ORACLE

A New User's Guide to Oracle Text in Oracle Database

Putting your text data to work.

September, 2023, Version 1.0

ORACLE

Table of contents

ORACLE

Introduction

We're all familiar with using search engines such as Google to do word-based searching on the Internet. But many people don't realize that the Oracle Database has similar search technology built right in.

Oracle Text, Oracle's integrated full-text retrieval technology, is part of Oracle Database for all editions. Oracle Text uses standard SQL to index, search, and analyze text and documents stored in the Oracle database.

Plain text is typically stored in VARCHAR2 and CLOB columns, but you can also index binary documents such as Word or PDF files stored in BLOB columns. You can even index files held externally on a file system, on the web or on cloud storage, with just a pointer to the external file stored in the databases.

Oracle Text supports multiple languages and character sets, including all space separated languages and pictogram languages such as Chinese, Japanese or Korean.

Queries against text indexes allow for inexact searches, with advanced, customizable relevance ranking algorithms to return the most appropriate results first.

Some Use-Cases

There is no such thing as a "typical" Oracle Text application. But here are just a few examples of how it is used by real customers.

Company Name Search

A purchasing system has a list of supplier companies. The company name is held in a short VARCHAR2(80) field. Often people won't know the full company name but will know one or two words in that name. Creating a text index on the company name field allows users to do searches such as:

- **Partial name lookup** - find all the companies with a particular word - such as "Chemical" in their name.
- **Fuzzy match** - if you're not sure whether it's "Philips" or "Phillips" you can do a fuzzy search to find both.
- **Stem search** - find all companies with a word starting with "chem", eg. "chemical", "chemicals" or "chemistry".

Document Management

A document management system has checkin/checkout/download capabilities. Documents are stored in their original binary form, either in a BLOB column or in external files. Other columns in the table include information such as author, creation date, checked-out status etc. Creating an index on the binary data allows users to do content-based searches, often in conjunction with searches against the other, more structured, columns in the table. In this application users might do searches such as:

- **AND searches** - find all the documents that mention "Microsoft" and "employment"
- **Topic-based searches** - find all the documents which are about finance or politics (even if those words aren't specifically mentioned)
- **Proximity searches** - find all the documents that mention "Microsoft" near the word "windows". Documents will be ranked according to the number of matches and the closeness of the terms.

Content Analysis

A major online retailer and marketplace collects product reviews and sales data. Customers can use text search to find products in the catalog, but the retailer is also able to do machine learning based analysis on product review data to understand more about their customer base and their products. Advanced capabilities of Oracle Text used here include:

- **Sentiment analysis** - the ability to recognize positive and negative reviews from the review text
- **Clustering** - group similar reviews together to identify fake reviews. Where a particular seller has many similar reviews from different "customers", there is likely to be a problem.
- **Named Entity Recognition (NER)** - the ability to recognize particular "things" in review text. Review policies forbid the mention of prices (which can change), and other products or sellers within reviews. NER identifies such things within review text and can flag that review for further checks.

Getting started with Oracle Text

The following examples work in any supported Oracle database release. When applicable, differences in syntax between releases will be explicitly mentioned.

Let's say we have a table called "customers" which was created as follows:

```
CREATE TABLE customers (
     cust_id     NUMBER,
     cust_name   VARCHAR2(80),
     create_date DATE );
```

And we've added some data to it as follows:

```
INSERT INTO customers VALUES (1, 'The Acme Manufacturing Company, Inc.', SYSDATE);
INSERT INTO customers VALUES (2, 'Coyote Trap Construction GMBH', SYSDATE);
```

Now we can create a text index using the "create search index" syntax. In Oracle Database 21c and beyond it's as easy as follows:

```
CREATE SEARCH INDEX cust_text_index ON customers (cust_name);
```

If you're using a database version prior to 21c, you will need to use a different syntax to create the index. The following will create an index which is synchronized automatically at commit time:

```
-- this syntax is used prior to 21c:
CREATE INDEX cust_text_index ON customers (cust_name)
INDEXTYPE IS ctxsys.context
PARAMETERS ('SYNC (ON COMMIT)');
```

Now we can use the CONTAINS query operator to find rows in this table. CONTAINS is a function that returns 0 if there's no match or more than 0 if there is a match. It takes the column name to search and a text query:

```
SELECT cust_id, cust_name FROM customers
WHERE CONTAINS ( cust_name, 'acme' ) > 0;
```

Since this is SQL we can easily combine CONTAINS with other operators:

```
SELECT cust_id, cust_name FROM customers
WHERE CONTAINS ( cust_name, 'construction' ) > 0
AND create_date > '01-JAN-23';
```

To look at relevance we'll need to add another text:

```
INSERT INTO customers VALUES (3, 'Trapezium and Trappist Developments', SYSDATE);
```

We'll need to commit that,and wait a few seconds for the index to be updated by the background indexing task (unlike normal BTREE indexes, text indexes are not synchronous by default).

```
COMMIT;
```

Now, relevance is a complex topic, but at the basic level if a search term you're looking for occurs more frequently in one result, that result is more relevant. So let's look for words beginning 'trap':

```
SELECT cust_id, cust_name FROM customers
WHERE CONTAINS ( cust_name, 'trap%' ) > 0;
```

Note the '%' wildcard. As in a SQL LIKE clause, % represents a multi-character wildcard, but compared with the SQL operator LIKE. Text CONTAINS searches are case-insensitive by nature.

That will return rows 2 and 3, but in an indeterminate order. If we want to see the most relevant one first, we must use the SCORE() operator, along with a further parameter to CONTAINS. This third parameter is a number (it doesn't matter what), which is also

used in the SCORE operator. The number just ties the SCORE to a specific CONTAINS, in case your query has more than one CONTAINS, each of which would have its own score.

```
SELECT SCORE(1), cust_id, cust_name FROM customers
WHERE CONTAINS ( cust_name, 'trap%' ,1 ) > 0
ORDER BY SCORE(1) DESC;
```

We used "1" here but could have used 99, or 768. It doesn't matter, so long as it's the same in each place.

The output will look something like:

```
 SCORE(1)    CUST_ID CUST_NAME
---------- ---------- ----------------------------------------
         7          3 Trapezium and Trappist Developments
         4          2 Coyote Trap Construction GMBH
```

There is no *absolute* meaning to that SCORE() value. The maximum value is 100 but you can't say "the first one is 7% relevant". It is designed simply so that more relevant results can be returned before less relevant ones. Generally speaking, your application will sort by the score and decide on the top N to show, but will not return it as a user-visible value.

Architecture

The examples we've looked at so far are for a standard, default text index. If you want to index plain English text in a database column, a default index should be all you need. But a text index is almost infinitely customizable. To understand what you can customize (and why you might want to) we need to understand a bit more about how documents are indexed.

This section looks at the mechanism for processing text with Oracle Text. This process can be considered as a pipeline (Figure 1). Each stage of the pipeline is customizable. This section discusses each stage, and considers some of the options available at that stage.

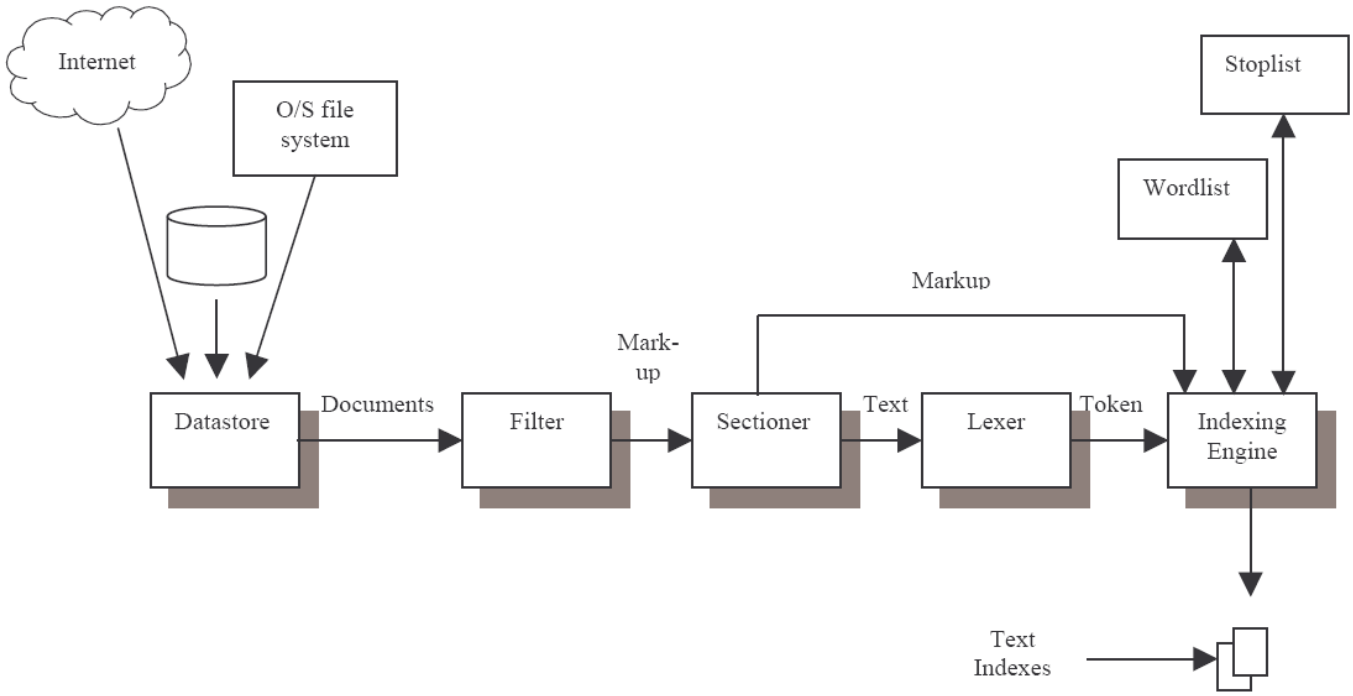

Figure 1: Indexing Architecture

Datastore

The datastore defines from where the text to be indexed is stored. If the text is stored in a normal database column, there is no need to specify a datastore (the default DIRECT_DATASTORE is used). Supplied datastores allow for text which is stored within a database, on a file system (the DIRECTORY_DATASTORE), or accessed remotely via the HTTP or HTTPS protocols (the

NETWORK_DATASTORE). Custom datastores may be defined which fetch the data from a location, protocol or application of the customer's choice.

Default Datastore

The default datastore is in the database itself. Text may be stored in a VARCHAR2 column or in a CLOB (Character Large OBject) column. Formatted text (such as Word or PDF documents) can be stored in BLOB (Binary Large Object) columns.

Directory Datastore

Text to be indexed is stored on a file system which is accessible to the database server. A database DIRECTORY object is provided when the datastore is defined, and file names (optionally with sub-directory paths) are stored in the indexed column of the table.

Network Datastore

The database contains an HTTP protocol URL, and the text to be indexed is fetched directly from the URL at indexing time. This uses the standard database Access Control List (ACL) mechanism to ensure controlled access to external sources.

User Defined Datastore

A PL/SQL procedure is specified, which will be called for each row in the table being indexed. The PL/SQL procedure may, in turn, call other language programs such as Java (directly, if running in the database) or C/C++ programs via the EXTPROC external procedures mechanism. This gives the customer complete control over what gets indexed.

Filter

The filter stage is responsible for processing "formatted" documents such as Microsoft Office files or PDF documents. The built-in AUTO_FILTER recognizes all common document formats and can translate them into indexable HTML text.

Application developers may replace the filter stage with their own custom-built filter, or a filter purchased from a third-party.

A custom filter is simply an executable program or script that takes two arguments, the first being the file containing the formatted input text, and the second being the name of the file where the filtered output should be written. If required, a custom filter can call the standard — (auto) filter. This allows it to process any file formats unique to the business, but pass on any standard file formats to the standard filter.

Sectioner

The sectioner object is responsible for identifying the containing section(s) for each text unit. For example we might want to index the following text:

```
<title>A Tale of Two Cities</title>
<author>Charles Dickens</author>
It was the best of times, it was the worst of times…
```

We could use the AUTO_SECTIONER to index this text, which would identify the XML-like section tags around the title and author, and would allow us to do a search such as :

```
WHERE CONTAINS (text, 'Dickens WITHIN author') > 0
```

Alternatively we could have used the BASIC_SECTIONER and predefined the title and author sections with the tags that identify them.

Sectioners also allow us to include structured values (strings, numbers and dates) within a text search.

Lexer

The lexer's job is to separate the sectioner's output into words or tokens. In the simplest case for a Western European language, the lexer just splits text into uninterrupted strings of alphanumeric characters. So the string:

> Aha! It's the 5:15 train, coming here now!

would be split into the words, minus any punctuation or special symbols:

aha it s the 5 15 train coming here now

The lexer typically removes stopwords, which are common words defined by the application developer, or taken from a default list. That would likely reduce the list above to:

aha * * * 5 15 train coming * now

Note the asterisks representing removed stopwords. Although they are not actually indexed, the presence of a stopword at the position is noted in the index. In a search, any stopword will match that word when used as part of a phrase. For example, the search "*coming in now"* would match the text above (because "*in*" is a stopword and will match any stopword in the text), but "*coming rapidly now*" would not match it, because "rapidly is not a stopword, and is not found in the original phrase.

The set of stopwords may be specified by the application developer, who can also choose to explicitly define all numbers as stopwords.

Lexer Preferences

There are many options available for fine-tuning the lexer. For example, the developer can choose that an index should be case sensitive or case insensitive, and can choose whether particular characters should split tokens or be indexed as part of them – for example, should "*PL/SQL*" be indexed as two terms "*PL*" and "*SQL*" or the single string "*PL/SQL*".

Indexing Engine

The indexing engine creates the inverted index that maps tokens to the documents that contain them. In this stage, Oracle Text uses - if specified - a *stoplist* where users can specify words or themes which should be excluded from the text index.

The final output of the pipeline is an *inverted index*. This is a list of the words from the document, with each word having a list of documents (and sometimes document sections) in which it appears. It is called inverted because it is the inverse of the normal way of looking at text, which is a list of documents where each document contains a list of words.

Customizing the Index

An index is customized using the *parameters* clause of the create index statement. This applies whether we're looking at the "*create search index*" syntax in 21c onwards, or the "*create index … indextype is ctxsys.context*" of earlier versions. Let's look at a simple example. We're indexing a set of songs:

```
create table songs (songtitle varchar2(80));

insert into songs values ('let it be');

create search index songtitleindex on songs(songtitle);
```
(Oracle Database 21c onwards)

```
CREATE INDEX songtitleindex ON songs (songtitle) INDEXTYPE IS ctxsys.context PARAMETERS ('SYNC
(ON COMMIT)');
```
(earlier versions)

Now, if I do a search for 'let' I will find the entry:

```
SQL> select * from songs where contains (songtitle, 'let') > 0;

SONGTITLE
--------------------------------------------------------------------------------

let it be
```

But if I search for 'it', I won't find it:

```
SQL> select * from songs where contains (songtitle, 'it') > 0;

no rows selected
```

What's going on here? The word 'it' is a default stopword (assuming your database was installed with English as its default language) and stopwords aren't indexed. If we want all words to be indexed, we can use a pre-defined STOPLIST called empty_stoplist in the CTXSYS schema:

```
create search index songtitleindex on songs(songtitle) parameters ('stoplist
ctxsys.empty_stoplist');
```
(Oracle Database 21c onwards)

```
CREATE INDEX songtitleindex ON songs (songtitle) INDEXTYPE IS ctxsys.context PARAMETERS ('SYNC
(ON COMMIT) stoplist ctxsys.empty_stoplist' ); (earlier versions)
```

Now I'll be able to search for common words such as 'it':

```
SQL> select * from songs where contains (songtitle, 'it') > 0;

SONGTITLE
--------------------------------------------------------------------------------
let it be
```

We won't go into the system of creating customized preferences and stoplists here, you should refer to the documentation for more details.

Benefits of Integrated Text Search Capability

External text engines are widely available. Most rely on text in files, or just text passed to the engine over a REST API. But if your text is in the database to start with, it makes much more sense to use the integrated text engine within the Oracle Database.

This means the developer or application owner has:

- A single repository for all data (text and structured) instead of two. This is easy to maintain, backup, etc.
- Indexes in the same repository. This makes for efficient processing of text and mixed queries.
- A single API for developing applications. Leverage Oracle's converged database and combine text search with any other processing in the database, e.g. spatial, JSON processing, or graph.
- Integration with the Oracle SQL execution engine and query plan optimizer. Let Oracle decide on the most efficient execution plan for the whole statement.

To summarize, the advantages of integration are apparent:

- **Low Cost**
  Oracle Text is part of all editions of Oracle Database. There are no separate products to buy, integrate, or maintain.
- **High Performance**
  The database will choose the fastest plan to execute queries that involve both text and structure content.
- **High Integrity**
  Since text is stored in the database it inherits all the integrity benefits œ for example, any update to the database can be reflected to the text search functionality, which means users can get an integrated, holistic view of all their data.
- **Low Complexity**
  Text is treated just like structured data. It is easy to develop and integrate text search applications with existing systems.
- **Superior Manageability**
  Oracle Text can be managed from standard enterprise management tools, leveraging commonly available administrators' skills.
- **Security**
  Oracle Text leverages the security features of the database.

ORACLE

Connect with us

Call +**1.800.ORACLE1** or visit **oracle.com**. Outside North America, find your local office at: **oracle.com/contact**.

blogs.oracle.com facebook.com/oracle twitter.com/oracle