An Oracle White Paper
June 2012

New Features in Oracle Text with Oracle Database12c

Table of Contents

Introduction

Oracle Text is the text searching and analysis framework in the Oracle Database. In Oracle Database 12c, Oracle Text continues to add features which maintain its position as the leading RDBMS-based text handing system.

This paper introduces the new features added in this release. It is intended for an audience already familiar with the capabilities of Oracle Text in Oracle Database 11g.

Near Real Time Indexes

In previous versions of Oracle Text, there was always a trade-off between latency and fragmentation of indexes.

Users generally want their indexes to be updated as fast as possible, by having the index synchronization done frequently, or even immediately "on commit". However, since the size of $I "chunks" written to the indexes depend on the number of records synced at a time, this leads to excessive fragmentation of the $I table and the need to run index optimization more often, and for longer.

In 12c we introduce the concept of two-level indexes under the feature name "near real time indexes". This allows you to have a small, fragmented index containing all the recent updates to the table, without having to modify the large "standard" index. The aim is that the small index should be small enough to fit comfortably into SGA memory, so the fragmentation is not really an issue (since memory "seeks" are virtually free).

The feature is turned on with a storage preference STAGE_ITAB set to true:

```
create table my_table( id number primary key, text varchar2(2000) );

exec ctx_ddl.drop_preference  ( 'my_storage' )
exec ctx_ddl.create_preference( 'my_storage', 'BASIC_STORAGE' )
exec ctx_ddl.set_attribute    ( 'my_storage', 'STAGE_ITAB', 'true' )

create index my_index on my_table( text ) indextype is ctxsys.context
parameters( 'storage my_storage sync (on commit)' );
```

After doing this, we find we have a new "dollar" table - DRMY_INDEXG which has exactly the same layout as the $I table. Similarly, there will be an index on this table DRMY_INDEXH, which matches the normal $X index.

If we create the index empty (as above), then add a row to the table commit it (note that we used "sync (on commit)"), we will see that there are rows in the $G table but not the $I table. In fact, let's add two rows, committing/syncing after each:

```
insert into my_table values( 1, 'hello world' );
commit;

insert into my_table values( 2, 'goodbye world' );
commit;

select count(*) from dr$my_index$i;
```

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

select count(*) from dr$my_index$g;

  COUNT(*)
----------
         4
```

We can see that there are no entries in the $I table, and four entries in the $G table. That's two for "world" (since the rows were synced seperately) and one each for "hello" and "goodby". We can optimize this using a new option - MERGE. This will take rows from the $G table and move them to the $I table, optimizing as it does so.

```
exec ctx_ddl.optimize_index('my_index', 'MERGE')

select count(*) from dr$my_index$i;

  COUNT(*)
----------
         3

select count(*) from dr$my_index$g;

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

Now we can see that there are three entries in the $I - the two "world" records have been combined - and no entries in $G.

Note that the entries in $I are optimal in terms of fragmentation, they can still contain garbage. If a record is deleted, the $I table may contain pointers to records which no longer exist, or even words that no longer exist in the index at all. In order to remove garbage, it is still necessary to run optimize in FULL mode occasionally.

Now we mentioned earlier that we wanted to keep the $G table in memory. How do we ensure that happens? We could of course just rely on normal SGA caching - if there are regular updates to the index then we can really expect that it will remain in memory. Alternatively, if you have a "keep" pool, which is sized to ensure nothing gets flushed to disk, you can make use of this using appropriate STORAGE attributes, such as:

```
exec ctx_ddl.set_attribute    ( 'my_storage', 'G_TABLE_CLAUSE', 
'storage (buffer_pool keep) )'
exec ctx_ddl.set_attribute    ( 'my_storage', 'G_INDEX_CLAUSE', 
'storage (buffer_pool keep) )'
```

Auto Optimize of Near Real Time Indexes

In the previous discussion, we considered the use of the "MERGE" optimization mode, to move index data from the "in memory" $G table to the "on disk" $I table. At times, it can be beneficial to run this merge manually. But there's an alternative to simplify things - a new ctx_ddl procedure called add_auto_optimize. We can easily set our previous index to be auto-optimized with

```
exec ctx_ddl.add_auto_optimize( 'my_index' )
```

This registers the index in a text data dictionary table, which can be seen through the view ctx_user_auto_optimize_indexes:

```
select aoi_index_name, aoi_partition_name from
ctx_user_auto_optimize_indexes;

AOI_INDEX_NAME                 AOI_PARTITION_NAME
------------------------------ ------------------------------
MY_INDEX
```

Optimize merge, when configured in this way, is actually performed by a dbms_scheduler job called DR$BGOptJob, which we can see if we're logged in as a DBA user:

```
select owner, job_name, program_name from dba_scheduler_jobs where
owner='CTXSYS';

OWNER                JOB_NAME             PROGRAM_NAME
-------------------- -------------------- --------------------
CTXSYS               DR$BGOPTJOB          DR$BGOPTPRG
```

Auto Optimize of Near Real Time Indexes

In the previous discussion, we considered the use of the "MERGE" optimization mode, to move index data from the "in memory" $G table to the "on disk" $I table. At times, it can be beneficial to run this merge manually. But there's an alternative to simplify things - a new ctx_ddl procedure called add_auto_optimize. We can easily set our previous index to be auto-optimized with

```
exec ctx_ddl.add_auto_optimize( 'my_index' )
```

This registers the index in a text data dictionary table, which can be seen through the view ctx_user_auto_optimize_indexes:

```
select aoi_index_name, aoi_partition_name from
ctx_user_auto_optimize_indexes;

AOI_INDEX_NAME                 AOI_PARTITION_NAME
------------------------------ ------------------------------
MY_INDEX
```

Optimize merge, when configured in this way, is actually performed by a dbms_scheduler job called DR$BGOptJob, which we can see if we're logged in as a DBA user:

```
select owner, job_name, program_name from dba_scheduler_jobs where
owner='CTXSYS';

OWNER                JOB_NAME             PROGRAM_NAME
-------------------- -------------------- --------------------
CTXSYS               DR$BGOPTJOB          DR$BGOPTPRG
```

BIG_IO large TOKEN_INFO option

The $I table in Oracle Text is generally the largest index table. It contains all of the indexed words from the source, and for each word it has a set of "postings", consisting of the docid of the row containing the word, and the word positions within each docid. This information (held in the token_info column) is encoded into a binary string. In previous versions of Oracle database, each token_info entry was restricted to somewhat less than 4000 bytes in length, to ensure that it could be held "in line" in the same database block as the token_text to which it refers. This avoids an extra seek which would be necessary if the token_info was in a separate "out of line" LOB.

As postings lists get larger and larger, we need more and more rows in the $I table for each word. We also find that as disk technology improves, the performance of sustained large reads improves much faster than the performance of the seeks necessary to find another row in the table. Therefore in 12c we have provided an option to use much larger token_info entries, making use of Oracle's "secure files" technology.

The option is switched on using the BIG_IO attribute in the BASIC_STORAGE preference. Note that you must be using ASSM (Automatic Segment Space Management) to use this option. ASSM is the default for all tablespaces other than SYSTEM and TEMP, but if your tablespace does not use ASSM, the BIG_IO setting will be ignored, and Text will revert silently to normal BASICFILE lobs.

BIG_IO is switched on as follows:

```
exec ctx_ddl.create_preference( 'my_storage', 'BASIC_STORAGE' )
exec ctx_ddl.set_attribute    ( 'my_storage', 'BIG_IO', 'true' )

create index ... parameters( 'storage my_storage' )
/
```

If you want to check whether the option is on, you can use dbms_metadata.get_ddl to make sure the $I table is using SECUREFILE for its LOB. For example if you have an index called MY_INDEX, you could run this in SQL*Plus:

```
set long 50000
set pagesize 0
select dbms_metadata.get_ddl ( 'TABLE', 'DR$MY_INDEX$I' ) from dual
/
CREATE TABLE "DEMO"."DR$MY_INDEX$I"
   (    "TOKEN_TEXT" VARCHAR2(64) NOT NULL ENABLE,
        "TOKEN_TYPE" NUMBER(10,0) NOT NULL ENABLE,
        "TOKEN_FIRST" NUMBER(10,0) NOT NULL ENABLE,
        "TOKEN_LAST" NUMBER(10,0) NOT NULL ENABLE,
        "TOKEN_COUNT" NUMBER(10,0) NOT NULL ENABLE,
        "TOKEN_INFO" BLOB
   ) SEGMENT CREATION IMMEDIATE
  PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255
 NOCOMPRESS LOGGING
  STORAGE(INITIAL 8388608 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS
2147483645
  PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1
  BUFFER_POOL DEFAULT FLASH_CACHE DEFAULT CELL_FLASH_CACHE DEFAULT)
  TABLESPACE "TS_YS_ASSM"
 LOB ("TOKEN_INFO") STORE AS SECUREFILE (
  TABLESPACE "TS_YS_ASSM" ENABLE STORAGE IN ROW CHUNK 8192
  NOCACHE LOGGING  NOCOMPRESS  KEEP_DUPLICATES
  STORAGE(INITIAL 8388608 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS
2147483645
  PCTINCREASE 0
  BUFFER_POOL DEFAULT FLASH_CACHE DEFAULT CELL_FLASH_CACHE DEFAULT))
  MONITORING
```

And here's a testcase which will confirm that BIG_IO is working. We insert the same word into 5000 rows and index it. With BIG_IO on, we will see just a single big row in the $I table, whereas with BIG_IO off, we would see at least two rows:

```
create table my_table (text varchar2(80));

begin
  for i in 1 .. 5000 loop
    insert into my_table values ('hello');
  end loop;
end;
/

exec ctx_ddl.drop_preference  ( 'my_storage' )
exec ctx_ddl.create_preference( 'my_storage', 'BASIC_STORAGE' )
exec ctx_ddl.set_attribute    ( 'my_storage', 'BIG_IO', 'true' )

create index my_index on my_table (text) indextype is ctxsys.context
parameters ('storage my_storage')
/

column token_text format a15
select token_text, length(token_info) from dr$my_index$i;

TOKEN_TEXT      LENGTH(TOKEN_INFO)
--------------- ------------------
HELLO                        15023
```

Whereas if we had BIG_IO set to false, we would see:

```
TOKEN_TEXT      LENGTH(TOKEN_INFO)
--------------- ------------------
HELLO                          996
HELLO                         3501
HELLO                        3501
HELLO                        3501
HELLO                        3501
```

Session Duration Stored Query Expressions (SQEs)

When creating queries, it is often useful to have a shared component which is used in many or all queries. These components typically restrict the query to one part of the data set, or implement security criteria. Let's say I want to search for documents which are public or may be viewed by the Sales department. If I'm looking for "sales data" my query might be:

```
SELECT text FROM mydoc WHERE CONTAINS (text, '( sales data ) AND ( (
SDATA (public=true) OR (sales WITHIN allowed_depts) )*10*10 )' ) > 0
```

Note: In case you're not familiar with the "*10*10" part above, that forces the score of the right-hand-side of the "AND" to the maximum value, and since "AND" scores the *minimum* of the two sides, ensures that this security clause has no effect on the overall score of the query.

Instead of coding this right-hand-side string in each query, we can save it as a "stored query expression" - let's say we call it SQE1 - and then run the simpler query

```
SELECT text FROM  mydoc WHERE CONTAINS (text, '( sales data ) AND
SQE1' ) > 0
```

This simplifies development, and keeps the size of the query down, which is especially important if you are using the VARCHAR2 version of the CONTAINS function (you can use a CLOB instead, which makes this less of an issue, but CLOBs aren't always easy to handle in client interfaces like JDBC).

Oracle Text has supported SQEs for some time. But for a variety of reasons, developers often want to create temporary SQEs which do not persist in the database. Before, that meant assigning generated names to each SQE, and keeping track of them so they can be deleted at the end of the session so as not to clutter up the system. This isn't always easy to do - often you don't know when a session is finishing so you can clean up properly. So in 12c we've introduced session duration queries:

```
ctx_query.store_sqe(
   query_name => 'tempsqe1',
   text_query => '( ( SDATA (public=true) OR ( sales WITHIN
allowed_depts ) )*10*10 )',
   duration   => CTX_QUERY.DURATION_SESSION
   )
```

The other option for duration is CTX_QUERY.DURATION_PERSISTENT.

There are several advantages for session-duration SQEs:

- There is no need to synchronize SQE-name creation across different sessions
- They are automatically cleaned up at the end of the session
- They are more efficient, since they are stored in PL/SQL package variables, and nothing needs to be written to or read from the database.

SQEs themselves are "query text" which must be interpreted and run as part of each query they are used in. If you want to speed up the execution of SQE (or indeed any shared query components) you should take a look at the new Query Filter Cache feature as well.

Note the "namespace" of SQEs is shared between SESSION and PERSISTENT SQEs. So any number of session can use SESSION SQEs called "sqe1" so long as none of them stores a persistent SQE with the same name. As soon as an SQE is create with PERSISTENT duration, that name is unavailable for use as a SESSION SQE.

Snippet Support in Result Set Interface

Oracle Text introduced the Result Set Interface (RSI) in 11g. RSI is an efficient way of running Text queries since it avoids the overheads of using the SQL layer, and is also very useful for queries which produce summary information that can't easily be coded in a standard SQL "SELECT".

Although it has advantages, there are limitations on what can be fetched back through the RSI. In 11g, you were restricted to ROWID, score and SDATA values. Crucially, you could not use "snippets" in your query. That made it impossible to build a "Google-like" query application without doing separate processing for each row to call ctx_doc.snippet.

In 12c, we have added a snippet clause to the Result Set Descriptor. For example:

```
<ctx_result_set_descriptor>
  <hitlist start_hit_num="1" end_hit_num="10" order="SCORE DESC">
    <rowid />
    <score />
    <sdata name="title" />
    <sdata name="author" />
    <snippet radius="20" max_length="160" starttag="&lt;b&gt;"
endtag="&lt;/b&gt;" />
  </hitlist>
  <count />
</ctx_result_set_descriptor>
```

This query will fetch back the rowid, score, two SDATA fields called "title" and "author", and the snippet. The "radius" and "maxlength" arguments are described elsewhere in the snippet enhancements section.

In the Result Set you may see multiple snippet segments, such as:

```
<ctx_result_set>
  <hitlist>
    <hit>
      <rowid>AAAUz8AAAAAADZBAAG</rowid>
      ...
      <snippet>
        <segment>running the <b>Oracle</b> Database</segment>
        <segment>can be seen with <b>Oracle</b> Text</segment>
        <segment>from <b>Oracle</b> Corporation based in
California</segment>
      </snippet>
```

It is up to the application to assemble the various snippet segments into something to display - typically they would be concatenated with an ellipsis ("...") between each.

Separate Offsets

The token_info column of the $I table contains the locations of all indexed tokens (words) within the index. The location information consists of two parts: the DOCID which identifies the particular record (or document) that the word appears in, and the OFFSET of the word within that document. Where documents are large, the OFFSET information is often many times bigger in size than the DOCID information, as most words appear many times in each document.

When do we need the OFFSET information? Only for certain query types:

- Phrase searches - where we need to know if two or more words occur next to each other
- NEAR searches - where we need to know the location of words
- Zone section queries - where we need to do an intersection between the word information and the zone section information

When do we *not* need OFFSET information?

- Single term queries
- Boolean or score-based searches - AND, OR, NOT, MINUS
- Field section searches

If your searches are mainly in the second list, then you are wasting I/O by fetching all the unnecessary OFFSET information. In that case, you should use the new storage option SEPARATE_OFFSETS.

```
exec ctx_ddl.drop_preference  ( 'my_storage' )
exec ctx_ddl.create_preference( 'my_storage', 'BASIC_STORAGE' )
exec ctx_ddl.set_attribute    ( 'my_storage', 'SEPARATE_OFFSETS',
'true' )

create index my_index on my_table( text ) indextype is ctxsys.context
parameters( 'storage my_storage' )
/
```

If we now DESCRIBE the dr$my_index$i table, we will see an additional column TOKEN_OFFSETS which contains the offset information:

```
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------
 TOKEN_TEXT                                NOT NULL VARCHAR2(64)
 TOKEN_TYPE                                NOT NULL NUMBER(10)
 TOKEN_FIRST                               NOT NULL NUMBER(10)
 TOKEN_LAST                                NOT NULL NUMBER(10)
 TOKEN_COUNT                               NOT NULL NUMBER(10)
 TOKEN_INFO                                         BLOB
 TOKEN_OFFSETS                                      BLOB
```

So when should you use the new separate_offsets option? It has to be a judgment call based on the types of queries your users or application mostly run (see the lists above), and the size of your typical documents. If the documents are very large, there will be much bigger savings than when the documents are small.

Document Lexer

Oracle Text has long supported the MULTI_LEXER. That allows you to define different lexers for different languages, such that you might, for example, use BASIC_LEXER for English text but CHINESE_LEXER for Chinese text. The MULTI_LEXER is controlled by the LANGUAGE column of the base table, as specified in the parameters clause when you create an index.

Sometimes customers need to be able to specify different lexer attributes (such as printjoins) according to the *type* of document being processed. This is a similar, but different challenge to the language requirement. To support it, we have introduced the concept of a *language independant sublexer*, which may be added to the MULT_LEXER. This still uses a "language" column to specify which sublexer to use, but is no longer directly tied to an actual language. Additionally, users can

- Dynamically add and remove sublexers on an existing index
- Dynamically update sublexers
- Add and remove stopwords in a MULTI_STOPLIST

Note: An important restriction on the Document Lexer is that it must use the BASIC_SECTION_GROUP. The default when you create an index is the NULL_SECTION_GROUP, and this will not work with language independant sublexers - and neither will AUTO_SECTION_GROUP or PATH_SECTION_GROUP.

Now there's a potential issue when searching documents which have been indexed with multiple lexers. How do we know how to lex the search string? Let's say the term "multi-functional" is indexed by two lexers, one of which has "-" as a PRINTJOINS and one which doesn't. The expression will be in the index as two separate terms MULTI and FUNCTION, but also as a single term MULTI-FUNCTIONAL. If the user searches for MULTI-FUNCTION (or, as he has to escape the special character MULTI\-FUNCTIONAL) then we still need to find both of these. With normal language lexers, we know the session query language and can apply the appropriate options from that. With language-independant sublexers, we don't. So the anwer is simple - we lex the query terms with ALL possible lexers, and use all the resulting terms in the query.

If we use ctx_query.explain we can easily see this:

```
exec ctx_query.explain('my_index', 'multi\-function', 'explain_tab')

select lpad(' ',2*(level-1))||level||'.'||position||' '||
          operation||' '||
          decode(options, null, null, options || ' ') ||
          object_name plan
     from explain_tab
    start with id = 1
    connect by prior id = parent_id
/
PLAN
----------------------------------------------------------------------
---------------------------------------------------
1.1 OR
  2.1 AND
    3.1 WORD MULTI-FUNCTION
    3.2 WITHIN
      4.1 WORD DR$ML$HYP_DOCS
      4.2 SECTION DR$ML
  2.2 AND
    3.1 PHRASE
      4.1 WORD MULTI
      4.2 WORD FUNCTION
    3.2 WITHIN
      4.1 WORD DR$ML$DEFAULT_DOCS
      4.2 SECTION DR$ML
```

We can see both "MULTI-FUNCTION" and "MULTI FUNCTION" in there - but we can also see something else: There's a special section which has been created called DR$ML (ML for MULTI_LEXER) and a pair of words which represent the different sub-lexers. This is the way Oracle Text identifies which documents have been indexed by which lexer, and avoids false matches caused by the document lexer not matching the query lexer. That's why we need to use BASIC_SECTION_GROUP for the index.

SDATA Improvements

SDATA (**S**tructured-**DATA**) sections were introduced in 11g. We have made some improvements in 11c.

SDATA sections can be updated without reindexing the entire document

- You can add new SDATA sections to an existing index
- Queries using templates can order by one or more SDATA sections
- The number of SDATA sections has been increased from 32 to 99

Of these, the last is the easiest to cover. You can now have up to 99 SDATA sections. Note that the combined number of FILTER BY and ORDER BY columns (which are implemented under the covers as SDATA) is still limited to 32.

SDATA sections can be added to an existing index using an "alter index" command such as:

```
alter index my_index parameters( 'add sdata section stocklevel tag
stock datatype number' )
```

This adds a new SDATA section called "stocklevel" with datatype number. It will automatically added on all future documents where the tagset <stock> and </stockset> are found.

SDATA sections can be updated using the new ctx_ddl procedure update_sdata. For example:

```
ctx_ddl.update_sdata(
     idx_name     => 'prod_ind',
     section_name => 'stocklevel2',
     sdata_value  => sys.anydata.convertnumber(stocknum),
     sdata_rowid  => rid
```

Note the use of sys.anydata to convert a numeric vslue (stocknum) to the "anydata" datatype used by the sdata_value argument.

The update_sdata procedure is **synchronous** - it takes effect immediately without you needing to sync the index. It is therefore very useful for fast-changing values such as stock levels. Note however that the row itself must have been synced - if you try to update the sdata for a row which has been committed but not sync'ed, you will get:

```
DRG-50857: oracle error in ctx_ddl.update_sdata
ORA-20000: Oracle Text error:
DRG-11317: rowid AAAW+HAAEAAAUi0AAC is not in the index
```

Finally: ordering by sdata fields. This was previously partly-supported through the user-defined scoring mechanism, but the syntax was cumbersome. You used to have to use a score normalization expression, such as:

```
<score normalization_expr = "sdata(stock)"/>
```

Then rely on score sorting. Now, instead, you can supply a set of "orderkey" tags, using either SCORE or SDATA expressions. For example a full query template might have:

```
<query>
  <textquery>
    digital and sdata(stocklevel > 4)
  </textquery>
  <order>
    <orderkey> SDATA(stocklevel) desc </orderkey>
    <orderkey> score desc </orderkey>
  </order>
</query>
```

You can use any SDATA field here - not just ones that have been specified as ORDER BY columns when creating the text index.

Be aware that any clause in the template only affects the order the rows are returned from the text index. An explicit "ORDER BY" clause in the SELECT query will override this. Also, if the row-by-row "functional invocation" mode of the text index is used, the order section in the query template will be ignored.

Pattern Stopclass

It's common to define a set of "stopwords" - or words that shouldn't be indexed - for a text index. You can also define the stopclass NUMBERS to avoid indexing numbers. In 12c you can also create a set of regular expression patterns to define stopwords as well. Let's say my application indexes data which has a lot of coded values beginning with AZN. If I have 10 million documents, each of which has its own AZN code, then that's 10 million entries in my $I table that I may not want. If that's the case, then I can define a pattern stopclass as follows:

```
exec ctx_ddl.create_stoplist('stop', 'BASIC_STOPLIST')
exec ctx_ddl.add_stopclass  ('stop', 'azn values', 'AZN[[:alnum:]]*')
```

Note the use of the posix character class [:alnum:] meaning any alpha-numeric character. I could also have written the pattern as 'AZN\w*'. It might have been tempting to write:

```
exec ctx_ddl.add_stopclass  ( 'stop', 'azn values', 'AZN.*' )
```

but we have to be careful here. The .* in this case will match whitespace as well, and the text following the AZN code will all be removed as well.

Some other useful patterns

Remove all numeric strings longer than five digits (so keep years such as 2011, but remove 20112):

```
exec ctx_ddl.add_stopclass  ('stop', 'fivedignums',
'[[:digit:]\,\.]{5,}')
```

Remove any words with at least five digits somewhere within an alphanumeric string:

```
exec ctx_ddl.add_stopclass  ('stop', 'wordswith5digitssomewhere',
'([[:alpha:]]*[[:digit:]]){5,}[[:alpha:]]*')
```

Remove all words of any type which are longer than 20 characters

```
exec ctx_ddl.add_stopclass  ('stop', 'longwords', '[[:alnum:]]{20,}')
```

One thing to watch: in order to add a stopclass, you must define your own stoplist. This does NOT inherit the standard list of stopwords from the default system stoplist for your language. If defining your own stoplist you will need to explicitly add all the stopwords for your language. These are defined in an appendix to the Text Reference manual, or you can use ctx_report.create_index_script to dump the definition of a standard index, which will list all the stopwords for you.

Query Filter Cache

As mentioned in the SQE section, there are often situations where parts of queries are used repeatedly. SQEs enable you to save the text of a query for convenience, but they have no effect on performance. Any complex expressions in the SQE must be re-evaluated each time the SQE is run. This typically involves fetching blocks from the SGA (or even database files), decompressing postings lists, and applying boolean merge operations between each list.

The query filter cache, implemented by the ctxfiltercache operator, avoids this reprocessing by storing the final set of DOCIDs for a query expression in SGA memory. This means that when the same query expression is used subsequently, the list of DOCIDs can be used directly from the cache, avoiding the need to fetch from the SGA or do any decompression or merging.

In order to use ctxfiltercache, you must set a non-zero value for the index storage attribute QUERY_FILTER_CACHE_SIZE. This defines how much memory will be made available for ALL ftxfiltercache queries used for this index.

The filter cache data is invalidated whenever the database is restarted, but less obviously it is also invalidated when the index is synchronized. Once invalidated, the next query which runs will then re-cache the results. This technique is therefore probably not useful in systems with very frequently updated indexes.

The syntax for ctxfiltercache is

```
ctxfiltercache((query expression)[, save_score][, topN)')>0;
```

Where save_score and topN are both boolean (TRUE or FALSE) values.

save_score, as the name suggests, tells the system whether to save the score of the expression in the cache. Frequently, the sort of clauses which you would want to cache are *filter* clauses, and should not contribute to the score of the query. For example a security clause decides whether or not the user should see a particular item, but is not relevant to the score of that item. In this case you would set save_score to FALSE, and the expression would return 100 - the maximum score value. Since the AND operator always returns the lower of its operands, this ensures that the expression cannot affect the score.

TopN can only be set to TRUE if save_score is TRUE. It specifies whether only the highest-scoring hits should be cached. This saves storage, but means that if the stored hits are exhausted (due to the query being sorted by criteria other than score, for example, or by the application fetching too many rows) then the query will have to revert back to standard operation, which is likely to be slower than not using the filter cache at all.

Forward Index and Save Copy

FORWARD_INDEX

To understand this feature, we're going to take another look at the internals of the $I table. We'll create an index on a table with two rows:

ROWID	Text
r1	Night and day, day and night
r2	It was a wild and stormy night

Conceptually, the $I index table will look like this:

TOKEN_TEXT	TOKEN_INFO
night	1 <1, 6> 2 <7>

day	1 <3, 4>
wild	2 <4>
stormy	2 <6>

In the token_info column, we have DOCIDs, and "offsets". It's important to realise that these are WORD offsets: we have the word "night" at word position 1 and 6 in DOCID 1. Word offsets are needed when doing co-location (phrase) searches, and NEAR searches. We don't care how many characters apart the target words are, but we do care how many words they are apart.

But what happens if we want to do highlighting of search results? ctx_doc.highlight has to return the *character* offsets of the search terms, so that we can mark up the original text. We simply can't get that information from the text index, so we must go back to the original document source, and pass the document through the whole document indexing pipeline - which may include a complex datastore procedure, AUTO_FILTER, sectioning and lexing. All this is expensive, and especially so if the document is large. Let's say the first page of our result set consists of 10 multi-page PDF documents - producing highlight information for all those documents is going to be very costly.

How can we avoid this? One obvious solution would be to store character offsets as well as word offsets in the $I table. But that would significantly increase the size of the $I table, and therefore slow down all queries, whether or not they needed to do highlighting.

So instead, we have introduced a new table, the $O or "Offsets" table. This has a row for each indexed document (that is, each DOCID) and a map between word offsets, as used in the index, and character offsets, as required by highlighting functions.

So now when we call ctx_doc.highlight, we can get the word offsets from the index (as an aside, what we're actually doing here is effectively a "functional lookup" in the index - we know the ROWID of the row in question but still need to check how the query terms appear in that row) and then we can translate those into character offsets using the $O table.

This $O table is known as the "forward index", and is controlled by a STORAGE preference attribute FORWARD_INDEX, taking the values 'TRUE' or 'FALSE'.

SAVE_COPY

The forward index gives us fast highlighting, but what about other ctx_doc document services: filter, markup and snippet?

filter returns the document text as output from the datastore and filter - effectively the text that was indexed. snippet and markup return some or all of that text, with the search terms highlighted. All of these have similar issues to those already discussed - we need to feed the document through the datastore and filter, or the whole document processing pipeline before we can return anything to the user.

In these cases, forward index alone doesn't help us. We must have the text of the document. So to solve that, we've introduced another storage preference attribute SAVE_COPY. This allows us to keep a compressed rendering of the document itself within the index, complete with stopwords, punctuation, etc. so that we can regenerate it on request. This is stored in another new table - the $D table.

SAVE_COPY comes in two flavors, PLAINTEXT and FILTERED. If you choose PLAINTEXT, the document will be stored without HTML tags. This is compact, and ideal for snippet generation, since snippets do not need HTML markup. However, if you want to use ctx_doc.markup, or ctx_doc.filter (both of which need to return HTML) then you will need to chose the FILTERED option, which stores the document as it comes back from the filter.

There is additional knob: SAVE_COPY_MAX_SIZE. This prevents excessive storage use by large documents. If you set this to 4000, then only the first 4000 characters of the filtered or plaintext document will be stored. This is usually enough to find snippets, but if the search term(s) are not found within this maximum size, the start of the document will be used as the snippet instead.

The following table acts as a quick lookup for the required parameters for the various services. If the required option is not in place, then the system will fall back to the original way of doing things, passing the document through the document processing pipeline. Note that if the function requires SAVE_COPY PLAINTEXT, and you've chosen SAVE_COPY FILTERED, there is no need to fall back to the document pipeline, but some extra work will be necessary to remove HTML formatting.

Service	**FORWARD_INDEX**	**SAVE_COPY FILTERED**	**SAVE_COPY PLAINTEXT**
highlight	R	-	-
filter	-	D	D
markup	R	D	D
snippet	R	-	R

KEY:

R : required.
D : Depends on "plaintext" parameter to service.
- : Not needed

New "Mild Not" (MNOT) Operator

Sometimes you want to find a word which is more commonly found as part of a phrase. For example, if I'm looking for "York", the English city, I don't want to find references to "New York". I decide to search for "york NOT new york", which would find all documents which contain the word "York" but not the phrase "New York". Almost right, but what about documents which have references to both? If "York" appears somewhere in the document on its own, but the document also contains the phrase "New York" then I won't find that document using NOT. Hence we have introduced the new Mild Not (MNOT) operator, which allows us to search for "york MNOT new york", or "the word York when it's not part of the phrase 'New York'".

Note that this operator is present but not documented in 11g, in versions 11.2.0.2 or later. Since it's not documented, it's not officially supported, but many customers have used it successfully.

Snippet Enhancements

We've introduced a few enhancements to the popular "snippet" functionality for display hit words in their document context. Previously, you only ever got a single string out, which wouldn't necessarily contain all your search terms. Now you can get multiple "chunks" (segments) out. New arguments to ctx_doc.snippet control this behavior:

- **entity_translation**: Converts HTML/XML entities such as "<" and "&" into their alternate form "&lt;" and "&amp;"
- **separator**: the string used to separate snippet segments. Defaults to elipsis (three dots) in HTML bold tags: "<b>...<b>"
- **radius**: the approximate number of characters to display either side of the hit term. The radius will be adjusted to try to get the snippet to start and end on a word, or preferably a sentence, boundary. A value of 0 will cause Oracle Text to make its own decisions.
- **max_length**: the total length of all segments, plus the necessary separators. Defaults to 250 and cannot exceed 4000.
- **use_saved_copy**: Controls behaviour when save_copy is used. See the documentation for further details.

BIGRAM mode for the Japanese VGRAM Lexer

The Japanese VGRAM lexer is very efficient in terms of space storage. However, certain queries cannot be satisfied directly from the index, and we need to wildcard searches within the index. This can be quite slow. Using BIGRAM mode means all possible 2-gram tokens are stored, avoiding the need for wildcard searches at the cost of extra storage in the index.

ORACLE®

New Features in Oracle Text with Oracle Database 12c
June 2013
Author: Roger Ford

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200

oracle.com

Oracle is committed to developing practices and products that help protect the environment

Hardware and Software, Engineered to Work Together