Int. J. of Computers, Communications & Control, ISSN 1841-9836, E-ISSN 1841-9844
Vol. V (2010), No. 5, pp. 735-743

Full-Text Search Engine using MySQL

C. Gyorodi, R. Gyorodi, G. Pecherle, G. M. Cornea

Cornelia Gyorodi, Robert Gyorodi, George Pecherle, George Mihai Cornea
Department of Computer Science
Faculty of Electrical Engineering and Information Technology
University of Oradea, Str. Universitatii 1, 410087, Oradea, Romania
E-mail: cgyorodi@uoradea.ro, rgyorodi@rdsor.ro, gpecherle@uoradea.ro, generalmip@yahoo.com

Abstract: In this article we will try to explain how we can create a search engine using the powerful MySQL full-text search. The ever increasing demands of the web requires cheap and elaborate search options. One of the most important issues for a search engine is to have the capacity to order its results set as relevance and provide the user with suggestions in the case of a spelling mistake or a small result set. In order to fulfill this request we thought about using the powerful MySQL full-text search. This option is suitable for small to medium scale websites. In order to provide sound like capabilities, a second table containing a bag of words from the main table together with the corresponding metaphone is created. When a suggestion is needed, this table is interrogated for the metaphone of the searched word and the result set is computed resulting a suggestion.
Keywords: full-text, search, MySQL, index, search engine, ranking, metaphone, Levenstein

1 Introduction

Before the advent of the search engine, users had to search manually through dozens or hundreds of articles to find the ones that were right for them. Nowadays, in our more user-centered world, we expect the results to come to the user, not the other way around. The search engine gets the computer to do the work for the user.

Full-text search is widely used for various services of the Internet. A high-speed and a more efficient full-text search technology are necessary because of the amount of increasing handled document and corresponding document data every day [6].

According to the MySQL manual, full-text is a "natural language search"; the words are indexed and appear to represent the row, using the columns you specified. As an example, if all your rows contain "MySQL" then "MySQL" won't match much. It's not terribly unique, and it would return too many results. However, if "MySQL" were present in only 5% of the rows, it would return those rows because it doesn't appear too often to be known as a keyword that's very used.

MySQL full-text search doesn't have too many user-tunable parameters. If you want more control over your search you will have to download the sources and compile them yourself after you've made the changes you wanted. Anyway, MySQL full-text is tuned for best effectiveness. Modifying the default behaviour in most cases can actually decrease effectiveness [1].

Our implementation comes to bring a plus of functionality to the basic search capabilities that MySQL offers by returning better quality results to the user. For small data sets to be searched, its performance can be easily compared with the one of the more advanced dedicated full-text search engines.

2 Full-Text Search

In text retrieval, full-text search refers to a technique for searching a computer-stored document or database. In a full-text search, the search engine examines all of the words in every stored document as it tries to match search words supplied by the user.

When dealing with a small number of documents it is possible for the full-text search engine to directly scan the contents of the documents with each query, a strategy called serial scanning. This is what some rudimentary tools, such as grep, do when searching.

However, when the number of documents to search is potentially large or the quantity of search queries to perform is substantial, the problem of full-text search is often divided into two tasks: indexing and searching. The indexing stage will scan the text of all the documents and build a list of search terms, often called an index, but more correctly named a concordance. In the search stage, when performing a specific query, only the index is referenced rather than the text of the original documents. Some indexers also employ language-specific stemming on the words being indexed, so for example any of the words "drives", "drove", or "driven" will be recorded in the index under a single concept word "drive" [2].

Before passing on to the search, we would like to talk a little about how MySQL full-text indexes work. A MySQL full-text query returns rows according to relevance. But what is relevance? It is a floating-point number based on formulas. Researchers have shown that these formulas produce results that real users want.

For every word that isn't too short or isn't a too common one, MySQL calculates a number to determine its relevance inside the text. To be noticed that MySQL will not increase weight if two keywords are close to each other. Local weight, global weight, and query weight are the only things that matter. MySQL can work with stemming but as we've seen from different tests it isn't working very well, usually bombarding the user with tons of irrelevant results.

There are three formulas we have to mention for full-text index [3].
local weight = (log(dtf)+1)/sumdtf * U / (1+0.0115*U)
global weight = log((N-nf)/nf)
query weight = local weight * global weight * qf

where:
dtf - How many times the term appears in the row
sumdtf -The sum of "(log(dtf)+1)" for all terms in the same row
U - How many unique terms are in the row
N - How many rows are in the table
nf - How many rows contain the term
qf - How many times the term appears in the query

Notice that local weight depends on a straight multiplier, the term-within-row frequency times the unique frequency.

But most simply: If a term appears many times in a row, the weight goes up.

Why does local weight depend on how many times the term is in the row? Think of the document you are reading now. I inevitably mention "MySQL" and "full-text" several times. That's typical: if words appear several times, they are likely to be relevant.

Notice that global weight depends on an inverse multiplier, the count of rows MINUS the count of rows that the term appears in. Put most simply: If a term appears in many rows, the weight goes down.

To illustrate this better we've chosen to give a practical example. For this example, we've used a smaller database in order to be able to calculate the relevancies. To see what is in a full-

text index, we can use the myisam_ftdump program. It comes with the standard distribution. The name of owner table is demo and it is located in dmp database. The table consists in 2 columns, the second one being the one with full-text index.

In order to see the local weights of the different words we have to use the command dump index. This command will return the data offsets and the word weights (Fig. 1).

```
C:\wamp\bin\mysql\mysql5.1.30\bin>myisam_ftdump c:\wamp\bin\mysql\mysql5.1.30\data\dmp\demo 1 -d
```

88	0.9560229 create	c0	0.9560229 mysql
0	0.8421108 database	0	0.8421108 open
88	0.9560229 engine	0	0.8421108 popular
44	0.9456265 feature	44	0.9456265 search
0	0.8421108 free	88	0.9560229 search
44	0.9456265 fultext	c0	0.9560229 search
c0	0.9560229 fultext	c0	0.9560229 slow
44	0.9456265 grate	0	0.8421108 source
44	0.9456265 indexing	0	0.8421108 world
0	1.4258175 mysql *		
88	0.9560229 mysql		

Figure 1: Dump index

In order to see the global weights of the words we have to use the command calculate per-word stats. This command will return for us a word count and the global weight (Fig. 2).

```
C:\wamp\bin\mysql\mysql5.1.30\bin>myisam_ftdump c:\wamp\bin\mysql\mysql5.1.30\data\dmp\demo 1 –c
```

1	1.0986123 create	3	-1.0986123 mysql **
1	1.0986123 database	1	1.0986123 open
1	1.0986123 engine	1	1.0986123 popular
1	1.0986123 feature	3	-1.0986123 search
1	1.0986123 free	1	1.0986123 slow
2	0.0000000 fultext	1	1.0986123 source
1	1.0986123 grate	1	1.0986123 world
1	1.0986123 indexing		

Figure 2: Calculate per-word stats

For the first formula: (log(dtf)+1)/sumdtf * U/(1+0.0115*U);
where Dtf is how many times the term appears in the row
"MySQL" appears 2 times in row 0
so log(dtf()+1) = 0.6931472 + 1 = 1.6931472
sumdtf - The sum of "(log(dtf)+1)" for all terms in the same row

"MySQL" appears 2 times in row 0, so add log(2)+1
"world" appears 1 times in row 0, so add log(1)+1
"popular" appears 1 times in row 0, so add log(1)+1
"open" appears 1 times in row 0, so add log(1)+1
"source" appears 1 times in row 0, so add log(1)+1
"database" appears 1 times in row 0, so add log(1)+1
"free " appears 1 times in row 0, so add log(1)+1

so sumdtf = log(2)+1 + (log(1)+1)*6 = 7.6931472

U - How many unique terms are in the row there are 7 unique terms in row 0
so U/(1+0.115*U) = 7/(1+0.0115*7) = 6.478482

local weight = 1.6931472 / 7.6931472 * 6.478482= 1.4258175 (check Figure 2 for the first occurrence of the term “MySQL” *)

For the second formula: log((N-nf)/nf);
where:
N - How many rows are in the table; there are 4 rows in the ’demo’ table
nf - How many rows contain the term; the term "special" occurs in 3 rows

log((4-3)/3)= -1.0986123 (check Fig. 3 for the term “MySQL” **)
Note that because this term appears in more than 50% of the rows it has a negative global weight. In an actual search, this term will practically be ignored, only the adjacent terms being representative.

3 Implementation

The primary table is the table where we keep all the data that has to be searched and ranked. Also, from this table we will create later the “bag of words” together with their metaphones in order to provide correction suggestions to the user. The primary table will consist of 6 columns, the first one being the row id and the others all being meant for full-text search. The structure of the table is:

ID - unique identifier for every row
URL - the URL where the text was taken from
TITLE - the title of the page the text was taken from
CONTENT - everything on the page that is not part of the formatting and functionality (only text, no HTML or other scripts, including the dealers and strong text)
HEADERS - H1,... contents from the page. They will have higher relevancy than STRONG and CONTENT but lower than URL and TITLE
STRONG - words that appear between B or STRONG tags. They will have higher relevancy than ordinary text from CONTENT but lower than all the others.

During the construction of the table in the indexing process we decompose every page to obtain the required fields. First we will obtain the URL and the TITLE fields. After we have those two we can get rid off the head part of the web page and move on to the body. Before parsing out the entire HTML and other script tags from the text, we will have to take out the header and the strong keywords that will have a higher relevancy in our future search. After we’ve done that we can parse all the remaining script tags and send the resulting data into the MySQL database.

An improvement to this table would be a new column that contains all the keywords linking to this page. This way you can extend the article relevancy behind the actual page. Those linking keywords can be considered the ones between the anchor tag or the most relevant keywords from that page (calculated by their density after we’ve previously removed the stop words) or a combination of both. However, this approach requires many more pages to be scanned and many of them could come from outside our website.

After the first indexing is complete we can create the full-text index on the table. From now on we can make full-text searches on that table. Any new insert into the database will

be attached on the fly internally by MySQL so that we don't have to worry about this [9]. The SQL syntax for doing this is: ALTER TABLE 'content' ADD FULLTEXT ('url','title', 'content','headers','strong')

Searching the database is not very hard. Actually we will only use a query to apply a search on the database and let MySQL do the rest. It is a lot harder to determine the best values for the weights of the results. Depending on those constants, we will have different search results as we will show in the following examples.

In order to query the database we will use the following query. The query is virtually composed of three parts. The first part is meant to determine the relevancies of every column in part, the second one is meant for determining the matching rows and the last one is for ordering. You can see the query in Fig. 3.

```
SELECT * ,
MATCH (`url`) AGAINST ('search term') AS relUrl,
MATCH (`title` ) AGAINST ('search term') AS relTitle,
MATCH (`content` ) AGAINST ('search term') AS relContent,
MATCH (`headers` ) AGAINST ('search term') AS relHeaders,
MATCH (`strong` ) AGAINST ('search term') AS relStrong
FROM `content`
WHERE MATCH (`content` , `url` , `title` , `headers` , `strong`) AGAINST ('search term')
ORDER BY relUrl *U + relTitle *T + relContent *C+ relHeaders *H+ relStrong *S DESC
```

Figure 3: Search query

relUrl, relTitle, relContent, relHeaders and relStrong are the different relevancies returned by MySQL after the preliminary search of every different column in part. The parameters U, T, C, H and S represent the importance of those relevancies in part.

The query will execute like this: first, different relevancies will be returned from the search on the individual columns. Those relevancies will be later used to calculate the order of the results. After we have got the relevancies we have to move on to the retrieval of all the rows that contain the terms that we searched. In order to do this we have to select all the matches inside the text fields ('content', 'url', 'title', 'headers', 'strong'). Once we have those results we will have to reorder them by relevance. In order to do this we will determine the ORDER BY argument using the following formula.

$$\text{relevance} = \text{relUrl} *U + \text{relTitle} *T + \text{relContent} *C + \text{relHeaders} *H + \text{relStrong} *S$$

Figure 4: Relevance formula

The relevance for every column should be carefully selected in order to achieve the best result order. The URL relevance is not so important in our opinion. We have added it because all the big search engines have it. We will treat it as it would have the same relevance as the page title. The next column on the relevance scale, after the URL and the Title would be the Headers relevance.

Let's think about a certainty search. We want to search for an article about "Mysql Fulltext". The script will first determine the pages that contain the keywords inside the Title and URL of the page because those are the most suitable for us. The next in importance is the Headers column.

The last two of the columns are the Content column and the Strong column. The Strong column contains keywords that the writer of that page thought they are of a higher importance, with strong relevance to the subject.

The order of the relevancies is clear. The plain content relevance is the lower one, followed by the keywords relevance (Strong), Headers, Title which is about the same to the URL relevance. The problems appear when we have to determine the best magnitude for those relevance parameters.

If the difference of the relevance parameters is too small, we will not have effective search results ordering. But if we fall into the other side, we can provide excessive ordering based only on the hierarchy we've determined for the parameters and less based on the relevancies of the results.

A schematisation of the proposed indexing and search algorithm is provided below:
A) Indexing
1. The web page is processed by reading its contents.
2. The internal links from the page are determined.
3. The unique internal links and full page contents are inserted in a 'pages' table.
4. The next link to follow is taken from the 'pages' table and the new corresponding page is processed by going to step 1, until there are no more unprocessed pages.
5. The 'pages' table is processed to determine the page title, headings and strong text and the results are placed in a new 'contents' table.

B) Full-Text Search
1. Query the 'contents' table against a search term.
2. Determine the relevancies of every column in part.
3. Determine the matching rows.
4. Order the results by relevance, calculated using the formula in Fig. 4. The parameters U, T, C, H and S are chosen by us and they are an indicator of the importance of each of the relevancies (URL, Title, Content, Headers and Strong Text).

4 Testing

To test the algorithm, we have set up a database of 320 MB (including data and index) and a total number of 11,670 records. The database was built by indexing a subset of the wikipedia.com website, using the indexing algorithm described in the previous section.

For testing purposes, we have chosen the following values for the U, T, C, H and S parameters. The user can set his own values. A higher value means a higher importance of that element:
U=1.14, T=1.14, C=1, H=1.3, S=1.2

The testing process was done by asking different users to give ratings to each of the search results. We have performed a total of 143 tests. The ratings given by the users proved that our algorithm is more relevant than the default MySQL method by 19.47% .

Here are some examples of tests done with specific keywords:
Keyword: "programming"
Results: Our algorithm (the left panel) returned relevant results in positions 1 and 2 (an article titled "Computer Programming" and another one "Application programming interface"). This was because our algorithm assigned a higher importance to the page title and URL (1.14) than the page contents (1). The right panel did not return many relevant results (a result which was close to our expectations was in position 2, but it was not very specific because it was about a specific programming language and not about programming languages in general).

Keyword: "universal serial bus"
Results: Our algorithm (the left panel) returned a relevant result in position 1 (an article titled

"Universal Serial Bus"). The right panel did not return any relevant results (instead it returned pages that contain "universal" only and not the whole search term).

5 Search Suggestions

In many cases, there appears the necessity to correct the spelling mistakes made by the users when they don't know exactly how to spell something or are just making a typing mistake. To do this, we will have to give them the closest correct form that best matches the initial search.

Metaphone is a phonetic algorithm, an algorithm published in 1990 for indexing words by their English pronunciation. The algorithm produces variable length keys as its output, as opposed to Soundex's fixed-length keys. Similar sounding words share the same keys.

Metaphone was developed by Lawrence Philips as a response to deficiencies in the Soundex algorithm. It is more accurate than Soundex because it uses a larger set of rules for English pronunciation. Metaphone is available as a built-in operator in a number of systems, including later versions of PHP. The original author later produced a new version of the algorithm, which he named Double Metaphone, that produces more accurate results than the original algorithm. However I will use the first version of the script because it returns accurate enough keys with a much better performance than the second one. [4]

In order to provide fast enough processing for the suggestions we will create a "bag of words" containing all the distinct words contained inside the content column. For each one of them we will attach the metaphone key. When a search is done we will select the candidates by calculating the metaphone of every word inside the search string and select the matches inside the metaphone table. After that operation we will have about 3 to 8 possible words to choose from.

The Levenshtein distance is defined as the minimal number of characters you have to replace, insert or delete to transform str1 into str2 . The complexity of the algorithm is O(m*n), where n and m are the length of str1 and str2 (rather good when compared to similar_text(), which is O(max(n, m)**3), but still expensive).

In its simplest form the function will take only the two strings as parameters and will calculate just the number of insert, replace and delete operations needed to transform str1 into str2. [5]

We will use this algorithm to determine the closest form from the list of candidates by selecting the candidate with the smallest Levenshtein distance to the searched keyword. To be noticed that this algorithm only works for word spelling mistakes. It does not work for mistakes like missing space, or composed words.

Missing space example: "MySQLFultext" will not suggest "MySQL Fultext"
Composed words example: "Pine apple" will not suggest "Pineapple"

Although some of the suggestion results can be hidden from the users because of the use of metaphone equivalence, this approach gives an incredible boost of performance. With a suggestion table containing over 130,000 rows it would take way too long to determine the Levensthein distance between the searched form and the correct form regarding the fact that, as we have said before, the Levenshtein algorithm has a complexity O(m*n). In owner case we would have a complexity O(m*n)*NrRows, where NrRows exceeds 130,000. Also, by applying this direct approach, we wouldn't solve the problems we've illustrated above.

By applying owner algorithm we can reduce the candidates from 130,000 to approximately 4-7 words. This represents a significant improvement of performance with only little disadvantages. Furthermore, those disadvantages could be counteracted by completing the suggestion table according to the newly searched term if the suggestion we provided is wrong and if there

is a sufficient similarity between the previous and the actual search, similarity which can be determined by dividing the Levenshtein distance with the string length.

A schematisation of the proposed search suggestions algorithm is provided below:
1. Create a table containing all the words in the contents column and their attached metaphone keys.
2. The metaphone of the searched keyword is determined using the metaphone function [4].
3. The Levenshtein distance is calculated as in [5].
4. The word with the smallest Levenshtein distance is chosen.

6 Comparison with other similar technologies

In comparison with other similar technologies we have strong points and weak points at the same time. It is impossible to have all of them at the same time and there has to be a compromise between costs, speed, maintenance and quality of the results.

In comparison with MySQL basic full-text search we have the advantage of a better ordered result set. However, this comes with the price of an increased search time because of the extra computation needed for the relevance ordering, according to Fig. 4. Both methods have the advantages of being free, with easy maintenance (the update of the full-text index is done on the fly) and they don't require special permissions on the server.

If we compare it with the free open-source SQL full-text search engine Sphinx, we can mention other differences. The common part of the two solutions is that they are both free. The strong points of our solution are that we have a better ordered result set and easier maintainability. The strong point for Sphinx is the search speed making it suitable for searching bigger databases. Another week point for Sphinx is the maintainability. The full-text index is not able to be updated on the fly so that it has to be recomputed periodically, task that can take from 15 minutes for small databases to a few hours for large ones.

7 Conclusions

By using our implementation it is possible to offer a complete and powerful search option. For the future, we would like to improve the search speed by using a dedicated full-text search engine.

Another improvement we would like to implement would be a new parameter which will determine the popularity of a result based on determining if a user found what he wanted or not. This will use a combination of the time the user spent on that certain page divided by the content length, how many times the user returned to this page or if a page is or is not a stop page (which can indicate that the user found what he wanted to know).

Also, in addition to search, we would also like to implement high-speed insert and delete, allowing full-text search to be used in the same way as other types of database search in which data can be searched right after data is inserted [7].

Another idea for future improvement is handling scientific document searches, especially mathematical text and mathematical operations [8].

For the suggestion tool we want to improve the algorithm in such a way to be able to correct as many mistakes as possible.

Bibliography

[1] Fine-Tuning MySQL Full-Text Search - http://dev.mysql.com/doc/refman/5.0/en/fulltext-fine-tuning.html

[2] Full text search - by Wikipedia - http://en.wikipedia.org/wiki/Full_text_search

[3] MySQL's Full-Text Formulas - by Database Journals - `http://www.databasejournal.com/features/mysql/article.php/3512461/MySQLs-Full-Text-Formulas.htm`

[4] Metaphone - by Wikipedia - http://en.wikipedia.org/wiki/Metaphone

[5] The Levenstein distance - http://us2.php.net/levenshtein

[6] Atlam, E.-S., Ghada, E.-M., Fuketa, M., Morita, K., Aoe, J., A compact memory space of dynamic full-text search using Bi-gram index, Computers and Communications, 2004. Proceedings ISCC 2004. Ninth International Symposium

[7] Ikeda, T., Mano, H., Itoh, H., Takegawa, H., Hiraoka, T., Horibe, S., Ogawa, Y., "TRMeister: a DBMS with high-performance full-text search functions", Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference

[8] Misutka, J., Galambos, L., "Mathematical Extension of Full Text Search Engine Indexer", Information and Communication Technologies: From Theory to Applications, 2008. ICTTA 2008. 3rd International Conference

[9] D. Zmaranda, G. Gabor, Issues on Optimality Criteria Applied in Real-Time Scheduling, *International Journal of Computers Communications & Control*, ISSN 1841-9836, Suppl.S, 3(S):536-540, 2008.