Full Text Search in MySQL

Pattern matching (using the LIKE operator) enable you to look through any number of rows containing tex. It is also a common task to search for the same text in several string columns, but with pattern matching, that results in unwieldy queries as follows:

```
SELECT * from tbl_name
WHERE col1 LIKE 'pat' OR col2 LIKE 'pat' OR col3 LIKE 'pat' ...
```

However, when the text column is large and the number of rows in a table is increased, using these methods has some limitations:

- Performance: MySQL has to scan the whole table to find the exact text based on a pattern in the LIKE statement or pattern in the regular expressions.
- Flexible search: with the LIKE statement and regular expression search, it is difficult to have a flexible search query e.g., to find product whose description contains car but not classic.
- Relevance ranking: there is no way to specify which row in the result set is more relevant.

A useful alternative is full-text searching, which is designed for looking through large amounts of text and can search multiple columns simultaneously. FULLTEXT indexes are created on text-based columns (CHAR, VARCHAR, or TEXT) to help speed up queries and DML operations on data contained within those columns, omitting any words that are defined as stopwords.

1 FULLTEXT indexes

FULLTEXT indexes have an inverted index design. Inverted indexes store a list of words, and for each word, a list of documents that the word appears in. To support proximity search, position information for each word is also stored, as a byte offset.
For each InnoDB FULLTEXT index, a set of index tables is created, as shown in the following example (you cannot run this example on NMS's server as you don't have admin privileges, the following code is illustrative only):

```
CREATE TABLE opening_lines (
id INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY KEY,
opening_line TEXT(500),
author VARCHAR(200),
title VARCHAR(200),
FULLTEXT idx (opening_line)
) ENGINE=InnoDB;

mysql> SELECT table_id, name, space from INFORMATION_SCHEMA.INNODB_SYS_TABLES
WHERE name LIKE 'test/%';
+----------+--------------------------------------------------------+-------+
| table_id | name                                                   | space |
+----------+--------------------------------------------------------+-------+
|      333 | test/FTS_0000000000000147_00000000000001c9_INDEX_1 |   289 |
|      334 | test/FTS_0000000000000147_00000000000001c9_INDEX_2 |   290 |
|      335 | test/FTS_0000000000000147_00000000000001c9_INDEX_3 |   291 |
|      336 | test/FTS_0000000000000147_00000000000001c9_INDEX_4 |   292 |
|      337 | test/FTS_0000000000000147_00000000000001c9_INDEX_5 |   293 |
|      338 | test/FTS_0000000000000147_00000000000001c9_INDEX_6 |   294 |
```

```
|      330 | test/FTS_0000000000000147_BEING_DELETED            |   286 |
|      331 | test/FTS_0000000000000147_BEING_DELETED_CACHE      |   287 |
|      332 | test/FTS_0000000000000147_CONFIG                   |   288 |
|      328 | test/FTS_0000000000000147_DELETED                  |   284 |
|      329 | test/FTS_0000000000000147_DELETED_CACHE            |   285 |
|      327 | test/opening_lines                                 |   283 |
+----------+----------------------------------------------------+-------+
```

The first six tables represent the inverted index and are referred to as auxiliary index tables. When incoming documents are tokenized, the individual words (also referred to as "tokens") are inserted into the index tables along with position information and the associated Document ID.

2 Text Dataset

Full-text searching is best illustrated with a reasonably good-sized body of text. In this session, we are going to use the complete text of the King James Version of the Bible (KJV), which is both relatively large and nicely structured by book, chapter, and verse. Some sample records look like this:

```
O    Genesis    1    1    1    In the beginning God created the heaven and the earth.
O    Genesis    1    1    2    And the earth was without form, and void; and darkness was upon the face of the deep.
```

Each record contains the following fields:

- Book section (O or N, signifying Old or New Testament)
- Book name and corresponding book number, from 1 to 66
- Chapter and verse numbers
- Text of the verse

1. To import the records into MySQL, create a table named kjv that looks like this:

```
CREATE TABLE kjv
(
bsect ENUM('O','N') NOT NULL, # book section (testament)
bname VARCHAR(20) NOT NULL, # book name
bnum TINYINT UNSIGNED NOT NULL, # book number
cnum TINYINT UNSIGNED NOT NULL, # chapter number
vnum TINYINT UNSIGNED NOT NULL, # verse number
vtext TEXT NOT NULL, # text of verse
FULLTEXT (vtext) # full-text index
) ENGINE = InnoDB;
```

2. After creating the kjv table, load the kjv.txt file into it.

3 Using Full-Text Searches

1. To perform a search using the `FULLTEXT` index, use `MATCH()` to name the indexed column and `AGAINST()` to specify what text to look for. For example, you might wonder, "How many times does the name Hadoram occur?" To answer that question, search the vtext column using this statement:

```
mysql> SELECT COUNT(*) from kjv WHERE MATCH(vtext) AGAINST('Hadoram');
```

The `MATCH()` function performs a natural language search for a string against a text collection. A collection is a set of one or more columns included in a `FULLTEXT` index. The search string is given as the argument to `AGAINST()`. For each row in the table, `MATCH()` returns a relevance value; that is, a similarity measure between the search string and the text in that row in the columns named in the `MATCH()` list. When `MATCH()` is used in a WHERE clause, as in the example shown earlier, the rows returned are automatically sorted with the highest relevance first.

2. Read the information about the relevance function used by MySQL here:

   http://dev.mysql.com/doc/refman/5.7/en/fulltext-boolean.html

3. Find out what those verses are and display their relevance. Tips: to improve readability you can truncate the vtext column (using the command LEFT) and use \G at the end of your select so the results better fit the page. You should get something like this:

```
*************************** 1. row ***************************
bname: Genesis
cnum: 10
vnum: 27
vtext: And Hadoram, and Uzal, and Diklah,
score: 8.65983963012695
*************************** 2. row ***************************
bname: 1 Chronicles
cnum: 1
vnum: 21
vtext: Hadoram also, and Uzal, and Diklah,
score: 8.65983963012695
*************************** 3. row ***************************
bname: 1 Chronicles
cnum: 18
vnum: 10
vtext: He sent Hadoram his son to king David, to inquire of his welfare,...
score: 7.35226821899414
*************************** 4. row ***************************
bname: 2 Chronicles
cnum: 10
vnum: 18
vtext: Then king Rehoboam sent Hadoram that was over the tribute; and th...
score: 7.02105093002319
```

4. Run the following queries:

```
SELECT COUNT(*) from kjv WHERE MATCH(vtext) AGAINST('Abraham');
```

```
mysql> SELECT COUNT(*) from kjv
    -> WHERE MATCH(vtext) AGAINST('Abraham Sarah');
```

```
mysql> SELECT COUNT(*) from kjv
    -> WHERE MATCH(vtext) AGAINST('Abraham Sarah Ishmael Isaac');
```

   Adding more words to your query makes it more or less specific? Is the query performing and AND or OR search for any of the words?

5. Write a query that searches for rows with the words "Abraham" and "Sarah". (Solution: There are 19).

4 Boolean Full-Text Searches

MySQL can perform boolean full-text searches using the `IN BOOLEAN MODE` modifier. With this modifier, certain characters have special meaning at the beginning or end of words in the search string. The boolean full-text search capability supports among others the following operators:

- +

  A leading plus sign indicates that this word must be present in each row that is returned.

- -

  A leading minus sign indicates that this word must not be present in any of the rows that are returned.

- @distance

  This operator tests whether two or more words all start within a specified distance from each other, measured in words. Specify the search words within a double-quoted string immediately before the @distance operator, for example, MATCH(col1) AGAINST('"word1 word2 word3" @8' `IN BOOLEAN MODE`)

- *

  The asterisk serves as the truncation (or wildcard) operator. Unlike the other operators, it is appended to the word to be affected. Words match if they begin with the word preceding the * operator.

  The wildcarded word is considered as a prefix that must be present at the start of one or more words. If the minimum word length is 4, a search for '+word +the*' could return fewer rows than a search for '+word +the', because the second query ignores the too-short search term the.

- "

  A phrase that is enclosed within double quote (") characters matches only rows that contain the phrase literally, as it was typed. The full-text engine splits the phrase into words and performs a search in the `FULLTEXT` index for the words. Non-word characters need not be matched exactly; phrase searching requires only that matches contain exactly the same words as the phrase and in the same order. For example, "test phrase" matches "test, phrase".

  If the phrase contains no words that are in the index, the result is empty. The words might not be in the index because of a combination of factors: if they do not exist in the text, are stopwords, or are shorter than the minimum length of indexed words.

Using these operators write statements that:

1. Find the number of rows that contain either of the names David or Goliath (Solution 898)
2. Find the number of rows that contain the names David and Goliath (Solution 2)
3. Find the number of rows containing the name David but not Goliath (Solution 892)
4. Find the number of rows starting by whirl (Solution 28)
5. Find the number of rows containing the sentence still small voice (Solution 1)

5 Full-Text Searches with Query Expansion

In some cases, users want to search for information based on the knowledge that they have. Users use their knowledge to define keywords to search for information, and typically those keywords are too short. To help users to find what they want based on the too-short keywords, MySQL full-text search engine introduces a concept called query expansion.
The query expansion is used to widen the search result of the full-text searches based on automatic relevance feedback (or blind query expansion). Technically, MySQL full-text search engine performs the following steps when the query expansion is used:

- First, MySQL full-text search engine looks for all rows that match the search query.
- Second, it checks all rows in the search result and finds the relevant words.
- Third, it performs a search again but based on the relevant words instead of the original keywords provided by the users.

From the application perspective, you can use the query expansion when the search results are too few. You perform the searches again but with query expansion to offer users more information that are related and relevant to what they are looking for.
To use the query expansion, you use the `WITH QUERY EXPANSION` search modifier in the `AGAINST()` function.

1. Find the rows that contain the name Gopher (Solution 1 row)
2. Use query expansion to obtain more results related to Gopher. (To abort the query if it takes too long you can press Crtl+c)

Notice that blind query expansion tends to increase noise significantly by returning non-relevant results. It is highly recommended that you use query expansion only when the searched keyword is short.

6 Personal Study

Read the information about full-text searches in MySQL here:

`http://dev.mysql.com/doc/refman/5.7/en/fulltext-search.html`

Note the built-in MySQL full-text parser uses the white space between words as a delimiter to determine where words begin and end, which is a limitation when working with ideographic languages that do not use word delimiters. To address this limitation, MySQL provides an n-gram full-text parser that supports Chinese, Japanese, and Korean (CJK):

`http://dev.mysql.com/doc/refman/5.7/en/fulltext-search-ngram.html`