS, but they do not become part of the word. Declaring SKIPJOINS of "*" lets SQL*Plus be lexed as "SQLPlus".
STARTJOINS are like PRINTJOINS, but they only come at the start of the word. If they occur in the middle of a word, a new word is started. Having "*" as STARTJOINS lets SQL*Plus be lexed as "SQL" and "*Plus". Multiple STARTJOINS are allowed at the start of a word.
ENDJOINS are like STARTJOINS, but for the end of the word. They implicitly end a word, too. So, "*" as ENDJOINS lexes SQL*Plus as "SQL*" and "Plus". Multiple ENDJOINS are allowed at the end of a word. STARTJOINS and ENDJOINS used to be important for section searching in previous versions, but with the new sectioner class, they are no longer needed for that purpose.
Each of these four JOINS attributes is a set of characters. Setting PRINTJOINS to "_%*" means that "_", "%", and "*" are all PRINTJOINS.
NUMJOIN is a single character which is the numeric join character. NUMGROUP is the single numeric group character. They are defaulted based on your NLS_LANG setting. For US, NUMJOIN is "." and NUMGROUP is ",".
Finally, CONTINUATION are those characters which indicate line continuation. These and the following newline are removed from the text. Default value is hyphen and backslash.
Then there are three attributes important for sentence/paragraph indexing. PUNCTUATIONS are the set of punctuation marks (?!. by default). WHITESPACE is a set of additional whitespace characters other than space and tab, which you get for free and cannot change. NEWLINE can be set to NEWLINE or CARRIAGE_RETURN. This controls whether lines are ended by \n or \r\n.
A sentence is then recognized as a PUNCTUATION followed by one or more WHITESPACE. A paragraph is a PUNCTUATION followed by a NEWLINE. There are other patterns which are variations on this, but that's the basic idea.
The third set of basic lexer attributes controls term normalization -- the process of converting words to a standard form. BASE_LETTER is a boolean attribute which, if set to YES, will convert accented characters to their unaccented forms.
By default, each word is uppercased during indexing so that queries are case-independent. MIXED_CASE is a boolean, which, if set to YES, does not do this. This makes indexing and queries case-sensitive.
ALTERNATE_SPELLING is an attribute which, if set, uses letter transformation rules for common alternate spellings. It can be set to GERMAN, DANISH, or SWEDISH. In GERMAN mode, for instance, A-umlaut gets transformed to AE.

Since the lexer is used at both indexing and query time, query for a word containing A-umlaut will find the word spelled with A-umlaut or AE.
Finally, COMPOSITE controls word decompounding. In German, for instance, multiple words are often combined into a single string. This makes lexing difficult, because the words are no longer whitespace-delimited. If COMPOSITE is set, then the lexer breaks up these compound words, which allows *inter*Media to index and query as usual. COMPOSITE can be set to GERMAN or DUTCH. Unset or DEFAULT means no decompounding.
The last set of basic lexer attributes control text and theme indexing. New for 8i, the basic lexer can generate and index themes for each document, in addition to splitting it up into words. This merges the functionality found in the separate THEME LEXER and BASIC LEXER of previous versions.
INDEX_TEXT is a boolean which controls word indexing. INDEX_THEMES controls theme indexing. Setting INDEX_THEMES to NO makes it a text-only index. Setting INDEX_TEXT to NO makes it a theme-only index. They cannot, of course, be both NO. Indexing themes takes longer and uses a bit more index space, but improves the efficacy of ABOUT queries. We'll talk more about this when we discuss ABOUT query.
The CTXSYS.DEFAULT_LEXER preference has theme indexing ON for the English language-specific scripts.

JAPANESE_VGRAM_LEXER

Japanese is not a whitespace-delimited language (except in textbooks) so it is very difficult for computers to pick out individual words. The Japanese V-gram lexer gets around this problem by indexing overlapping clusters of characters. The Japanese word ABC might be decomposed into AB and BC, for instance. Query terms are also decomposed this way. Thus, the contains query is not looking for words, but for patterns of characters.
The Japanese lexer does not have any attributes in this version.

CHINESE_VGRAM_LEXER

Chinese is also not whitespace-delimited, so a similar solution has been applied for Chinese, as well. The Chinese v-gram lexer also does not have any attributes.

CHINESE_LEXER

New for Oracle8*i* is a segmenting Chinese lexer, which can actually recognize some Chinese words as whole entities. The rest of the text is still broken into v-grams, but this should be more space-efficient and have faster query than the pure v-gram method. The Chinese segmentation lexer does not have any attribtues.

KOREAN_LEXER

Korean is whitespace-delimited, but has problems with verbs, which can have thousands of different forms. Our Korean lexer is a lexicon-driven engine (using a third-party 3-soft lexicon) which simply eliminates verbs from indexing. New for Oracle8i is the ability to eliminate adverbs and adjectives, do various form conversions and perform morphological and segmentation decomposition.

Other Indexing Classes

The STOPLIST Class

Not every word in a document is worth indexing. Linguistic lubricant like prepositions and conjunctions are important for language understanding, but are not very useful for information retrieval -- they are very common and so convey very little information about the document by their presence. Most of the time, it's not worth the space to index these words.
The stoplist class holds a list of these words, called stop words. During indexing, the engine consults the stoplist and filters out the stop words. The stoplist class does not use the create_preference API. Instead, it has its own API:

```
ctx_ddl.create_stoplist('mylist');
```

Words are added using add_stopword:

```
ctx_ddl.add_stopword('mylist', 'the');
```

Here the article "THE" has been added to the stoplist. You can see the words in a stoplist using the ctx_stopwords view. A stoplist can have a maximum of 4095 stopwords. Each can be up to 64 bytes in length.
Stopwords are case-sensitive in a stoplist, but if the index is not case-sensitive, the stopwords will not be case-sensitive. So, you could add "THE" and "the" to a stoplist, but if the index is not case-sensitive, there would be no difference -- you'd just be wasting a stopword slot.
Stoplists are plugged into indexes using the stoplist keyword:

```
create index
...
parameters ('stoplist mylist');
```

New for Oracle8i, stopwords can be added to the index without re-indexing:

```
alter index myidx rebuild parameters ('add stopword AND');
```

This adds "AND" as a new stopword after indexing. Remember that the index makes its own copy of everything, so this does not affect the stoplist used to create the index.
The default stoplist, CTXSYS.DEFAULT_STOPLIST, is language-specific, and default stoplists are provided for most European languages.

Other Stop Objects

Stoplists can also hold two other kinds of objects: stop themes and stop classes. Stop classes are classes of words, rather than individual words. The only stop class available in Oracle8*i* is NUMBERS, which stops all numbers from being indexed. Alphanumeric words, such as 123.456.789 and abc231 are still indexed. This behavior depends on the lexer to recognize numbers.
Stop themes are used only during theme indexing. The entries added as stop themes are themed, and these themes AND THEIR CHILDREN are not indexed. For instance, say you enter "biology" as a stop theme. This will prevent themes like "Physical Sciences:Biology:Genetics" and "Physical Sciences: Biology:Botany" from being indexed. This is just an example, and does not actually reflect the knowledge base.

Stop themes are used only during theme indexing, so adding "biology" as a stop theme will not stop the word "biology" or biology-associated words from being indexed.

The WORDLIST Class

The wordlist class is an odd class because it has no effect on indexing. Instead, it holds fuzzy and stem expansion settings used at query time.
Wordlist preferences are plugged into indexes using the wordlist keyword:

```
   create index
   ...
   parameters ('wordlist mylist');
```

There is only one wordlist object: BASIC_WORDLIST, with the following attributes:

STEMMER	This attribute is set to the default stemming type. Stemming is an expansion of a word to different forms. The stem expansion of GO might include GO, GOING, WENT, and GONE. The rules for this vary from language to language. (See expansion query operators). STEMMER can be set to one of the following: ENGLISH (inflectional), (English) DERIVATIONAL, DUTCH, FRENCH, GERMAN, ITALIAN, SPANISH, and NULL, which means no stemming. The CTXSYS.DEFAULT_WORDLIST setting is language-specific.
FUZZY_MATCH	This attribute is set to the default type of fuzzy match. Fuzzy match is an expansion technique designed to find words close in form, such as mistyped or mis-OCR'ed versions. FUZZY_MATCH can be set to one of the following: GENERIC, JAPANESE_VGRAM, KOREAN, CHINESE_VGRAM, ENGLISH, DUTCH, FRENCH, GERMAN, ITALIAN, SPANISH, and OCR. The CTXSYS.DEFAULT_WORDLIST setting is language-specific.
FUZZY_SCORE	This is the default score floor for fuzzy expansions. New for Oracle8i is the ability to limit the fuzzy expansion to the best matches. The fuzzy score is a measure of how close the expanded word is to the query word -- higher is better. Setting fuzzy score means that fuzzy expansions below this score will not be produced. You can set this from 0 to 80.
FUZZY_NUMRESULTS	This is the default maximum number of fuzzy expansions. Setting fuzzy numresults limits the fuzzy expansion to a certain number of the best matching words. You can set this up to 5000.

The STORAGE Class

The storage class holds storage parameters for the index tables. The Text index is made up of five oracle objects, so the single storage clause in the create index statement is insufficient to specify storage. Storage preferences are plugged into indexes using the storage keyword:

```
   create index
   ...
   parameters ('storage mystore');
```

There is only one storage object: BASIC_STORAGE, with the following attributes:

I_TABLE_CLAUSE	This attribute is the storage parameters for the I table, which is the main index table. There will be at least one row per unique word in your index -- probably multiple rows per word. Each row has a BLOB which tries to stay inline, making the row a bit large. The value is simply tacked on to the end of the create table statement, so you could set it like this:
	`ctx_ddl.set_attribute('mystore', 'I_TABLE_CLAUSE',`
	`'tablespace foo storage (initial 1K)');`

K_TABLE_CLAUSE	This attribute is the same, but for the K table, which is an IOT for rowid to docid conversion. There will be one row per row in your base table. Each row consists of a number and a rowid.
R_TABLE_CLAUSE	This attribute is the same, but for the R table, which is a BLOB table for docid to rowid conversion. There will be only a few rows in this table, and will have a large BLOB in each row.
N_TABLE_CLAUSE	This attribute is the same, but for the N table, which has invalid docids waiting for garbage collection. The number of rows in this table will depend on the frequencies of update and delete DML, and of garbage collection optimization. Each row is short, holding two numbers.
I_INDEX_CLAUSE	This attribute is the same, but for the unique index on the I table.

Indexing Memory

Those of you familiar with previous versions may be wondering where the index memory setting is. For new users, *inter*Media builds an inverted index, so documents are not added to the index one at a time, like a b-tree index. Instead, information from multiple documents is buffered up and periodically flushed to the index during indexing. In previous versions, you could set the size of this buffer in the analogue of the storage class. Setting this buffer higher reduces index fragmentation but uses more memory.
In Oracle8i, index memory can be set directly in the parameters string. This allows you to set index memory on a per-operation basis. The keyword used is MEMORY:

```
create index
...
parameters ('memory 2M');
```

Here a paltry 2 megabytes of memory is used for indexing. If not specified, the system parameter DEFAULT_INDEX_MEMORY is consulted for the systemwide default index memory. At install time, this is 12M.
The amount of memory cannot exceed system parameter MAX_INDEX_MEMORY. This allows the system administrator to disallow outrageous amounts of index memory.
The MEMORY keyword is also allowed in the alter index parameter string for batch DML (background DML uses DEFAULT_INDEX_MEMORY). This allows you to have different memory sizes for create index and DML, something which is not possible in previous versions. You simply specify a different memory amount in the create index parameter string then in the alter index parameter string.

DML

DML Processing

Most text search engines have asynchronous document updates and additions -- changes to the index are usually done in batches, after and separately from the document changes. This is due to three aspects of inverted indexing:

1. Text indexing a single document is a lot of work. The values are usually long, and at the very least you have to break it into words. Depending on the objects, you may also have to do web page requests, filtering, and HTML parsing.
2. Inverted indexes, composed of lists of documents by word, are best updated in batches of documents at a time. Updating the index one document at a time leads to either word lists one document long or a lot of appending to existing lists.
3. Most text applications are fairly static, having relatively lower DML frequency, and less need for full consistency semantics than traditional transaction processing systems.

*inter*Media faces the same challenges. Here is what it does for each DML on an indexed column:

INSERT	The document rowid is placed into a queue, dr$pending, for later addition to the text index. Queries before this DML is processed will not find the new document contents.
UPDATE	The old document contents are invalidated immediately, and the document rowid is placed into the dr$pending queue for later reindexing. Queries before this DML is processed will not find the old contents, but neither will it find the new contents.
DELETE	The old document contents are invalidated immediately. No further DML processing is required -- queries will no longer find the document.

Thus, *inter*Media has synchronous invalidation but asynchronous addition. This extends to transactions, as well:

```
  delete from table where contains(text, 'delword') > 0;

  select * from table where contains(text, 'delword') > 0;
  --> you get no rows returned

  rollback;

  select * from table where contains(text, 'delword') > 0;
  --> you get the rows returned
```

and:

```
  insert into table values (1, 'insword');

  select * from table where contains(text, 'insword') > 0
  --> you do not get the row

  commit;

  select * from table where contains(text, 'insword') > 0
  --> you still do not get the row
```

The synchronous invalidation is new to this version. Also new is that the DML notification is done through integrated kernel code, and not through triggers as in previous versions.

Processing Additions

So how do you process your inserts and updates when they are sitting in the queue? *inter*Media provides two methods: sync and background.
Sync is the manual way -- you control when and how it's invoked. Each invocation processes all pending inserts and updates for an index:

```
    alter index myindex rebuild online parameters ('sync')
```

The ONLINE keyword is very important. Without this, during the sync operation queries are blocked and DML on the base table fails. Make sure to use it for sync operations.

Background DML requires you to start a ctxsrv background daemon in the operating system:

```
ctxsrv -user ctxsys/
```

Once started, the daemon polls the dr$pending queue for DML, and automatically processes additions as they become available.

You can use either or both -- it's largely a matter of your application requirements. Background DML scans for DML constantly. This leads to new additions being indexed automatically and quickly. However, it also tends to process documents in smaller batches, which increases index fragmentation. Sync DML is done at the user's request, so usually the batches are larger and thus there's less index fragmentation. A happy compromise could possibly be reached by invoking sync through dbms_sql in a dbms_job which runs at longer intervals.

The DML Queues

The dr$pending queue holds the documents waiting to be indexed. It should be queried through the ctx_user_pending view, which makes it more readable. Additionally, there is a dr$waiting queue which is used when documents are waiting to be indexed and they are updated again. The drq_inprog, drq_batches, and drq_batchno tables from previous versions are all no longer needed.

Optimization

Index Fragmentation

The Text index is an inverted index, so essentially consists of words and lists of documents which contain that word. When new documents are added to this index, these lists have to be extended. Instead of appending to these lists, more rows for the word are created. However, this can lead to many rows with sub-optimal, short lists -- called "index fragmentation". For instance, say we have the following in our index:

```
DOG   DOC 1 DOC 3 DOC 5
```

Now a new document with "dog" comes along. Simply add it to the index as a new row:

```
DOG   DOC 1 DOC 3 DOC 5
DOG   DOC 7
```

Subsequent DML will also create new rows:

```
DOG   DOC 1 DOC 3 DOC 5
DOG   DOC 7
DOG   DOC 9
DOG   DOC 11
```

This is why background DML generally produces more index fragmentation than spot invocation of sync: processing DML less frequently in larger batches allows newly added rows to have longer lists, which reduces the number of rows in the index table.

Keep in mind that a certain amount of fragmentation is expected in 8i. *inter*Media will try to keep index table rows below 4000 bytes long, to inline the LOB and to speed up index row merges used during phrase search, AND, section search, etc. However, frequent DML will probably result in much more fragmentation than this.

Document Invalidation

During an update or delete, the old document information must be removed from the index so that queries will no longer hit it. However, because the inverted index consists of words and rows of occurrences, this would entail finding all words in the old version of the document (which may not be available), and removing an occurrence from the lists of those rows in the index table. This is too much work to do synchronously, and deletes must be synchronous in the integrated model.
Instead, *inter*Media marks the old document contents as invalid and does not touch the index. The mark signals queries to remove the document from any query result sets. However, this leaves the old information in the index, taking up space in the index tables.

Optimization

To solve these potential problems in the index, you run optimization. Optimization has two modes: FAST and FULL. FAST optimization targets fragmentation only:

```
    alter index myindex rebuild online parameters ('optimize fast');
```

The ONLINE keyword is important -- without it, queries are blocked and DML fails with an error. FAST optimization runs through the whole index table and glues fragmented rows together, to a maximum of 4000 bytes for a single row. This reduces the number of rows in the index table.
FULL optimization does both defragmentation and garbage collection, which removes the old information left over after document invalidation:

```
    alter index myindex rebuild online parameters ('optimize full');
```

It loops through each row of the index table, but, unlike fast optimization, cracks open each row's list, removing old document information. It also glues separate rows together where possible.
Because garbage collection is more involved and time-consuming than defragmentation, FULL optimization does not have to be run on the whole table at one time. Instead, you can run it for a limited period of time:

```
    ... parameters ('optimize full maxtime 5')
```

which means run for a maximum of 5 minutes. At the end of 5 minutes, it will stop. The next time you run optimization, it will pick up where it left off. This lets you do a little of bit optimization each night during off times, and ensure that it is not loading the system when the off time ends.
Garbage collection is needed only after document invalidation, which happens only for delete and updates. If your document table is append-only, meaning that documents are inserted and never changed or removed, then it should suffice to run only defragmentation.

Optimization Concurrency

Unlike previous versions, optimization (either mode) does not block background DML processing. However, because Oracle prevents alter index operations from running concurrently, sync invocations will be prevented.

Querying

The Basics

At this point all aspects of building an index and keeping it up to date have been discussed. However, indexes are a means to enabling content- based queries.
IMPORTANT: Unlike a value query, which is slower without a b-tree index, contains queries are completely disallowed without a Text index.
You do a query using the contains operator:

```
select id
  from texttab
 where contains(textcol, 'query') > 0
```

The first argument to contains is the name of the text column. The second is the text query, which is limited to 2000 bytes in length. It returns a number, which indicates the strength of match. 0 means not a match, so it is constrained to > 0. The result is that this query finds the id's of all rows in texttab where the textcol has the word "query".

Scoring

Of course, now you're wondering how to use that return value in the select list and order by clause. An operator is provided just for that purpose:

```
select id, score(1)
  from texttab
 where contains(textcol, 'query', 1) > 0
 order by score(1) desc
```

The "1" in the score and contains operators is called the "contains label" and it's a number which links the score in the select list to the contains in the where clause -- it can be any number, so long as it matches.
Score can be between 0 and 100, but the top-scoring document in the query will not necessarily have a score of 100 -- scoring is relative, not absolute. This means that scores are not comparable across indexes, or even across different queries on the same index. Score for each document is computed using the standard Salton formula:

```
3f(1+log(N/n))
```

Where f is the frequency of the search term in the document, N is the total number of rows in the table, and n is the number of rows which contain the search term. This is converted into an integer in the range 0 - 100.
Query operators like AND and OR operate on the scores of their operands.

Where Can I Use contains?

Unlike the transparent query rewrite of previous versions, you can include a contains clause in any SQL statement using any tool which can issue oracle SQL. Here is a short list of new places where contains queries can pop up:

- subqueries
- virtual tables
- PL/SQL
- View definitions
- DML (insert as select, e.g.)

There is no longer a limitation on the length of the contains SQL statement, although there is a limit on the query term length.

Other Query Methods

For those of you familiar with previous versions, there is no two-step or text cursors (n-memory query). If you absolutely must have result tables, use insert as select. Instead of text cursors, use real PL/SQL cursors. To sort by score, use an order by clause with a FIRST_ROWS hint. The hint will avoid an oracle SORT ORDER BY.
However, *inter*Media still supports count_hits. count_hits is a fast way to get the hit count for a text query, in PL/SQL:

```
   num_hits := ctx_query.count_hits('indexname','query',TRUE);
```

The first argument is the name of the index. This should be a string, so use quotes if you are using a literal value. The second argument is the query string, and the third is a boolean toggle called "exact". If exact is TRUE, then the result is accurate. If exact is FALSE, then the result is only an upper bound. The actual count could be lower. exact FALSE is faster, because it doesn't screen out invalid documents (see issues on DML and optimization for more details).
You could also do "select count(*)" with a contains clause, and this is recommended when you have structured conditions, but count_hits is faster for text-only queries.

Simple Queries

The simplest query is a single word:

```
   contains(text, 'dog') > 0
```

You can escape your word with curlies:

```
   contains(text, '{dog}') > 0
```

This is useful when you want to query on a reserved word or your word has special characters which are query operators. The list of reserved words and the query operator characters are in the documentation.
You can query on a phrase just as easily:

```
   contains(text, 'dog my cat') > 0
```

Unlike internet search engines, this searches for the phrase "dog my cat", not just any document which has these words.
If your phrase contains a stopword:

```
   contains(text, 'dog the cat') > 0
```

then the stopword ("the" in this case) is treated as a wildcard, matching any word. This query would match all the following documents:

```
   dog the cat
   dog my  cat
   dog cat cat
```

but not this document:

```
   dog cat frog
```

In other words, the stopword must match something. Stopwords alone disappear from the query.
The query:

```
   contains(text, 'the & cat') > 0
```

is reduced to

```
   contains(text, 'cat') > 0
```

Queries on just stopwords will return no rows.

Query Operators (Part 1)

Boolean Operators

Boolean operators are used to combine the results of subqueries using boolean algebra. *inter*Media has three boolean operators:

AND (&) OR (|)	AND and OR are the basic boolean operators. However, the operands are returning numeric scores, rather than boolean values, so we've implemented AND as a minimum of its operand scores, and OR as a maximum of its operand scores. This results in expected behavior -- if any operand of AND scores 0 (the word is not in the document) -- then the AND retuns 0 (the document is not returned). You can use either word or symbol: 'dog and cat' is the same as 'dog & cat'.
NOT (~)	NOT is not the unary negator. Instead, it is "AND NOT". 'dog NOT cat' returns all document which have "dog" except those which also have "cat". The score returned is the score of the left child.

Subqueries and Grouping

You can use parentheses for subqueries and precedence, as in:

```
(dog my cat AND fido) OR horse
```

Without parentheses, the operators follow a precedence hierarchy which can be found in the documentation. Since AND is higher precedence than OR, the parens are not actually needed in the example above. However, they are needed in this query:

```
dog my cat AND (fido OR horse)
```

in order to override the precedence hierarchy.

Scoring Operators

The scoring operators operate on the scores of their operands. There are four scoring operators:

WEIGHT (*)	The weighting operator multiplies a search term's score to make it more or less important in the query. The multiplier can be from .1 to 10: `contains(text, '(dog*2) AND cat') > 0` This query looks for documents with "dog" and "cat", but the score for "dog" is multiplied by 2. This makes documents with "dog" more likely to have a higher score in the result set than those with "cat". WEIGHT does not have a word equivalent -- you must use the character *.
THRESHOLD (>)	The threshold operator filters out documents below a particular score. 'dog > 10' will result only those documents containing "dog" which score higher than 10. You can also do it this way: `contains(text, 'dog') > 10` but threshold is implemented at a lower level, so will be faster. THRESHOLD does not have a word equivalent -- you must use >.
MINUS (-)	The minus operator takes the left operand score and subtracts the right operand score. 'dog - cat' looks for documents with "dog", but those documents which also have "cat" will probably score lower than those without. This is not the same as NOT, which completely filters out the document if the right operand search term is present.
ACCUM (,)	Accumulate groups several words or phrases, and scores higher when more of its operands are hit. 'dog, cat, frog' will score any document with all three words higher than any document with two of the three, which will score higher than any document with only one of the three. This is changed from previous versions, where ACCUM merely added the scores of its operands. You can either use the symbol ,

	or the word ACCUM.

Set Operators

For those of you familiar with previous versions, the MAXDOC (:) and FIRST/NEXT (#) operators have been removed. You can use cursor fetching loops to get the same effect.

Word Expansion Operators

Word expansion operators expand a word to find similar forms of the word in the document set. For Oracle8i, the limit on maximum number of expansions has been removed. However, the more words a term expands to, the slower the query will run. There are five word expansion operators:

WILDCARD (% _)	You can use the SQL like wildcards % and _ to find words matching a pattern. 'do %', for instance, finds all documents with words beginning with do, such as dog, door, etc. This is done via a like query on the index table, so a wildcard query like '%do%' will be slow, as a full table scan of the index table is required.
FUZZY (?)	Fuzzy query finds words with similar form, using a proprietary algorithm. This is useful for finding mis-typed or mis-OCR'd words. The fuzzy operator is ?, as in '? dog'. You can change the rules used for fuzzy and limit the expansion using the wordlist class. FUZZY has no word equivalent -- you must use the ? operator.
STEM ($)	Stem query finds words with the same stem form, using integrated Xerox linguistic code. This is useful for finding "GOING" and "WENT" from "GO", for instance. You can change the settings for stem expansion using the wordlist class preferences. STEM has no word equivalent -- you must use the $ operator, as in '$go'.
SOUNDEX (!)	Soundex query finds words which sound alike. This no longer uses a separate table, but is instead a specialized fuzzy expansion. SOUNDEX has no word equivalent -- you must use the ! operator, as in '!dog'.
EQUIV (=)	Equivalence is for manually inputting the various forms of a word, as in 'dog = cat', searching for documents with 'dog' or 'cat', treating them as different forms of the same word. Equiv only works on simple words -- you cannot equiv phrases. You can either use the symbol = or the word EQUIV.

Proximity Operator

Other than phrases, *inter*Media has one proximity operator -- NEAR, which finds documents where the input phrases are close to each other. The closer the words are to each other, the higher the score. NEAR has two forms. The shorthand form uses the ; character;

```
dog ; cat ; boat
```

This query finds documents which have dog, cat, and boat, and scores the document higher the closer they are to each other.

The function form has options to set range and directionality:

```
NEAR((dog,boat), 10, TRUE)
```

The first argument to NEAR is the list of words. In the example, we are looking for dog and boat. The second argument is the maximum allowed span. In the example, we are constraining it to within a 10-word span. That means there cannot be more than 10 words between "dog" and "boat". This range can be up to 100. It is optional -- if omitted, then it defaults to 100. The third argument is directionality. If TRUE, as in the example, then the words have to appear in the order specified in the list. If FALSE, the words can appear in any order, as long as they fit within the specified word span. This can be omitted, in which case FALSE is the default.

Query Operators (Part 2)

Within

The within operator limits a subquery to a particular section:

```
dog within title
```

will find all documents with "dog", but only if it occurs in the "title" section. The name of the section is used rather than the tag. The "title" section has to have defined in your section group -- you cannot specify arbitrary tags. If your section name has non-alphanumeric characters, you should enclose the name in curlies:

```
dog within {my_title}
```

which is why it is't recommended to use non-alphanumeric characters in section names. Within is high on the precedence hierarchy, so if you have a subquery, it is safer to use parentheses to group it:

```
(dog and cat) within title
```

If you don't use parentheses, it will search for documents with "dog" anywhere and with "cat" in the title section:

```
dog and cat within title     ==     dog and (cat within title)
```

which is a different query.

If the section is a zone or special section, the subquery is constrained to a particular instance of the section:

```
(dog and cat) within sentence
```

means dog and cat within the SAME sentence. This is different from:

```
(dog within sentence) and (cat within sentence)
```

for instance -- which means dog in any sentence, and cat in any sentence. Section types and query semantics were covered in issue #7.

Within does not nest, so this is not allowed:

ABOUT is an advanced query operator. We take the input to ABOUT and do our best to increase precision and recall. If your index has a theme component (INDEX_THEMES is YES for BASIC_LEXER) then ABOUT does a theme query. It determines the theme of the input term, and finds documents with the same theme. This allows a query like this:

```
contains(text, 'about(canines)')
```

to find relevant documents even if they don't have the word "canines" in them -- they might have "dogs" or "wolves" instead, for instance. It will also filter out documents which have the word "canines" but are not significantly about canines, such as an article on dentistry, for instance.

The input to ABOUT does not have to be a single term. It can be an unstructured phrase:

```
about(japanese banking investments in indonesia)
```

This phrase will most likely have multiple themes -- banking, Japan, etc. The documents retrieved by this query will have all those themes. This allows you to use plainly-worded descriptive phrases instead of rigid query syntax and still find relevant documents.

If your index does not have a theme component, then ABOUT will do a query expansion to increase recall -- it will split the input into words, stem each, then string it back together using accumulate:

```
   about(go home now)
```

is transformed to

```
   $go,$home,$now
```

Note that the purpose of ABOUT is to have *inter*Media do what it can to find the best-fitting documents. It does not have a formulaic definition like the other operators.

ABOUT can be combined with other operators:

```
   contains(text, 'about(dogs) and $cat')
```

for instance. This allows you to do a combined theme and text search, which was not as easy to do in previous versions.

Thesaurus Operators

The thesaurus operators allow you to expand a thesaurus phrase using a thesaurus. The SYN operator, for instance, expands a word into synonyms:

```
   SYN(dog)   ==    {dog} | {mutt} | {canine}
```

The expansion uses a previously loaded thesaurus. This is usually a thesaurus of your own creation -- *inter*Media does not install a thesaurus by default, although a sample thesaurus is included in ?/ctx/sample/thes. You can see the expansion by using the ctx_thes package functions:

```
  declare
    output varchar2(80);
  begin
    output := ctx_thes.syn('dog');
    dbms_output.put_line(output);
  end;
```

Thesaurus operators take simple thesaurus terms as input -- they cannot nest or take expansions as input:

```
   SYN($dog)        <-- BAD
   SYN(BT(dog))      <-- BAD
```

SQE

SQE's are Stored Query Expressions -- a macro operation. You store a query string and give it a short name, using the ctx_query PL/SQL package:

```
   ctx_query.store_sqe('cats', 'cat = feline = kitty');
```

The first argument is the name of the SQE -- "cats". The second argument is the stored expression. When I use it in a query:

```
   contains(text, 'SQE(cats)')
```

*inter*Media expands it to the stored query expression, making this equivalent to:

```
   contains(text, 'cat = feline = kitty')
```

SQE text can even refer to other SQE's:

```
   ctx_query.store_sqe('animals', 'frog | hog | sqe(cats)');
```

Although if you set up a circular reference, you will get an error at query time. There are fundamental changes to SQE in 8i. First, stored query expressions are no longer tied to an index -- you can use any SQE in any query on any index. You can even use other user's SQE's by prepending the owner name:

```
   SQE(bob.cats)
```

Second, SQE partial results are no longer stored. Instead, SQE works as a macro operation, replacing lexically during query parsing. Finally, there is no longer the notion of session-level SQE -- store_sqe does a commit.

PL/SQL

For those of you familiar with previous versions, the PL/SQL operator (@) has been removed. This is due mainly to stricter transactional restrictions resulting from tighter integration with the kernel

Thesaurus

Overview of Thesaurus Functionality

*inter*Media provides a program to load formatted, file-based thesauri into ctxsys- owned tables in the database. These thesauri can then be used at query time to expand query terms for improved recall. *inter*Media also provides PL/SQL functions for browsing the thesaurus while it's in the database, and a program for exporting the thesaurus back out to a file.
These thesauri are usually customer-defined. *inter*Media provides a sample thesaurus in ?/ctx/sample/thes, but this is not imported by default.

The Thesaurus File

The first step in thesaurus import is construction of the thesaurus file. This is a list of terms in the thesaurus, each followed by their relationship to other terms. For instance:

```
canine
  SYN mutt
  BT  mammal
  NT  wolf

lizard
  BT  reptile
```

The first term in this thesaurus is "canine". The second line declares that "canine" has a synonym, "mutt". The following lines declare that the broader term of "canine" is "mammal", and a narrower term of "canine" is "wolf".
Terms, such as "canine" or "lizard" in our example above, must be on their own line, and must be at the start of the line -- no whitespace is allowed between the beginning of the line and the term.
Relationship lines, such as "SYN mutt", must have whitespace before them. The relationship word (SYN) is not case-sensitive. You cannot have more than one relationship word / word pair on a single line:

canine

Relationships cannot be on the same line as the term:
These rules are required for proper parsing of the thesaurus file. The more relaxed rules in previous versions were not able to correctly parse certain cases, especially when terms had common substrings with relationship words, such as "NT 4.0".

Thesaurus Relationships

The thesaurus file supports the following set of relationship words:

SYN	Synonymity. A SYN B means that A and B mean the same thing. A SYN B implies B SYN A. A SYN B and B SYN C implies A SYN C.
UF	Use For. Same as SYN.
PT	Preferred term. A PT B means that B is the preferred term for A. There can be only one preferred term for any thesaurus term. If multiple PT lines are seen, latter PT lines override previous lines.
USE	Same as PT.
SEE	Same as PT.

BT	Broader Term. A BT B means that B is a broader term of A. A BT B implies B NT A (A is a narrower term of B). Terms can have multiple broader terms. Circular references should not be introduced.
	
	You can specify whole hierarchies under a single term using BTn:
	
	`dog`
	`  BT1 canine`
	`    BT2 mammal`
	`      BT3 vertebrate`
	`        BT4 animal`
	This isn't declaring multiple broader terms of dog, but instead the whole hierarchy above dog -- canine is the first-level BT of dog, then mammal is the second-level BT -- meaning that mammal is the BT of canine. BTn goes up to BT16.

BTP	Broader Term Partative. A BTP B means that B is a broader term of A in the sense that A is a part of B. Hard drive BTP Computer, for instance -- Computer is a broader term of hard drive in the sense that a hard drive is a part of a computer (NCA notwithstanding). A BTP B implies B NTP A, but does not imply A BT B -- the partative hierarchy is separate from the normal hierarchy. This relationship is sometimes called "meronymity", although not by anyone with a life.
BTG	Broader Term Generic. A BTG B means that B is a broader term of A in the sense that B is a generic name for A. Rats BTG Rodents, for instance. A BTG B implies B NTG A, but does not imply A BT B. This relationship is sometimes called "holonymity".
BTI	Broader Term Instance. A BTI B means that B is a broader term of A in the sense that A is an instance of B. Cinderella BTI Fairy Tale, for instance. A BTI B implies B NTI A, but does not imply A BT B. I don't know any fancy name for this relationship.
NT	Narrower Term. The opposite of BT. You can use NTn, just like BTn.
NTP	Narrower Term Partative. The opposite of BTP.
NTG	Narrower Term Generic. The opposite of BTG.
NTI	Narrower Term Instance. The opposite of BTI.
RT	Related Term. A RT B means that A and B are associated, but are not synonyms and cannot be arranged into a hierarchy. A RT B implies B RT A. A RT B and B RT C does not imply A RT C.

<lang>:	Foreign language translation. <lang> can be any label you wish to use for the language, 10 characters or less. This applies to the immediately preceding term, even if it is a relationship word:
	`cat`
	`  french: chat`
	`  RT hat`
	`    french: chapeau`

	Here "chapeau" is the french translation for "hat", not "cat".

SN	Scope note. You can attach a little note to a term as a comment. This can be up to 2000 characters. If you need multiple lines, simply repeat the SN keyword -- the text is concatenated:
	`cat`
	`  SN I had a cat once.  He was brown and`
	`  SN grey and well-behaved.`

Our thesaurus functionality also supports homographic disambiguation using parenthesis:

```
mercury (god)
  BT Greek mythology

mercury (element)
  SYN quicksilver
```

as well as compound terms using the + sign:

```
thermometer
  SYN temperature + instrument
```

Importing a Thesaurus File

Once you have the file, you need to import it into the *inter*Media data dictionary. From the OS command line:

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

The login user can be any CTXAPP user -- it does not have to be ctxsys. -thes is the toggle for thesaurus import. mythes is the name for the new thesaurus, and mythes.txt is the file to import.

If a thesaurus of this name already exists, you will get an error and you must drop the existing thesaurus before importing this one. Thesaurus names are database-global.

Optionally, you can specify the thescase argument to make the thesaurus terms case-sensitive:

```
ctxload ... -thescase Y
```

The default is N, meaning that the thesaurus is not case-sensitive. Mercury (the god) and mercury (the element) are the same term if found in the thesaurus file.

Thesaurus Query Operators

Once the thesaurus is loaded, any user can use it in any query on any index using the thesaurus operators. These mimic the relationship names for the most part. For instance:

```
contains(text, 'SYN(dog, mythes)') > 0
```

will search for documents which contain any of the synonyms of "dog" defined in the "mythes" thesaurus. The actual expansion uses OR like this:

```
{canine}|{doggie}|{mutt}|{dog}
```

Homographic disambiguators are not included in the expansion:

```
mercury (element)
  SYN quicksilver

SYN(quicksilver) === {quicksilver}|{mercury}
```

Compound phrases are treated as a conjunctive:

```
thermometer
  SYN temperature + instrument

SYN(thermometer) == {thermometer}|({temperature}&{instrument})
```

Thesaurus operators take simple thesaurus terms as input -- they cannot nest or take expansions as input:

```
    SYN($dog)        <-- BAD
    SYN(BT(dog))      <-- BAD
```

Here are the specific thesaurus functions:

SYN(term [, thesname])	The input term is always part of the output; if no synonyms are found then term is the expansion. thesname is the name of the thesaurus to use. If omitted, thesaurus DEFAULT is consulted. There is no DEFAULT thesaurus installed automatically -- it is up to the user to load a thesaurus named DEFAULT.
PT(term [, thesname])	Preferred term. Only the preferred term is returned in the expansion.

BT(term [, level [,thesname]])
BTP(term [, level [,thesname]])
BTG(term [, level [,thesname]])
BTI(term [, level [,thesname]])
NT(term [, level [,thesname]])
NTP(term [, level [,thesname]])
NTG(term [, level [,thesname]])
NTI(term [, level [,thesname]])
	Broader/Narrower terms of all types. The optional second argument is a numeric scope indicating how many levels of hierarchy you want. A BT with a level of 2, for instance, would search for the term, all broader terms of the term, and all broader terms of those terms.

TT(term [, thesname])	Top term. Travel up the BT hierarchy to the root term. Only this root term is searched.
RT(term [,thesname])	Related term. Search for term and all related terms.
TR(term [, lang [, thesname]])	Foreign language translation of the term. Lang argument should match label used in the input thesaurus. If omitted all foreign language translations will be searched. Note that the foreign language translation must have been entered in the input thesaurus -- we don't do automatic translation.
TRSYN(term [, lang [, thesname]])	Foreign language translation of term synonyms. This is functionally equivalent to taking SYN of the term, then TR of each resulting term. Like TR, if the target language is omitted, all available translations will be searched.

Browsing a Loaded Thesaurus

There are numerous functions in the ctx_thes package for browsing the thesaurus through PL/SQL. For the most part, they have the same names and arguments as the query functions, but they return the expansion text. For instance:

```
    declare
      exp varchar2(4000);
    begin
```

```
     exp := ctx_thes.syn('dog','mythes');
   end;
```
Here the synonyms of "dog" in the "mythes" thesaurus are returned. The expansion is returned as a string, so exp will get the value:
```
   {canine}|{doggie}|{mutt}|{dog}
```
The return can be up to 32000 characters in length.

Exporting a Thesaurus

Thesaurus export is also done through ctxload:
```
   ctxload -thesdump -user ctxsys/ctxsys -name mythes -file mythes.txt
```
The exported version will be structurally similar to the imported version -- all the words will be there in the right places, but it may not be exactly the same -- in particular, it will probably be more verbose, specifying explicitly relationships which were implied in the input thesaurus.

Thesaurus Standards

The phrase ISO-2788 or -5964 thesaurus standard is somewhat misleading. The computing industry considers a "standard" to be a specification of behavior or interface. These standards do not specify anything. If you are looking for a thesaurus function interface, or a standard thesaurus file format, you won't find it here. Instead, these are guidelines for thesaurus compilers -- compiler being an actual human, not a program. They give general advice on how to build and maintain a thesaurus, such as "Adverbs should not be used alone as indexing terms".
What Oracle has done is taken the ideas in these guidelines and in ANSI Z39.19 -- an American version of ISO-2788 -- and used them as the basis for a specification of our own creation.Therefore *inter*Media can handle all the inter-word relationships mentioned in Z39.19, and even use the same abbreviations, but the interface is our own. the *inter*Media file format looks a bit like "Figure 1" of ISO-2788, but Oracle imposed its own rules to make it parseable. *inter*Media can implement foreign language lookups like ISO-5964, but the input thesaurus looks nothing like their examples. So, Oracle supports ISO-2788 relationships or ISO-2788 compliant thesauri.

Query Tuning

Analyzing Text Queries

In Oracle8*i*, tuning Text queries is not very different from tuning regular SQL queries. This is demonstrated with an extended example, using this table of magazine articles:
```
   create table articles (
     id        number       primary key,    -- article id
     issue      number,                    -- issue number
     title      varchar2(80),               -- article title
     published  date,                      -- publication date
     text       clob                       -- article text
   );
```
Assume issue has a non-unique b-tree index called art_issx, and text has a Text index called art_textx.

The first step in query tuning is analysis of the query plan. This is done using explain plan. Note that the SQL explain plan command is different from ctx_query.explain. explain plan works on a SQL query, and prints the SQL query plan executed by oracle. ctx_query.explain works on the input to the contains function, and prints out the Text query plan executed by *inter*Media. ctx_query. The Oracle Server Tuning manual details how to set up and run SQL explain plan, so check there if you do not know how this is done. You don't need to do anything special for *inter*Media queries -- thanks to tighter integration with the kernel, the optimizer now understands what the contains clause means and can generate plans for Text queries -- it's all transparent to the end user. Here is a simple query:

```
select title from articles                    -- query 1
 where contains(text, 'iced lolly') > 0
```

This will normally generate a query plan like this:

```
1.1 TABLE ACCESS BY INDEX ROWID ARTICLES
  2.1 DOMAIN INDEX  ART_TEXTX
```

The DOMAIN INDEX part is our equivalent of an INDEX RANGE SCAN -- it means that the index (the Text index, in this case) is producing a stream of rowids for the kernel. This is called rowsource invocation, and, like an index range scan, is a fairly efficient way of finding matching rows.

Score

add score to the select list:

```
select score(1) score, title from articles          -- query 2
 where contains(text, 'iced lolly', 1) > 0
```

This generates the same query plan as above -- score does not appear in the query plan. However, score does make query 2 execute a bit more slowly than query 1. The SCORE function is called "ancillary data", and it is delivered separately from the rowid stream. The kernel will ask for some rowids, then, in a separate call, will give a rowid and ask for that row's score. This second call imposes a bit of overhead, which is why query 2 is slower than query 1.

But how about a query like this:

```
select score(1) score, title from articles          -- query 3
 where contains(text, 'iced lolly', 1) > 0
 order by published desc

1.1 SORT ORDER BY
  2.1 TABLE ACCESS BY INDEX ROWID ARTICLES
    3.1 DOMAIN INDEX  ART_TEXTX
```

The kernel classifies the SORT ORDER BY as a "blocking operation". The presence of a blocking operation means that Oracle8*i* cannot guarantee that it will ask for scores in the same order as the produced rowid stream. It could produce rowid ABC first, but get asked for its score last. This means that the scores for all rows must be kept in memory, called "batch processing". If the hitlist is large, this could use disk, and be slower than query 2 score resolution. Of course, the order by would normally make query 3 slower anyway.

The presence of any blocking operation switches *inter*Media to batch processing, even something like this:

```
select score(1) score, title from articles          -- query 4
 where contains(text, 'iced lolly', 1) > 0
 order by score(1) desc
```

could use incremental processing, but still uses batch processing.

Now, query 4 is important in information retrieval, because it is very common to show only the first few top-scoring hits in the hitlist. Web search engines work this way, for instance. This case puts a premium on response time -- the time to get the first page is critical, the time to get the second page is not as important. Unfortunately, in the database world, the order by decreases response time, because all the hits have to be produced and sorted before the first row is displayed.

Because this is a common situation, *inter*Media has an alternative query method which improves response time for score sorting:

```
select /*+ FIRST_ROWS */ score(1) score, title         -- query 5
  from articles
 where contains(text, 'iced lolly', 1) > 0
 order by score(1) desc
```

The addition of the FIRST_ROWS hint causes the kernel to notify *inter*Media of the sort by score. *inter*Media produces the rowid stream in score-sorted order, and the kernel eliminates the SORT ORDER BY -- similar to a b-tree index range scan ordered by the key prefix:

```
1.1 TABLE ACCESS BY INDEX ROWID ARTICLES
  2.1 DOMAIN INDEX  ART_TEXTX
```

This is called "pushing down" the sort. Also, the elimination of the SORT ORDER BY allows us to use incremental score processing instead of batch score processing.

The payoff of FIRST_ROWS is improved response time; the cost is decreased throughput. *inter*Media cannot produce the hits in score-sorted order naturally. Instead, it runs the Text portion internally, and remembers the top-scoring 500 or so rows. Because the set size is small and because we keep it all in memory rather than disk, Oracle8i can sort it faster than a SORT ORDER BY. However, when it runs out of hits in this batch, it has to re-execute the query to get another batch. Thus, if you're fetching all the results, do the SORT ORDER BY; if you just want the first few hits, use the FIRST_ROWS hint.

The FIRST_ROWS hint will not just affect *inter*Media -- the entire query will be optimized for response time. This is not usually a problem, since you want the whole query optimized for response time or throughput. However, if you want to optimize the overall query one way and *inter*Media another way, you can use the DOMAIN_INDEX hints. To have *inter*Media score-sorting, but optimize the overall query for throughput:

```
select /*+ ALL_ROWS DOMAIN_INDEX_SORT */
```

Conversely, if you want the overall query optimized for response time, but you don't want *inter*Media score-sorting:

```
select /*+ FIRST_ROWS DOMAIN_INDEX_NO_SORT */
```

Mixed Queries

So far, the discussion has looked at rowsource invocation for contains -- Oracle8*i* asks for a stream of rowids which match the contains. However, there is an alternate invocation called "function invocation". In function invocation, Oracle8*i* produces the rowid stream some other way -- a b-tree index, a full table scan, etc. and then asks , for each rowid in that stream, if the row matches the contains:

```
select /*+ FULL(articles) */ title                  -- query 6
  from articles
 where contains(text, 'face flannel') > 0

1.1 TABLE ACCESS FULL ARTICLES
```

Here a full table scan is forced, which makes Oracle8*i* go through the table row by row, calling function contains for each one. Note that the contains function invocation is not shown in the plan, just as a structured condition filter would not be shown.

For the same number of rowids, function invocation is much slower than rowsource invocation. So why have it? Consider a query like this:

```
select title from articles                          -- query 7
 where contains(text, 'face flannel') > 0
   and issue = 7
```

Recall that issue has a non-unique b-tree index. rowsource invocation could be used - get a rowid stream from the Text index, and check each rowid to see if issue = 7:

```
1.1 TABLE ACCESS BY INDEX ROWID ARTICLES              -- plan 7
  2.1 DOMAIN INDEX  ART_TEXTX
```

On the other hand, the b-tree index could be used on issue - find all rows where issue = 7, and call function contains individually for each of those:

```
1.1 TABLE ACCESS BY INDEX ROWID ARTICLES              -- plan 8
  2.1 INDEX RANGE SCAN ART_ISSX NON-UNIQUE
```

The query plan is driving from the index on issue, which implies function invocation of contains. Why do that? Function invocation is slower than rowsource for the same number of rowids.

If the number of articles in issue 7 is very small, and the number of articles with "face flannel" in the text is very large (they were quite the rage in the 80's) then the number of rowids is not the same. It may be faster to drive from the b-tree index and use function contains on a small number of rows than to drive from the Text index and do a TABLE ACCESS for a large number of rows.

Of course, it is usually impossible to have a priori knowledge of the selectivities, which makes it impractical to use hints to specify which index to use. Thanks to integration with the cost-based optimizer, you don't have to hard-code it -- let the CBO decide which method is more efficient.

Using the CBO

In order to choose correctly, the cost-based optimizer needs to know the selectivities of the structured and Text conditions. You provide this information by generating statistics:

```
analyze table articles
 compute statistics
```

This will (eventually) generate stats for the table and all indexes, including the Text index. You can use estimate statistics, as well -- is makes no difference with respect to the Text index.

When *inter*Media is asked to compute statistics, it analyzes term frequencies in the Text index. For the most common terms, we record the number of documents which contain that term. It also calculates the average number of documents per term to use for all other terms.

Given a Text query, and using these stats, it can compute an estimate of the query selectivity. Because it doesn't record exact term frequency for every term, and because a Text query can combine terms using AND, phrase, WITHIN, etc. this selectivity is only a rough estimate, not an exact number.

The CBO, confronted with a query like query 7, can then use the b-tree stats and the Text index stats and -- without hints -- can choose the best index to use to drive the query.

If your b-tree data is not uniformly distributed, then histograms for the data can help:

```
analyze table articles
 compute statistics for column issue size 127
```

See Oracle SQL Reference and Server Tuning manuals for more information. Nothing special is done for histograms -- it simply provides more accurate stats to the CBO. Like any SQL query, you can use hints to override the CBO when it's wrong.

Rowid Bitmap Merges

The CBO can choose a third option for query 7 -- use both indexes, and perform a rowid bitmap merge of the two results. You can force this using the INDEX_COMBINE hint:

```
select /*+ INDEX_COMBINE(articles art_issx art_textx) */ -- query 9
       title
  from articles
 where contains(text, 'face flannel') > 0
   and issue = 7
```

which produces the following plan:

```
1.1 TABLE ACCESS BY INDEX ROWID ARTICLES              -- plan 9
  2.1 BITMAP CONVERSION TO ROWIDS
    3.1 BITMAP AND
      4.1 BITMAP CONVERSION FROM ROWIDS
        5.1 SORT ORDER BY
          6.1 DOMAIN INDEX  ART_TEXTX
      4.2 BITMAP CONVERSION FROM ROWIDS
        5.1 INDEX RANGE SCAN ART_TEXTX NON-UNIQUE
```

This is the best performer when the result lists for the structured and contains are roughly equal, somewhat large, and the intersection is small. In comparison, plan 8 would be hobbled by function contains overhead, and plan 7 would require unnecessary TABLE ACCESSes to check the structured condition. A similar plan is used when bitmap indexes are employed.

Bitmap merge, like SORT ORDER BY, is a blocking operation, which, if you are selecting SCORE, will force batch processing instead of incremental processing.

Query Feedback

Text Query Tweaking

Text queries rarely give you exactly what you want. Consider web search engines like AltaVista. Sometimes there's too many documents. Sometimes you wonder why a certain document turned up in the hitlist. Sometimes the document you wanted to find isn't in the results at all.

Similar results could ocur with *inter*Media, so it provides two query feedback methods which allow you to refine your Text query.

Explain

Explain is the *inter*Media analogue of explain plan, and let's you see exactly what your query looks like. This lets you trim unintended wildcard expansions, for instance, or add parentheses to correct precedence errors.

You need a result table, just like explain plan, with the following columns, in the following order:

```
create table xres (
  explain_id      varchar2(30),
  id              number,
  parent_id       number,
  operation       varchar2(30),
  options         varchar2(30),
  object_name     varchar2(64),
  position        number
)
```

explain_id	A label which distinguishes one query plan from another.
id	A step sequence number. This is an ascending sequence number assigned to each step which ignores hierarchy. The root step of each plan has id 1, the second step has id 2, etc.
parent_id	The id of the parent of this query plan step. The root step has parent_id = 0.
operation	The type of operation for this step. This can include WORD, AND, WITHIN, etc.
options	The expansion operators (fuzzy, wildcard, etc.) are transformed into equivalence operators. The options column tells you what the original expansion function was. options is also used to show arguments to NEAR.
object_name	The operand of the query step. For WORD operations, this is the actual word being searched. It also holds section names for SECTION operations, weight values, etc.
position	A sequence number to sequence children of the same parent. The first child step will have position 1, the second will have position 2, etc.

You then run explain for your query on your index:

```
ctx_query.explain('myindex', 'dog and ($run spot)', 'xres');
```

The first argument is the name of the index. The second argument is the query string, and the third is the name of the result table. The query is explained and the results inserted into the plan table. These results are not committed, which mimics the behavior of SQL explain plan. You can then print it out in explain plan style using this SQL:

```
  select lpad(' ',2*(level-1))||level||'.'||position||' '||
         operation||' '||
         decode(options, null, null, options || ' ') ||
         object_name plan
   from xres
  start with id = 1
connect by prior id = parent_id;
```

Which produces the following plan:

```
1.1 AND
  2.1 WORD DOG
  2.2 PHRASE
    3.1 EQUIVALENCE ($) RUN
      4.1 WORD RAN
      4.2 WORD RUN
      4.3 WORD RUNNING
      4.4 WORD RUNS
    3.2 WORD SPOT
```

The optional fourth and fifth arguments to explain, sharelevel and explain_id, allow multiple explains to use the same result table. When sharelevel is 0, the default, then the result table is automatically truncated before the explain is run. When sharelevel is 1, then any rows in the result table with the same explain_id are deleted before the explain is run, but other rows are not touched. Thus, multiple explains can use the same result table as long as they use distinct explain_id's.

When using shared result tables, the start with and connect by clauses of the example sql have to be modified to limit them to a specific explain_id:

```
  start with id = 1 and explain_id = 'YOURID'
connect by prior id = parent_id and explain_id = 'YOURID'
```

Hierarchical Query Feedback

Explain lets you see exactly how your query is run, which allows you to adjust your query to refine the results. Hierarchical query feedback goes one step further and actually suggests other search terms you might want to try.

You start by creating a result table. This result table looks a bit like the explain result table, but with the addition of nested tables for suggested alternative terms:

```
create table fres (
  feedback_id     varchar2(30),
  id              number,
  parent_id       number,
  operation       varchar2(30),
  options         varchar2(30),
  object_name     varchar2(64),
  position        number,
  bt_feedback     ctxsys.ctx_feedback_type,
  rt_feedback     ctxsys.ctx_feedback_type,
  nt_feedback     ctxsys.ctx_feedback_type
)
nested table bt_feedback store as fres_bt
nested table rt_feedback store as fres_rt
nested table nt_feedback store as fres_nt
```

The bt_feedback, rt_feedback, and nt_feedback store suggested broader terms, related terms, and narrower terms, respectively. Each is a nested table of ctx_feedback_type objects, which stores the suggested term in the attribute "text". The other attributes are not currently used, but are reserved for future use.

Now you run hfeedback on your query:

```
ctx_query.hfeedback('myindex', 'felines', 'fres');
```

hfeedback will then draw on the linguistics knowledge base to suggest broader, narrower, and related terms. The results are inserted into the result table. Like explain and SQL explain plan, these inserts are not committed implicitly. You can select from the nested tables to get feedback:

```
select text from the(select nt_feedback from fres
                     where object_name = 'felines')
TEXT
----

cat leopards tiger
```

Be aware that when getting feedback for queries with ABOUT clauses, the input is broken into themes, so you cannot simply select back. For instance, if your query were "ABOUT(African safari)", there would be no row with object_name = 'African safari' -- more likely, there would be separate rows for "Africa" and "safaris".

The feedback terms are obtained from the built-in linguistics knowledge base. If the knowledge base has been extended, feedback can also draw on these extensions.

The results will depend on the contents of your index -- only worthwhile terms are shown. "Lion" is a narrower term of feline, but my index does not have any documents with "Lion" in them, so it does not appear in the narrower term table. Note that for performance reasons, we do not screen out invalid documents, so it is possible (but unlikely) to get feedback terms which would not produce query hits.

The feedback for non-ABOUT portions of the query can contain alternate forms of words, as well -- narrower terms for "learning" might include "education", "educating", "educational", etc.

If the index is case-sensitive, then case-sensitive feedback terms are provided. For instance, narrower terms for "databases" might include both "PL/SQL" and "Pl/sql".

When the index is case-insensitive, queries are case-insensitive -- a contains query for "oracle" will pull up the same records as "Oracle". However, feedback uses the linguistic knowledge base, which is case-sensitive. So, even if your index is case-insensitive, the feedback you get depends on the case of the query -- the feedback for "oracle" will not be the same as the feedback for "Oracle".

You can also give hfeedback multiple terms in a query string:

```
   ctx_query.hfeedback('myindex', 'felines and canines', 'fres');
```

hfeedback will generate query plan steps as in explain, but phrases are not broken into individual words and expansion operators such as fuzzy, wildcard, soundex and stem are ignored. Each word or phrase step will have bt, nt, and rt tables. Operator steps like AND will not have anything in these tables.

Like explain, hfeedback has sharelevel and feedback_id arguments for result table sharing.

Highlighting

Document Services

Highlighting and linguistic extraction have been combined into a unified document services package called ctx_doc. A document service is any ad hoc operation on the document text.

Filter

The filter command fetches the document contents and filters it to readable text. This allows you to see what the indexing engines sees. You first make a result table:

```
  create table fres ( query_id number, document clob )
```

Now request a filter:

```
  ctx_doc.filter('myindex', '123', 'fres');
```

The first argument is the name of the index -- you can use USER.INDEX, but the issuing user must have SELECT access to the base table. The second argument is the primary key value of the document you wish to filter. If you have a composite primary key, use commas to separate the key fields:

```
  ctx_doc.filter('myindex', 'ABC,123', 'fres');
```

Alternatively, you can use the function ctx_doc.pkencode:

```
  ctx_doc.filter('myindex', ctx_doc.pkencode('ABC','123'), 'fres');
```

The third argument to filter is the name of the result table. *inter*Media will translate the primary key to rowid, fetch the document contents by rowid, and filter to readable text. It will insert a new row into the result table and the readable text output will be placed in the document clob column.

Query_ID

There is an optional fourth argument -- query_id -- which can be any integer. This will be the value of the query_id column for the row in the result table. If you are sharing result tables across several filter requests, you can have each request use a different query_id, and distinguish the outputs by this column. filter does not pre-truncate or pre-delete rows in the result table, regardless of the setting of query_id. The default value of query_id is 0.

Plaintext

Recall part of the indexing pipeline for a moment:

```
     +-----------+          +----------+ filtered   +-----------+ plain
     |           | doc data  |          | doc text   |           | text
     | datastore |---------->|  filter  |----------->| sectioner |----->
     |           |           |          |            |           |
     +-----------+          +----------+            +-----------+
```

There are two "readable text" outputs in this segment of the indexing chain -- the filtered text, output from the filter, and the plain text, output from the sectioner. When using the INSO filter, for instance, the filtered text is HTML and might have text like this:

```
      <BODY>
        <P>This is the first
        <B>sentence.</B>
      </BODY>
```

The HTML section group will transform this into plain text, removing tags and reformatting whitespace, much like a browser might save to text file:

```
    This is the first sentence.
```

By default, filter will provide the filtered text. An optional boolean parameter "plaintext" (the fifth argument) allows you to request the output from the sectioner instead:

```
    ctx_doc.filter('myindex', '123', 'fres', plaintext=>TRUE);
```

The plaintext output can be used for human readers, while the filtered text output can be used if you are sending the results to a browser. Note that if you are using the null section group, the plain text and filter text are the same, so the plaintext argument would have no effect.

The default value of plaintext is FALSE, meaning filter text is produced.

Highlight

The highlight service takes a query string, fetches the document contents, and shows you which words in the document cause it to match the query. The result table for highlight looks like this:

```
    create table hres (
      query_id   number,
      offset     number,
      length     number
    )
```

Now request the highlight:

```
    ctx_doc.highlight('myindex', '123', 'dog | cat', 'hres');
```

Like filter, the first argument is the name of the index and the second is the document primary key. The third argument is the text query, and the fourth is the name of the result table.

*inter*Media will parse the query and determine which words in the document match the query. It then creates rows in the result table, showing the character offset and length of each matching word. Let's say that our document looks like this for the highlight call above:

```
    I have a black cat and a white dog.
```

"I" is character 1, per oracle convention, so "cat" is at character 16 and has length 3. "dog" is at character 32 and has length 3. If you are using a multi-byte character set, remember that these are character offsets and lengths, not byte offsets and lengths. The hres table will look like this:

```
      QUERY_ID     OFFSET     LENGTH
    ---------- ---------- ----------
             0         16          3
             0         32          3
```

A single row in the result table can cover multiple words in the source document -- the example simply uses single words.

Highlight has query_id as the fourth argument, working just like it does in the filter service. Offset and length in the result table can apply to either the plain

text or filter text version of the document. This is controlled by the plaintext argument in the fifth position.
You can use the filter service output in conjunction with the highlight output to display the matching words:

```
  select dbms_lob.substr(fres.document, hres.length, hres.offset)
    from fres, hres
```

If your documents are in a varchar or clob you can do something similar with the base table instead of the filter output.

Markup

Markup takes the highlight service one step further, and produces a text version of the document with the matching words marked up. The markup result table is the same format as the filter result table:

```
  create table mres (
    query_id   number,
    document   clob
  )
```

Now call markup:

```
  ctx_doc.markup('myindex', '123', 'cat | dog', 'mres');
```

And the result is:

```
  I have a black <<<cat>>> and a white <<<dog>>>.
```

Like the other highlight services, markup has a query_id argument for result table sharing, and a plaintext argument to specify markup of filter text or plain text.
You can also override the text used to markup the matching words. This is broken into start and end tags. Start tag is the text which goes before the matching word, end tag is the text which goes after the matching word. By default, start tag is <<< and end tag is >>>. You can override this using the starttag and endtag arguments:

```
  ctx_doc.markup('myindex', '123', 'cat | dog', 'mres',
                 starttag => '-->', endtag => '<--');

  I have a black -->cat<-- and a white -->dog<--.
```

The length limit on tags is 30 characters.

Highlight Navigation

Highlight navigation is a new feature for Oracle8*i*, and allows you to markup your text so that an interpreting program can navigate between highlights. Let's use as an example HTML output being sent to a browser. The start and end tags could be overridden to display highlighted words in bold:

```
  ctx_doc.markup('myindex', '123', 'cat | dog', 'mres',
                 starttag => '<b>', endtag => '</b>');

  <b>cat</b> ... <b>dog</b>
```

Now you might want to have links next to each highlighted word which take you to the previous or next highlight. You would first have to anchor and name each highlight using the A tag:

```
  <A HREF=h1><b>cat</b></a> ... <A HREF=h2><b>dog</b></a>
```

Note that the start tags for these two words are different -- cat has h1 and dog has h2. You can do this using markup macros. Each highlighted word is numbered in document occurrence order. If %CURNUM is in a tag, it is replaced with this number. This lets you do the anchoring above using start and end tag definitions like so:

```
  ctx_doc.markup('myindex', '123', 'cat | dog', 'mres',
                 starttag => '<a name=h%CURNUM><b>',
                endtag => '</b></a>');
```

But clickable links for highlight navigation have not been achieved. A links would need to added before and after each highlight, like this:

```
<a href=h1><b>cat</b></a><a href=#h2>></a> ...
<a href=#h1><</a><a href=h2><b>dog</b></a>
```

The first problem is that start and end tag cannot be used for this. For highlight navigation, the first highlight will not have a prev link, and the last highlight will not have a next link. Start and end tags are added for every highlight, so those can't be used.

Luckily, markup has prevtag and nexttag specifically for this. Prevtag is added before starttag for every highlight except the first one. Nexttag is added after endtag for every highlight except the last one.

The second problem is that it is necessary to reference the next highlight number in nexttag, and the previous highlight number in prevtag. For this, two other markup macros, %PREVNUM and %NEXTNUM are used

Putting it all together, highlight navigation is obtained like this:

```
ctx_doc.markup('myindex', '123', 'cat | dog', 'mres',
            starttag => '<a name=h%CURNUM><b>',
            endtag => '</b></a>',
            prevtag => '<a href=#h%PREVNUM><</a>',
            nexttag => '<a href=#h%NEXTNUM>></a>');
```

This gives highlighted words in bold, with < and > links next to each one to take the reader to the previous or next highlighted word.

Tagset

The call is complex so three sets of common definitions for starttag, endtag, prevtag, and nexttag have been defined. You can use these sets by using the tagset argument of markup:

```
ctx_doc.markup('myindex', '123', 'cat | dog', 'mres',
           tagset => 'HTML_NAVIGATE')
```

This will give you similar functionality to the call above. Here are the defined tagsets:

```
TAGSET
-------------
TEXT_DEFAULT
  starttag = <<<
  endtag = >>>

HTML_DEFAULT
  starttag = <b>
  endtag = </b>

HTML_NAVIGATE
  starttag = <A NAME=ctx%CURNUM><b>
  endtag = </B></A>
  prevtag = <A HREF=#ctx%PREVNUM><</A>
  nexttag = <A HREF=#ctx%NEXTNUM>></A>
```

You cannot define your own tagsets in this version. The tagset default is TEXT_DEFAULT. Any specified values for starttag, endtag, prevtag, and nexttag override the values for the tagset, so something like this:

```
ctx_doc.markup('myindex', '123', 'cat | dog', 'mres',
           tagset => 'TEXT_DEFAULT',
           endtag => '<--')
```

would highlight like this:

```
<<<dog<--
```

That is, the endtag has been overridden, but the starttag definition from the tagset is still valid.

Linguistic Extraction

Linguistic Extraction

With the INDEX_THEMES attribute of the basic lexer, and the ABOUT clause, you can easily enable thematic indexing and searching for your document table. The linguistic extraction routines in the document services enable you to generate document themes or theme summaries, on-demand and per-document.
The big change from previous versions is the elimination of the serivces queue. In ConText, linguistics was asynchronous -- you submitted a linguistic request, then eventually the answer would appear in the result table. Vast improvements in linguistic engine performance have allowed us to make these requests synchronous in Oracle8*i*. Now you invoke the linguistic function, and when the function returns, you know the data is in the result table.
Linguistic extraction works only on English documents.

Themes

Themes allow you to generate the themes of a document. First you need a result table like this:

```
create table tres (
  query_id   number,
  theme      varchar2(80),
  weight     number
)
```

Then invoke the themes request:

```
ctx_doc.themes('myindex', '123', 'tres');
```

Like the highlighting calls, the first argument is the name of the index, the second is the document primary key, and the third is the name of the result table.
*inter*Media will fetch the document text, filter it, run it through the sectioner, then parse it for thematic content. It will generate up to 50 themes, and place the results in the result table, in the theme column, one row per theme. The weight column is a number showing relative importance of the theme. It is only comparable to other themes of the same document. You cannot compare theme weights from different documents.
An optional fourth query_id argument lets you share result tables. Like highlighting, themes will not pre-truncate or pre-delete the result table.
The optional fifth argument, full_themes, allows you to generate themes with their full knowledge catalog hierarchy. The call above might generate a theme of "cats", for instance. Setting full_themes on:

```
ctx_doc.themes('myindex', '123', 'tres', full_themes => TRUE);
```

would generate

```
:science and technology:hard sciences:life sciences:biology:zoology:
vertebrates:mammals:carnivores:felines:cats:
```

instead. If you are using full themes, the length of the theme column in your result table should be increased to 2000 or so.

Gist

Gist allows you to generate a theme summary of the document text. This does not generate an original abstract like a human would. Instead, it is composed of selected paragraphs or sentences which contribute most to the themes of the document. The gist result table looks like this:

```
create table gres (
  query_id  number,
  pov       varchar2(80),
  gist      clob
);
```

and the gist function looks like this:

```
ctx_doc.gist('myindex', '123', 'gres');
```

*inter*Media will fetch the document text, filter it, run it through the sectioner, then parse it for thematic content. It will generate the themes, then generate gists for those themes, one row per gist, with the gist column consisting of those paragraphs which contributed to the theme. The theme is contained in the pov (point-of-view) column. Additionally, it might generate a GENERIC gist, which consists of those paragraphs which best represent the overall themes of the document.
The optional fourth argument to gist is query_id, which allows you to share result tables, like all other calls in document services.
The optional fifth argument, glevel, is a flag which allows you to specify the gist level. If glevel is 'P', the default, the gists are composed of whole paragraphs from the document text. This is called the paragraph-level gist. If glevel is set to 'S', the gists are composed of individual sentences from the document text. This is called the sentence-level gist.
The optional sixth argument, pov, allows you constrain output to just one point-of-view. Usually, this is done to generate only the GENERIC gist. This is case-sensitive, so specify GENERIC in uppercase. This is not a way to force the gist to a particular point-of-view; you cannot specify pov's here which would not normally get produced. For instance, if you have a document in which "dogs" and "cats" are pov's, and you specify "frogs", you will get no output.
The optional seventh and eighth arguments, numparagraphs and maxpercent, allow you to constrain the size of the generated gist. numparagraphs specifies the maximum number of paragraphs (or sentences, if you are doing a sentence-level gist) which can make up each generated gist. maxpercent is similar, but limits it to a percentage of the source document. If both are specified, the smaller is the effective limit.

The Extensible Knowledge Base

All theme based features in *inter*Media -- themes, ABOUT queries, gists, ABOUT query highlighting, and hierarchical query feedback -- depend on an internal knowledge base (KB) that is supplied with *inter*Media. The knowledge base consists of a lexicon of English words and phrases organized in a hierarchy of categories. For the first time in Oracle8*i*, you can extend and customize this knowledge base by adding new terms or redefining existing ones.
Consider a simple example in the domain of coffee futures. Documents in this domain may mention the term ICO (International Coffee Organization). interMedia can identify the theme "ICO" in such documents, but it does not exist in the supplied KB -- it is known that the documents are talking about the ICO, but not what ICO is. ICO is not classified in the concept hierarchy, and is

not associated with terms such as "coffee" or "International Coffee Organization". ICO would look like this in a full-themes listing:

```
   :ICO:
```

if it showed up at all. Because it is an isolated term with no parents, it won't be judged a strong theme.

Loading a Custom Thesaurus

What is needed is to do is tell *inter*Media what ICO is all about. The first step is to create a simple thesaurus for the term:

```
international trading organizations
     BT international trade and finance
International Coffee Organization
     BT international trading organizations
     RT coffee
ICO
     USE International Coffee Organization
```

Now load the thesaurus using ctxload, just like any other thesaurus. Make sure the thesaurus is loaded case-sensitive, since case matters for theme generation:

```
 ctxload -user ctxsys/ctxsys -thes -thescase Y -name COFFEE -file
```

and run the KB compiler, ctxkbtc (a new executable in 8i):

```
 ctxkbtc -user ctxsys/ctxsys -name COFFEE
```

This will generate a new, extended KB. ctxkbtc would interpret the thesaurus above as the following commands:

```
  add "international trading organizations" to the KB under
      "international trade and finance"
  add "International Coffee Organization" under
      "international trading organizations"
  make "coffee" a related term to "International Coffee Organization"
  add "ICO" as an alternate form for "International Coffee Organization"
```

Note that "international trade and finance" is an existing category in the supplied KB. All supplied categories are listed in the documentation. It is recommended that new terms be linked to one of these categories for best results in theme extraction. If new terms are kept completely disjoint from existing categories, they might not be strong enough to qualify as themes.

There can be only one user extension per installation -- each run of ctxkbtc affects all users. However, you can specify multiple thesauri (up to 16) in the -name argument of ctxkbtc to add them all to the extended KB:

```
 ctxkbtc -user ctxsys/ctxsys -name COFFEE TEA MILK
```

Because ctxkbtc affects the entire installation, we recommend that only *inter*Media administrators should extend the knowledge base.

Using the Extended KB

Nothing special needs to be done to use the new KB, other than to close and re-establish open sessions -- iMT will use the new KB automatically. If you generate themes now, this time you will see:

```
    :business and economics:commerce and trade:international trade and
    finance:international trading organizations:International Coffee
    Organization:
```

Note that ICO has been normalized to "International Coffee Organization", it now has parents, and its weight is higher because it is a known term.

There are some size limits and other restrictions on thesaurus terms for extending the KB. See the Oracle8*i* *inter*Media documentation for details.