Virtuoso Text and XML Indexing

Abstract

This article presents an overview of Virtuoso's text and XML indexing capabilities. These are divided in two main functionalities:

- Full text index and text retrieval
- Filtering data against stored query patterns

The first is a classical full text extension to SQL, adding XML aware indexing logic for XML content.

The second feature allows associating "text triggers" to a text or XML indexed column. A text trigger is a full text or XPATH expression which gets matched against all incoming data. Logic is used to avoid matching all triggers to all documents. This feature makes it possible to efficiently support fine-grained personalized alerts and perform intelligent filtering on the content.

Introduction

Most applications deal with free form text at some point. Indexing text fields has become a normal feature in relational databases. The advent of XML as a common data transfer and semi-structured storage format places further requirements on text indexing.

Virtuoso has a flexible set of SQL extensions for both retrieving text data as well as tracking changes in text data.

Virtuoso's full text features make it possible to include search engine like capabilities for data hosted in Virtuoso's DAV repository as well as any application data. Having the text index tightly integrated in the database has the advantage of requiring no extra maintenance, installation or backing up. Further, it is easy to combine text and relational search criteria in SQL statements.

Overview

A long varchar or xmltype column may have an associated full text index. Optionally, such a column may also store other data, define pre-indexing processing and have the text index follow the data either in strict synchrony or at a delay, allowing for bundling operations together for better throughput.

In this article , we will develop the basics of a news tracking application which reads ATOM feeds and allows users to search the feed and to register personalized alerts.

To run the samples on your own Virtuoso installation, download textsample.sql. Connect to the server with the isql utility and use the load command to run the whole file or cut and paste individual commands to the SQL prompt.

Creating the News Table

First, we create the tables:

```
create table news (n_id int identity
(start with 1), n_date datetime, n_text xmltype,
n_source varchar,
n_title varchar,
primary key (n_id));

create text xml index
on news (n_text) with key n_id clustered with (n_date,
n_source,
n_title);

create text trigger on news (n_text);
```

Here we create a table where we will make one row per retrieved article. We make an XML-aware full text index on the table, specifying that the text index will use the table's already existing primary key for referencing the data and that the text index will also duplicate the source, title and date information. In this way, if we only access this information, we will not have to refer to the actual table at all since the whole query can be evaluated within the text index.

Filling the News Table

The below stored procedure reads and ATOM feed and updates the news table, inserting data if the data is not already there.

```
create procedure news_download (in url varchar)
{
  declare cnt any;
  declare xt, xp any;
  declare l int;

  cnt := http_client (url);
  xt := xml_tree_doc (cnt);

  xp := xpath_eval ('[xmlns:atom="http://www.w3.org/2005/Atom"]
//entry', xt, 0);
  l := length (xp);

  for (declare i int, i := 0; i < l; i := i + 1)
    {
      declare title, link, dt varchar;
      title := xpath_eval
```

```
('[xmlns:atom="http://www.w3.org/2005/Atom"]
./atom:title/text()', xp[i]);
      dt := cast (xpath_eval
('[xmlns:atom="http://www.w3.org/2005/Atom"]
./atom:published/text()', xp[i]) as varchar);
      link := xpath_eval
('[xmlns:atom="http://www.w3.org/2005/Atom"] ./atom:id/text()',
xp[i]);
      if (not exists (select 1 from news where n_source = link))
        insert into news (n_date, n_text, n_source, n_title)
        values (dt, xml_cut (xp[i]), link, title);
      commit work;
    }
}
```

The procedure takes the URL of the feed, retrieves the text, parses it as XML and extracts all atom:entry elements. It then loops over these elements, extracting title, date and link and stores the entry into the news table if it is not already there. It commits after every article so as not to leave locks on for too long.

We can call this right away, we will use the ATOM feed of the Virtuoso product blog as an example:

```
news_download
('http://virtuoso.openlinksw.com/blog/gems/atom.xml');
```

Querying News

Now we are ready to retrieve news based on content:

select * from news where contains (n_text, 'database');

Is the simplest text query. It returns all rows whose n_text contains the word database.

To sort them by relevance, we do:

select score, * from news where contains (n_text, 'database') order by score desc;

The score is a virtual column introduced by the contains predicate. It is an integer relative to the frequency of the word in the search hit and to word proximities if the search pattern has multiple words.

To have a nicer presentation, we can use the search_excerpt function. This makes a summary of the text of search hits, highlighting and showing the environment of the matched words. This is similr to how Google shows search matches.

select search_excerpt (vector ('database'), cast (n_text as varchar), 10000, 500, 80) from news where contains (n_text, 'database');

The n_text is cast as varchar, returning a string containing the text values of all the XML

elements in it, similarly to the string () XPATH function. The search_excerpt function takes an array of the words to highlight, the text of the search hit, a search limit, in this case looking for hits in the 10000 first characters, the desired length of the search summary, and the desired length of individual hit quotes.

```
SQL> select n_source, search_excerpt (vector
('database'), cast (n_text as varchar), 10000, 500, 80) from news
where contains (n_text, 'database');
N_SOURCE callret
VARCHAR VARCHAR
_______________________________________________________________
_____________

http://virtuoso.openlinksw.com/weblog/kidehen@openlinksw.com/127/
?id=951
... RDF Triple Store or <b>Database</b> that supports SPARQL
Query
Language Transport Protocol and XML...
http://virtuoso.openlinksw.com/weblog/vdb/136/?id=950 ...
Relational
<b>database</b>. Thus this continues to be improved ranging from
low
level <b>database</b> engine work to...
http://virtuoso.openlinksw.com/weblog/oerling/?id=949 ...
Relational
<b>database</b>. Thus this continues to be improved ranging from
low
level <b>database</b> engine work to...
http://virtuoso.openlinksw.com/weblog/kidehen@openlinksw.com/127/
?id=947
...
```

We don t want to provide a direct SQL query service over the main Swoogle **database** because it s... The search_excerpt function expects an array of words to highlight. We can define an SQL function that produces this array given the text of the search pattern:

```
create procedure  vt_words_1 (in tree any, inout res any)
{
  if (isstring (tree))
    {
      res := vector_concat (res, vector (tree));
    }
  else if (isarray (tree))
    {
      foreach (any elt in tree) do
        vt_words_1 (elt, res);
    }
}

create function vt_words (in q varchar) returns varchar array
{
  declare v any;
  v := vector ();
```

```
  vt_words_1 (vt_parse (q), v);
  return v;
}
```

The vt_parse function is a built in function returning the parse tree of the text search expression as nested arrays. The vt_words_1 procedure does a recursive descent of the tree and appends all strings into the output variable. Lisp is nicer for these things but Virtuoso/PL will also do in a pinch.

Now, for best throughput when reading hits from a text index, we wish to avoid having to retrieve the data row corresponding to every text hit, specially since we in practice never display all hits. This is where the offband data declared with the clustered with clause in the create text xml index statement comes in. In this exampl, we retrieve the five newest articles that mention the phrase "open source."

```
 select search_excerpt(vt_words ('"open source"'), cast (n_text
as varchar), 10000, 500, 90)
  from  (select top 5 n_id from news where contains (n_text,
'"open
source"', offband, n_date) order by n_date desc) f, news n
  where f.n_id = n.n_id option (order);
```

The nested select in the from clause gets the top 5 n_id's sorted by n_date descending. It does this without touching the news table itself. The n_id's are retrieved the from text index, then the corresponding n_dates are retrieved also from the text index and sorted. The outer select sees 5 integers. It then uses these to retrieve the corresponding news rows and n_text LOB's and make the search summaries. The option (order) at the end ensures that the SQL compiler will join from left to right, so as to get the intended evaluation order.

The offband keyword declares that the column mentioned after the keyword will come from the text index.

Now, since all columns referenced in the inner query come from the text index, there is no longer any need to retrieve the actual row from the news table. Due to the sequential access merge join operation of a text index, it is usually more efficient to get data from the text index than to join with the data table. This provides better working set behavior and can take advantage of smart read-ahead, which the random access seek into the document table would not do.

Using XML Conditions with a Text Index

So far, we have only used text predicates. The xcontains predicate allows us to use an XPATH expression instead of a text condition. Otherwise, all the options of contains such as offband data remain available.

The xcontains predicate preprocesses the XPATH condition so as to extract conditions which can be resolved using the text index. It will first perform a text lookup, then will retrieve the actual data and match the XPATH expression with the data, allowing for extracting specific

elements, testing order of elements, calling XPATH functions and so forth.

We can do

```
select n_title from news where xcontains (n_text,
'[xmlns:atom="http://www.w3.org/2005/Atom"] //atom:title[text-
contains
(., "virtuoso")]');
```

This will return the titles of all articles whose atom:title element contains the word Virtuoso.

But we can go further, also retrieving and returning XML elements. The following query returns the authors of all articles with Virtuoso in the title. The author is a virtual column that is bound to all evaluations of the path in xcontains:

```
select author, n_title from news where xcontains (n_text,
'[xmlns:atom="http://www.w3.org/2005/Atom"] .[
//atom:title[text-contains (., "virtuoso")]]//atom:author',
author);
AUTHOR N_TITLE
VARCHAR VARCHAR
_______________________________________________________________
_____________

<n0:author xmlns:n0="http://www.w3.org/2005/Atom">
<n0:name>Kingsley Idehen</n0:name>
<n0:email>kidehen@openlinksw.com</n0:email>
</n0:author> Virtuoso &
SPARQL Demo
<n0:author xmlns:n0="http://www.w3.org/2005/Atom">
<n0:name>Kingsley Idehen</n0:name>
<n0:email>kidehen@openlinksw.com</n0:email>
</n0:author> Virtuoso is
Officially Open Source!
<n0:author
xmlns:n0="http://www.w3.org/2005/Atom">
<n0:name>Virtuoso DataSpace
Bot</n0:name>
<n0:email>virtuoso@openlinksw.com</n0:email>
</n0:author> Introducing Virtuoso Open Source Edition
<n0:author
xmlns:n0="http://www.w3.org/2005/Atom">
<n0:name>Orri
Erling</n0:name>
<n0:email>oerling@openlinksw.com</n0:email>
</n0:author> Introducing Virtuoso Open Source Edition
```

The text-contains predicate, which is an XPATH extension allowing a free text match of an element's text value, is first applied to the entire text of the article. If the article is selected, the path retrieves the authors. The namespace declaration is given in the leading clause in brackets, not to be confused with an XPATH predicate.

The text index is used to look for the word Virtuoso and the elements atom:title and atom:author. Thus the path does not have to be evaluated against documents where all the three items are not present.

select n_title from news where xcontains (n_text,
'[xmlns:atom="http://www.w3.org/2005/Atom"] //atom:name = "Kingsley Idehen"');

Will use the text index to look for Kingsley Idehen as the value of am atom:name element. The text index can resolve the inclusion of a string inside anelement through the use of word position data and special tagging distinguishing between text and element names.

Personalized Alerts

Now we can add personalized alerts to the news application.

In the table declaration, we had the statement

Create text trigger on news (n_text;)

This creates tables for storing XPath and text queries that will automatically be applied to all data coming into the index.

Only those queries will be tried which have a chance of succeeding. The system constructs a reverse index on queries. When text is inserted into the index, the system looks for queries which mention a word of the text. For large masses of queries, this is much more efficient than periodically re-running queries.

The system will take the most discriminating term of each query and will evaluate the query only when a document containing this term comes into the index.

When a stored query matches an incoming document, a row is inserted into an automatically created hits table. These hits can then be read for sending notifications to users, for display on a dashboard or other such purposes.

The create text trigger statement automatically generates stored procedures for registering text or xpath triggers. These are named after the table concerned.

```
tt_query_news ('"open source"', 0, 'Article on open source
software', 'john@example.com');
tt_query_news ('virtuoso', 1, 'Article on
Virtuoso','mary@example.com');
```

The first argument is a text query. The next is a user reference number. This is an arbitrary number used to group together queries belonging to the same user. This does not have to refer to any system tables but in principle could. The next argument is a text description of the query and an email address for sending hit notifications.

When articles matching such a stored query come in and match the condition, the matches can

be read from a hits table where the title and notification recipient of registered with the query are repeated.

We can nowrefill the news table:

```
delete from hits;
news_download
('http://virtuoso.openlinksw.com/blog/gems/atom.xml');
```

We can now look at hits for the stored conditions in the imported text:

```
SQL> select * from
news_n_text_hit;
TTH_U_ID TTH_D_ID TTH_T_ID TTH_TITLE TTH_URL TTH_TS
TTH_NOTIFY
INTEGER NOT NULL VARCHAR NOT NULL INTEGER NOT NULL
VARCHAR VARCHAR BINARY NOT NULL VARCHAR
_______________________________________________________________
_____________

0 48 1 NULL NULL 2006-04-15 11:39:20 john@example.com
0 50 1 NULL
NULL 2006-04-15 11:39:20 john@example.com
0 51 1 NULL NULL
2006-04-15 11:39:20 john@example.com
0 52 1 NULL NULL 2006-04-15
11:39:20 john@example.com
1 48 2 NULL NULL 2006-04-15 11:39:20
mary@example.com
1 50 2 NULL NULL 2006-04-15 11:39:20
mary@example.com
1 51 2 NULL NULL 2006-04-15 11:39:20
mary@example.com
1 52 2 NULL NULL 2006-04-15 11:39:20
mary@example.com
```

The table news_n_text_hit is automatically created by the create text trigger statement. The name is the name of the table plus the indexed **column plus _hit**. The **tth u_id** referes to the user id given with **tt_news_query**. The **tth_d_id** is the document id used in the text index on **n_text** of news, in this case this is a foreign key to **n_id**.

Now we can make search summaries from the hits detected when retrieving the articles:

```
select search_excerpt (vt_words(tt_query), cast (n_text as
varchar), 10000, 400, 80), tt_query,
tt_comment, tth_notify
  from news, news_n_text_hit, news_n_text_query
  where n_id = tth_d_id and tt_id = tth_t_id;
callret TT_QUERY
TT_COMMENT TTH_NOTIFY
VARCHAR VARCHAR VARCHAR VARCHAR
_______________________________________________________________
```

```
--------------

... is one of the many interesting aspects of the recent Virtuoso
<b>Open</b> <b>Source</b> release.  "open source" Article on open
source software john@example.com
Virtuoso is Officially <b>Open</b>
<b>Source</b> http virtuoso. ... is now available in <b>Open</b>
<b>Source</b> form. ... Why <b>Open</b> <b>Source</b>...  "open
source" Article on open source software john@example.com
Introducing
Virtuoso <b>Open</b> <b>Source</b> Edition http virtuoso. ...
Virtuoso
<b>Open</b> <b>Source</b> Edition I am Orri Erling...  "open
source"
Article on open source software john@example.com
Introducing
Virtuoso <b>Open</b> <b>Source</b> Edition http virtuoso. ...
Virtuoso
<b>Open</b> <b>Source</b> Edition marks a new...  "open source"
Article on open source software john@example.com
<b>Virtuoso</b>
SPARQL Demo http <b>virtuoso</b>. ... recent <b>Virtuoso</b> Open
Source release. ... of <b>Virtuoso</b> 39 s SPARQL
Implementation...
virtuoso Article on Virtuoso software mary@example.com
<b>Virtuoso</b> is Officially Open Source http
<b>virtuoso</b>. ... the fact that <b>Virtuoso</b> is now
available in
Open Source...  virtuoso Article on Virtuoso software
mary@example.com
Introducing <b>Virtuoso</b> Open Source Edition
http <b>virtuoso</b>. ... <b>Virtuoso</b> Open Source Edition I
am
Orri Erling...  virtuoso Article on Virtuoso software
mary@example.com
Introducing <b>Virtuoso</b> Open Source Edition
http <b>virtuoso</b>. ... Orri Erling program manager for
<b>Virtuoso</b> at...  virtuoso Article on Virtuoso software
mary@example.com
```

This query joins the search hits to the queries table and the news table, composing search summaries of the hits that were found. To see the details of the generated tables, please refer to the text triggers section of the Virtuoso documentation.

The results of a similar query could be used for composing notification emails or could be shown on a dashboard web page.

Batching Text Processing

A text index is a Virtuoso database object, subject to the same transaction control as all other database objects. In some other systems, the text indexing engine is separate and does not automatically come under the same management as the regular database. For purposes of

backups, transaction logging, ACID properties etc., the text index is indistinguishable from any other index.

However, many applications do not require strict immediate consistency between a text index and the table. It is several times more efficient to bundle document insertions and deletions together and update the index in batch mode. To this effect, Virtuoso provides the function

```
--- set the news index into batch mode with a 2 min cycle
vt_batch_update ('DB.DBA.news', 'ON', 2);
```

This statement specifies that the index for news will be updated every two minutes, batching all updates in each two minute interval together. Note that the case of the table name given as a string must be an exact match. This will cause the text index processing never to block or slow down article insertion. The index will be asynchronously maintained on a separate thread. The same applies to text triggers. These too benefit from batching. These will be run immediately before each batch of updates to the text index.

Conclusion

We have seen how XML data can be accessed with both text and XPATH criteria. Virtuoso provides a powerful collection of features for optimizing free text and semi-structured text access.

The capacity to store queries and index them with the data is a specialized feature for high volume filtering of text and XML data. This makes implementing services like www.pubsub.com or any personalized notification scheme a mater of writing a few web pages for user interface.

These features make Virtuoso a powerful, flexible platform for building any filtering and content management applications.