10.7 Full-Text Search

MySQL uses Ranking with Vector Spaces for ordinary full-text queries.

Rank, also known as relevance rank, also known as relevance measure, is a number that tells us how good a match is.

Vector Space, which MySQL sometimes calls "natural language", is a well-known system based on a metaphor of lines that stretch in different dimensions (one dimension per term) for varying distances (one distance unit per occurrence of term). The value of thinking of it this way is: once you realize that term occurrences are lines in a multi-dimensional space, you can apply basic trigonometry to calculate "distances", and those distances are equatable with similarity measurements. A comprehensible discussion of vector space technology is here: https://en.wikipedia.org/wiki/Vector_space_model. And a text which partly inspired our original developer is here: ftp://ftp.cs.cornell.edu/pub/smart /smart.11.0.tar.Z ("SMART").

But let's try to describe the classic formula:

```
1  w = tf * idf
```

This means "weight equals term frequency times inverse of document frequency", or "increase weight for number of times term appears in one document, decrease weight for number of documents the term appears in". (For historical reasons we're using the word "weight" instead of "distance", and we're using the information-retrieval word "document" throughout; when you see it, think of "the indexed part of the row".)

For example: if "rain" appears three times in row #5, weight goes up; but if "rain" also appears in 1000 other documents, weight goes down.

MySQL uses a variant of the classic formula, and adds on some calculations for "the normalization factor". In the end, MySQL's formula looks something like:

```
1  w = (log(dtf)+1)/sumdtf * U/(1+0.0115*U) * log((N-nf)/nf)
```

Where:

```
dtf     is the number of times the term appears in the document
sumdtf  is the sum of (log(dtf)+1)'s for all terms in the same document
U       is the number of Unique terms in the document
N       is the total number of documents
nf      is the number of documents that contain the term
```

The formula has three parts: base part, normalization factor, global multiplier.

The base part is the left of the formula, "(log(dtf)+1)/sumdtf".

The normalization factor is the middle part of the formula. The idea of normalization is: if a document is shorter than average length then weight goes up, if it's average length then weight stays the same, if it's longer than average length then weight goes down. We're using a pivoted unique normalization factor. For the theory and justification, see the paper "Pivoted Document Length Normalization" by Amit Singhal and Chris Buckley and Mandar Mitra ACM SIGIR'96, 21-29, 1996: http://ir.iit.edu/~dagr/cs529/files/handouts/singhal96pivoted.pdf. The word "unique" here means that our measure of document length is based on the unique terms in the document. We chose 0.0115 as the pivot value, it's PIVOT_VAL in the MySQL source code header file myisam/ftdefs.h.

If we multiply the base part times the normalization factor, we have the term weight. The term weight is what MySQL stores in the index.

The global multiplier is the final part of the formula. In the classic Vector Space formula, the final part would be the inverse document frequency, or simply

```
log(N/nf)
```

We have replaced it with

```
log((N-nf)/nf)
```

This variant is more often used in "probabilistic" formulas. Such formulas try to make a better guess of the probability that a term will be relevant. To go back to the old system, look in myisam/ftdefs.h for "#define GWS_IN_USE GWS_PROB" (that is, global weights by probability) and change it to "#define GWS_IN_USE GWS_IDF" (that is, global weights by inverse document frequency).

Then, when retrieving, the rank is the product of the weight and the frequency of the word in the query:

```
R = w * qf;
```

Where:

```
w      is the weight (as always)
qf     is the number of times the term appears in the query
```

In vector-space speak, the similarity is the product of the vectors.

And R is the floating-point number that you see if you say: SELECT MATCH(...) AGAINST (...) FROM t.

To sum it up, w, which stands for weight, goes up if the term occurs more often in a row, goes down if the term occurs in many rows, goes up / down depending whether the number of unique words in a row is fewer / more than average. Then R, which stands for either Rank or Relevance, is w times the frequency of the term in the AGAINST expression.

The Simplest Possible Example

First, make a fulltext index. Follow the instructions in the "MySQL Full-Text Functions" section of the MySQL Reference Manual. Succinctly, the statements are:

```
CREATE TABLE articles (
  id INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY KEY,
  title VARCHAR(200),
  body TEXT,
  FULLTEXT (title,body) );
INSERT INTO articles (title,body) VALUES
  ('MySQL Tutorial','DBMS stands for DataBase ...'),
  ('How To Use MySQL Well','After you went through a ...'),
  ('Optimizing MySQL','In this tutorial we will show ...'),
  ('1001 MySQL Tricks','1. Never run mysqld as root. 2. ...'),
  ('MySQL vs. YourSQL','In the following database comparison ...'),
  ('MySQL Security','When configured properly, MySQL ...');
```

Now, let's look at the index.

There's a utility for looking at the fulltext index keys and their weights. The source code is myisam/myisam_ftdump.c, and the executable comes with the binary distribution. So, if exedir is where the executable is, and datadir is the directory name that you get with "SHOW VARIABLES LIKE 'datadir%'", and dbname is the name of the database that contains the articles table, then this works:

```
>/exedir/myisam_ftdump /datadir/dbname/articles 1 -d
       b8            0.9456265 1001
       f8            0.9560229 comparison
      140            0.8148246 configured
        0            0.9456265 database
       f8            0.9560229 database
        0            0.9456265 dbms
        0            0.9456265 mysql
       38            0.9886308 mysql
       78            0.9560229 mysql
       b8            0.9456265 mysql
       f8            0.9560229 mysql
      140            1.3796179 mysql
       b8            0.9456265 mysqld
       78            0.9560229 optimizing
      140            0.8148246 properly
       b8            0.9456265 root
      140            0.8148246 security
       78            0.9560229 show
        0            0.9456265 stands
       b8            0.9456265 tricks
        0            0.9456265 tutorial
       78            0.9560229 tutorial
       f8            0.9560229 yoursql
```

Let's see how one of these numbers relates to the formula.

The term 'tutorial' appears in document 0. The full document is "MySQL Tutorial / DBMS stands for DataBase ...". The word "tutorial" appears once in the document, so dtf = The word "for" is a stopword, so there are only 5 unique terms in the document ("mysql", "tutorial", "dbms", "stands", "database"), so U = Each of these terms appears once in the document, so sumdtf is the sum of log(1)+1, five times. So, taking the first two parts of the formula (the term weight), we have:

```
(log(dtf)+1)/sumdtf * U/(1+0.0115*U)
```

which is

```
(log(1)+1)/( (log(1)+1)*5) * 5/(1+0.0115*5)
```

which is

```
0.9456265
```

which is what myisam_ftdump says. So the term weight looks good.

Now, what about the global multiplier? Well, myisam_ftdump could calculate it, but you'll see it with the mysql client. The total number of rows in the articles table is 6, so N = And "tutorial" occurs in two rows, in row 0 and in row 78, so nf = So, taking the final (global multiplier) part of the formula, we have:

```
log((N-nf)/nf)
```

which is

```
log((6-2)/2)
```

which is

```
0.6931472
```

So what would we get for row 0 with a search for 'tutorial'? Well, first we want w, so: Multiply the term weight of tutorial (which is 0.9456265) times the global multiplier (which is 0.6931472). Then we want R, so: Multiply w times the number of times that the word 'tutorial' appears in the search (which is 1). In other words, R = 0.9456265 * 0.6931472 * 1. Here's the proof:

```
mysql> select round(0.9456265 * 0.6931472 * 1, 7) as R;
+-----------+
| R         |
+-----------+
| 0.6554583 |
+-----------+
1 row in set (0.00 sec)

mysql> select round(match(title,body) against ('tutorial'), 7) as R
    -> from articles limit 1;
+-----------+
| R         |
+-----------+
| 0.6554583 |
+-----------+
1 row in set (0.00 sec)
```

You'll need memory

The MySQL experience is that many users appreciate the full-text precision or recall, that is, the rows that MySQL returns are relevant and the rows that MySQL misses are rare, in the judgment of some real people. That means that the weighting formula is probably justifiable for most occasions. Since it's the product of lengthy academic research, that's understandable.

On the other hand, there are occasional complaints about speed. Here, the tricky part is that the formula depends on global factors -- specifically N (the number of documents) and nf (the number of documents that contain the term). Every time that insert/update/delete occurs for any row in the table, these global weight factors change for all rows in the table.

If MySQL was a search engine and there was no need to update in real time, this tricky part wouldn't matter. With occasional batch runs that redo the whole index, the global factors can be stored in the index. Search speed declines as the number of rows increases, but search engines work.

However, MySQL is a DBMS. So when updates happen, users expect the results to be visible immediately. It would take too long to replace the weights for all keys in the fulltext index, for every single update/insert/delete. So MySQL only stores the local factors in the index. The global factors are more dynamic. So MySQL stores an in-memory binary tree of the keys. Using this tree, MySQL can calculate the count of matching rows with reasonable speed. But speed declines logarithmically as the number of terms increases.

Weighting in boolean mode

The basic idea is as follows: In an expression of the form `A or B or (C and D and E)`, either `A` or `B` alone is enough to match the whole expression, whereas `C`, `D`, and `E` should ***all*** match. So it's reasonable to assign weight 1 to each of `A`, `B`, and `(C and D and E)`. Furthermore, `C`, `D`, and `E` each should get a weight of 1/3.

Things become more complicated when considering boolean operators, as used in MySQL full-text boolean searching. Obviously, `+A +B` should be treated as `A and B`, and `A B` - as `A or B`. The problem is that `+A B` can ***not*** be rewritten in and/or terms (that's the reason why thisextendedset of operators was chosen). Still, approximations can be used. `+A B C` can be approximated as `A or (A and (B or C))` or as `A or (A and B) or (A and C) or (A and B and C)`. Applying the above logic (and omitting mathematical transformations and normalization) one gets that for `+A_1 +A_2 ... +A_N B_1 B_2 ... B_M` the weights should be: `A_i = N`, `B_j=1` if `N==0`, and, otherwise, in the first rewriting approach `B_j = B_j = (1+(M-1)*2^M)/(M*(2^(M+1)-1))`.

The second expression gives a somewhat steeper increase in total weight as number of matched `B_j` values increases, because it assigns higher weights to individual `B_j` values. Also, the first expression is much simpler, so it is the first one that is implemented in MySQL.

© 2017, Oracle Corporation and/or its affiliates