Linnæus University
School of Computer Science, Physics and Mathematics

G1-Uppsats

Webbprogrammerare

Matching in MySQL

A comparison between REGEXP and LIKE

Author: Emil Carlsson
Supervisor: Morgan Ericsson
Semester: Spring 2012
Course code: 1DV40E

Abstract

When needing to search for data in multiple datasets there is a risk that not all datasets are of the same type. Some might be in XML-format; others might use a relational database. This could frighten developers from using two separate datasets to search for the data in, because of the fact that crafting different search methods for different datasets can be time consuming.

One option that is greatly overlooked is the usage of regular expressions. If a search expression is created it can be used in a majority of database engines as a "WHERE" statement and also in other form of data sources such as XML.

This option is however, at best, poorly documented and few tests have been made in how it performs against traditional search methods in databases such as "LIKE".

Multiple experiments comparing "LIKE" and "REGEXP" in MySQL have been performed for this paper. The results of these experiments show that the possible overhead by using regular expressions can be motivated when considering the gain of only using one search phrase over several data sources.

Abstrakt

När behovet att söka over flertalet typer av datakällor finns det alltid en risk att inte alla datakällor är av samma typ. Några kan vara i XML-format; andra kan vara i form av en relationsdatabas. Detta kan avskräcka utvecklare ifrån att använda två oberoende datakällor för att söka efter data, detta för att det kan vara väldigt tidskrävande att utveckla två olika vis att skapa sökmetoderna.

Ett alternativ som ofta är förbisett är att använda sig av reguljära uttryck. Om ett sökuttryck är skapat i reguljära uttryck så kan det användas i en majoritet av databasmotorerna på marknaden som ett ”WHERE” påstående, men det kan även användas i andra typer av datakällor så som XML.

Detta alternativ är allt som ofta dåligt dokumenterat och väldigt få tester har utförts på prestandan i jämförelse med ”LIKE”.

Som grund för denna uppsats har flertalet experiment utförs där ”LIKE” och ”REGEXP” jämförs i en MySQL databas. Försöken visar på att den eventuella försämringen i prestanda kan betala sig vid användande av multipla datatyper.

Acknowledgements

I would like to thank my supervisor Dr. Morgan Ericsson for helping me with this paper and for the interesting discussions about potential data and challenging me to go beyond my starting point with this paper.

I would also like to thank lecturer Daniel Toll for letting me evolve and indulge in regular expressions during his course about PHP.

Elin Nilsson, Klas Lundin and Eric Johansson for reading the first pre-draft and helping me understand that I needed to explain, to me obvious, things more thoroughly.

Also a big thanks to my classmates Ella Källman and Stefan Sahlin to have spurred me when I was ready to drop out.

Table of content

Introduction

When searching in MySQL databases containing large strings there can be a problem to match particular words or strings. Traditionally the statement LIKE is used to filter the results. This is a very straightforward method that has a limited amount of wildcards[1] and also a limited way of creating the matches. Regular expressions on the other hand have a more complex usage of wildcards and can be constructed to create a very precise match. This complexity also gives the regular expressions an agility that makes it possible to craft expressions that also can give a general match.

The regular expressions are more agile and can be made to create search terms that could be used in both database queries and also in most programming languages to search in for example XML files, text files and other type of data that is based on strings. Using regular expression in a database engine to search with could therefore make it possible to search multiple data storage methods with a single expression that doesn't have to be custom made for each data storage type.

The use of Regular Expressions in databases is fairly untested. A guess on why this is could be that most developers either use them after they have fetched data or they find it too complicated to use when creating queries. It could also be the fact that few people know of the possibility to use regular expressions in queries.

The questions I would like to raise in this paper are:

1. Would using regular expressions be more efficient in execution time compared to the LIKE method?
2. Would using regular expressions be more efficient in finding tuples when performing a free word search than the LIKE method?

[1] Here used in the context of something to represent zero or more characters without having to write them out or specify them.

Background

Regular expressions

Regular expressions are a convenient way of matching pattern in strings. It dates back to the early 1940's. It was first introduced by two neurophysiologists [1]. But it was later adapted by mathematicians who developed the basic notation further. Ken Thompson published an article [2] in 1968 where he mentions a regular expression compiler. This compiler produced IBM 7094 object code from regular expressions as source language. It was the beginning of *qed,* that later became *ed*[2]. This was the first attempt to make regular expressions wide spread among users. Today, almost all computer languages implement some kind of regular expression engine.

Different dialects

There are three dialects within regular expressions, traditional NFA[3], NFA POSIX and DFA[4]. At first glance we see some notable differences. One of them is backtracking. This means that the regular expression engine keep track of certain points of interest in the text where a match can differ. Another big difference is what dictates how the match is made. While DFA is text dictated the NFA is expression dictated. That makes the backtracking functionality unneeded in the DFA engines.

Suppose the expression "m(rs|r|s)"[5] is executed against the sentence "My dear mrs Astor". The traditional NFA engine will let the expression dictate how the comparison is made. It will first look for an "m" in the text. The first letter is an "M" so it would start there. After the "M" a point is saved to go back to since we

[2] A UNIX text editor.

[3] Nondeterministic Finite Automaton

[4] Deterministic Finite Automaton

[5] Matches mr, ms and mrs.

have three different options. After the "M" it would then first try to match the "r", if that is unsuccessful it would match the "s" and final the "rs", without having to rematch the "m" in the beginning for each attempt. Since there is no match here it will carry on trying to match the "m". It will find another "m" at the ninth character and it will redo the procedure to get a match at "mrs" and then return a match.

The DFA however is text dictated. This would mean that it can match all three different matches at the same time. This would mean that it will try to match "mr", "ms" and "mrs" simultaneously when it hit an "m" in the text. This would, at a first glance, make it a slower matcher than the traditional NFA. But since it doesn't do matches multiple times as with "mr" and "mrs" where the match "mr" might lead to a "mrs" the DFA have excluded the match of "ms" and it doesn't have to rematch the "r" in "mrs". So in the end, this makes it the fastest matcher of them all.

The NFA POSIX works similar to the traditional NFA. This dialect will however always strive to get the longest possible match. This will mean that it will try to match all possible outcomes every time, even when it encounters a match. This makes the NFA POSIX the slowest matcher of them all. It does on the other hand comply with the POSIX standard [3].

Another difference between the traditional NFA, NFA POSIX and the DFA is the functionality. With the traditional NFA you can use something called none capturing parenthesis. When using a parenthesis in regular expressions the match in that parenthesis is stored as a variable to be called in later. This is mostly used when performing replacements or when a certain block of text need to be stored separately. For example if one would like to store all names after the title mr, ms or mrs, the expression ".*m(rs|r|s)\.?\s(\S+)\.?"[6] could be used. In a DFA or an NFA POSIX engine, the name would be stored as the second variable. But in a traditional NFA the expression could be changed into ".*m(?:r|s|rs)

[6] Matches something previous to mr, ms or mrs, then an optional ., after that a whitespace and then something that is not a whitespace at least once and after that an optional . added for the capturing in a variable.

\.?\s(\S+)\.?"[7] and nothing but the name would be stored as a variable. This leave room for optimization, i.e. in PHPs PCRE[8] the use of non-capturing parentheses can be up to 150 times more efficient than by not using them [4].

In the traditional NFA you have an option of using lazy quantifiers. These can be used when matching with a * in your expression to ensure that you match up to the first occurrence of a condition after instead of to the last. If the previous ".*m(rs|r|s) \.?\s(\S+)\.?" would be changed to ".*?m(rs|r|s) \.?\s(\S+)\.?" it would match to the first point where a mr, mrs or ms is found in case there are multiple sentences.

Another difference worth mentioning is the lookahead and the lookbehind that can be found in the traditional NFA. To do that we add "(?=*expression*)" in front of our expression. This would make it look like this: (?=Dear).*m(rs|r|s)\.?\s(\S+)\.?. Now for this expression to match there must be a "Dear" previous to the main part of the expression. The lookbehind work in the same way, but look for something behind the main part of the expression to match. The lookbehind should also be placed in front of the main part and it would look like this: (?<=*expression*). As you can see there is a question mark (?) right after the left parenthesis (). This makes it not to be captured in as variable.

The expression used in this explanation is fairly overcomplicated. Since the only difference between "mr" and "mrs" is the "s" at the end a more efficient way of crafting this expression would be "m(rs?|s)". In this expression the "s" is optional after the "r".

Regular expressions in MySQL

Regular expressions in MySQL were probably introduced in version 4. There is a lack of documentation stating exactly when. But support forums have questions

[7] The ?: combination tell the regular expression engine the match within the parenthesis does not have to be stored as a variable.

[8] Perl Compatible Regular Expression

asked at 2006, and the documentation of the older versions (1-4) are merged on the official MySQL website. MySQL uses Henry Spencer's regex(7) implementation of regular expressions [5]. This is a DFA engine that complies with the POSIX standards.

MySQL have documented Regular Expressions as a way of replacing text in results. But there is a possibility to use them when doing a SELECT query. In that case they would be a substitute for the LIKE command to filter results to ensure that the results returned contains, or do not contain, a particular combination of characters. In these experiments the REGEXP command was used to perform SELECT queries and comparing them with the LIKE command.

LIKE

Like is a SQL[9] statement that is almost universal in all dialects of SQL. It is used to filter out search results in a SELECT query. When a database would be queried with "SELECT * FROM foo.bar;" it would return all tuples found in the table bar from the database foo. This might be too many tuples to be manageable. In the case of this report it would return over 200 000 tuples.

To limit the amount of tuples a LIKE-statement can be added to the query. For example "SELECT * FROM foo.bar WHERE content LIKE 'Joffrey';". This query however would only return tuples where the column named content is the text "Joffrey". So, it would be the same as "SELECT * FROM foo.bar WHERE content = 'Joffrey';".

With MySQL there are two kinds of wildcards. The percent character (%) that represent zero or more characters, and the underscore character (_) that represent exactly one character. With these two you can use LIKE to filter the results a bit more. With the previous example we could use "SELECT * FROM foo.bar WHERE content LIKE "%Joffrey%';". This would match all tuples where content contains the text "Joffrey" and is surrounded by zero or more characters.

[9] Structured Query Language

With these two wildcards a search term could be created that is a bit more complex. Consider this usage of wildcards "J_ffrey". This would match both Jeffrey and Joffrey. There are however a limited usage of these wildcards and they are not near the complexity of how you can use regular expressions when it come to alternation of letters and amount of alternations to be made. The term "J%ffrey" would match 'Just about ffrey' as well as 'Jeoffrey', 'Joffrey' and 'Jeffrey'.

Method

Method introduction

All experiments were conducted with the same base format. A query was constructed. Then a connection to the database was established. To prevent overhead from the program query execution the query was executed and later a timestamp was created using the current time. After this was done the response time and all search times except for the first response was received. After all matches were read, a second timestamp was created. Then a comparison between the two timestamps was made and the difference was established as the response time.

This method makes it easier to link a time to a request. Since the sheer amount of queries executed against the database would make it very time consuming to read log files generated by MySQL. It would also have been impossible to monitor MySQL workbench during the entire execution period.

All queries were also executed manually to assure that the measurements were similar to the automatic tests.

Hardware

Server

The server was a virtual machine that used VM ware vSphere 5. The hardware was configured with two Intel Xeon X5680 processors running at 3.33 GHz as top performance. It had 4 GB DDR3 (18x4GH Dual rank RDIMMs) 800MHz rams using 1333MHz DIMMs shared on 3 memory channels. This powered a Windows Server 2008 R2 Enterprise with service pack 1 that was the host of a MySQL v5.5.22 database engine.

Software

Dataset

The dataset consist of 252 759 rows containing the body e-mail conversations. This dataset was selected because of the imparities of the strings it contains and also the size of the strings to be searched in. This is to improve the hit rate of random keywords.

The dataset is a scaled down version of the Enron[10] dataset. The manipulation consisted in removing all information that would not be used. The information used was e-mail body and the ID of the e-mail.

Debris

Debris[11] is a program written in C# using .NET 4. The main function is to search in databases with the help of keywords. It was used and modified for seven types of experiments to see the difference between REGEXP and LIKE statements when querying.

The main part of the program consists of a class library and is independent of a user interface. In this experiment the class library was used and a simple console GUI was created to give some insight in how far the program had come along in its executions.

[10] Can be found at http://www.isi.edu/~adibi/Enron/Enron.htm

[11] Stem from the sound of DB, database, and RE, Regular Expression.

Experiments

Calibrating experiments

A fully randomized query was executed a random amount of times using both LIKE and REGEXP to see whether or not there would be a difference between the individual queries. Out of these every 250th execution was stored to get more precise information about that loop. The experiment was executed until the response time could be assured that there were no large offsets in response time. This was made by comparing each time to see if they were similar.

The experiment also gave a hint of how the response times could differ and still be within a normal time span.

Actual words

The list of search words is statically increased by one for each run. The list is identical except for the word added. This means that the hit rate can only be increased and never decreased. This was performed until the list reaches 15 words in length. Each query is executed five times to get a margin for time differences. The words chosen are in English. But they are not guaranteed to be found in the dataset.

Random strings

In this experiment the word list was created by words that have a low or no hit rate in the dataset. The words consisted of a combination or random alphanumeric characters to ensure that a match would be highly improbable. Each query was executed five times and the maximum length of the search array was 15 words.

Common two letter words

Searching for the most common prepositions found in the text. In these cases however regular expressions were used in a more complex way. The preposition used was:

- is
- as
- it
- at
- in
- an
- if
- of
- on

In the LIKE statement the query looked like this:

```
SELECT * FROM enron.bigstrings WHERE content LIKE '%is%'
OR content LIKE '%as%' OR content LIKE '%it%' OR content
LIKE '%at%' OR content LIKE '%in%' OR content LIKE
'%an%' OR content LIKE '%if%' OR content LIKE '%of%' OR
content LIKE '%on%';
```

With regular expressions the query looked like this:

```
SELECT * FROM enron.bigstrings WHERE content REGEXP
'[ai][st]' OR content REGEXP '[aio]n' OR content REGEXP
'[io]f';
```

A second version of the regular expressions query was constructed to test if there was a difference between separating the expressions and to use the built in regular expression or separator. That expression looked like this:

```
SELECT * FROM enron.bigstrings WHERE content REGEXP
'[ai][st]|[aio]n|[io]f'
```

There was also a control experiment performed with regular expressions where the query string was designed like this:

```
SELECT * FROM enron.bigstrings WHERE content REGEXP
'is|as|it|at|in|an|if|of|on';
```

The purpose of this control is to see if there is a difference in how you design the expression and if there is a difference it how the database execution time of the different expressions.

In these experiments no word boundary was used. This means that a match can be made with the "is" in "island" and "isotope" and not only where "is" can be found as a separate word. Each type of query was executed 250 times to get a result where you can reduce the occurrence of temporary spikes.

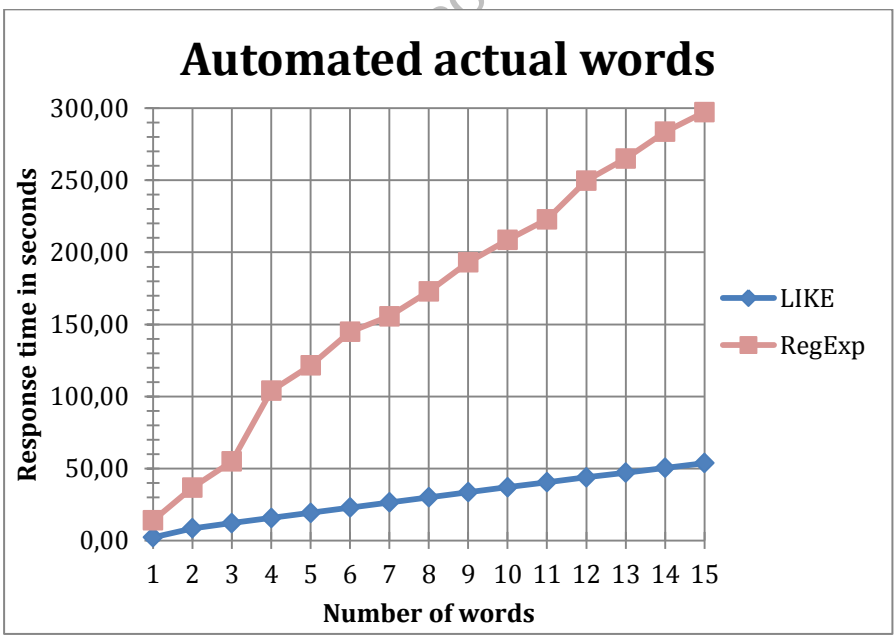
Result

Calibrating tests

When executing 2000 queries with the LIKE expression and 2000 queries with REGEXP with searching for the words: phone, Europe, sleep, food, ketchup, thin, Facebook, away, and re. No large offset in response time was found. With LIKE the largest difference between the individual query found was 0.25 seconds. In the REGEXP case the largest offset was 3.02 seconds.

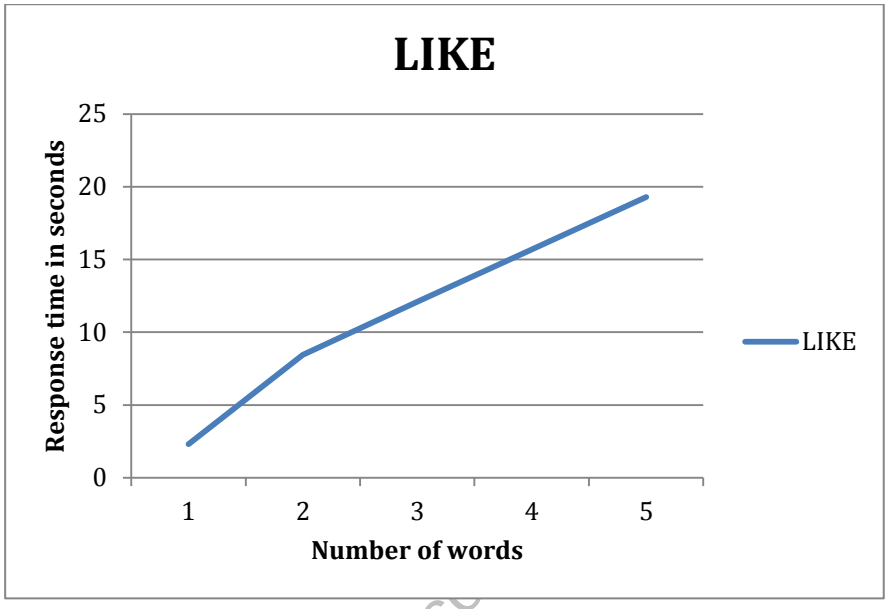
Actual words

When using a query to ensure a match the difference in the time it took to get a full response from the server differs heavily between the different statements. With one word the LIKE statements highest response time was 2.3 seconds whereas the highest response on the REGEXP statement was 14.1 seconds. (graph 1).



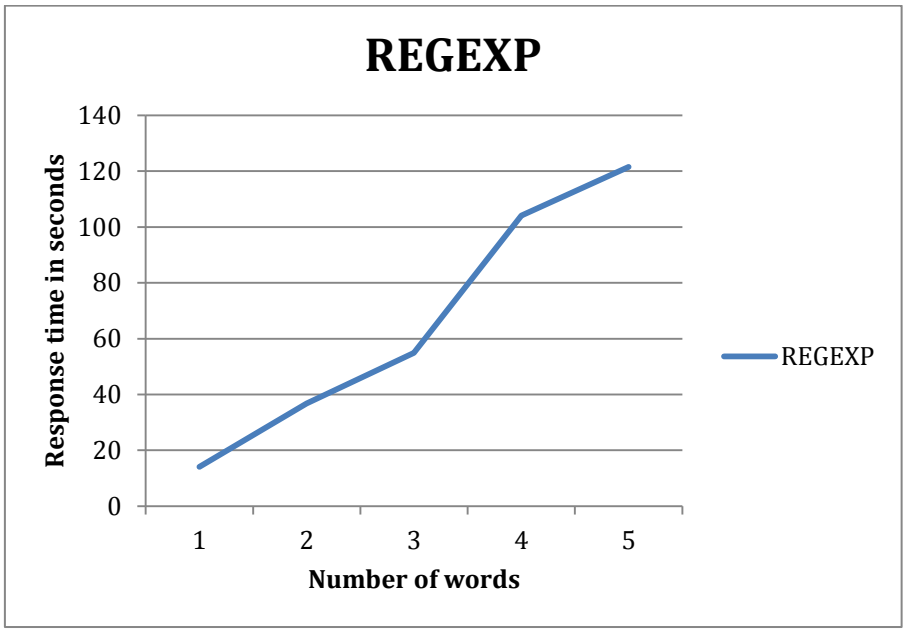
Graph 1 Difference between LIKE and REGEXP.

It should be noted that the increase in execution time for the LIKE statement had a large percentage increase between one and two key words (graph 2). The response time went up from 2.3 seconds to a highest response of 8.4 seconds, this is an increment of 365% in response time, and after this point the increase was almost linear.



Graph 2 Highest increase of LIKE.

An increase of the response time of this magnitude never occurs; the highest increase in percent is 260%. This occurs when the keywords are increased from one (14.1 seconds) to two (36.8seconds) keywords. The REXEXP (graph 3) however is not linear in the increment in this experiment. So there are other breaking points where the increment is over 150%. One to be noted is between three and four keywords where the difference is 189%.



Graph 3 Largest increase of REGEXP. X: Word count.

Random strings

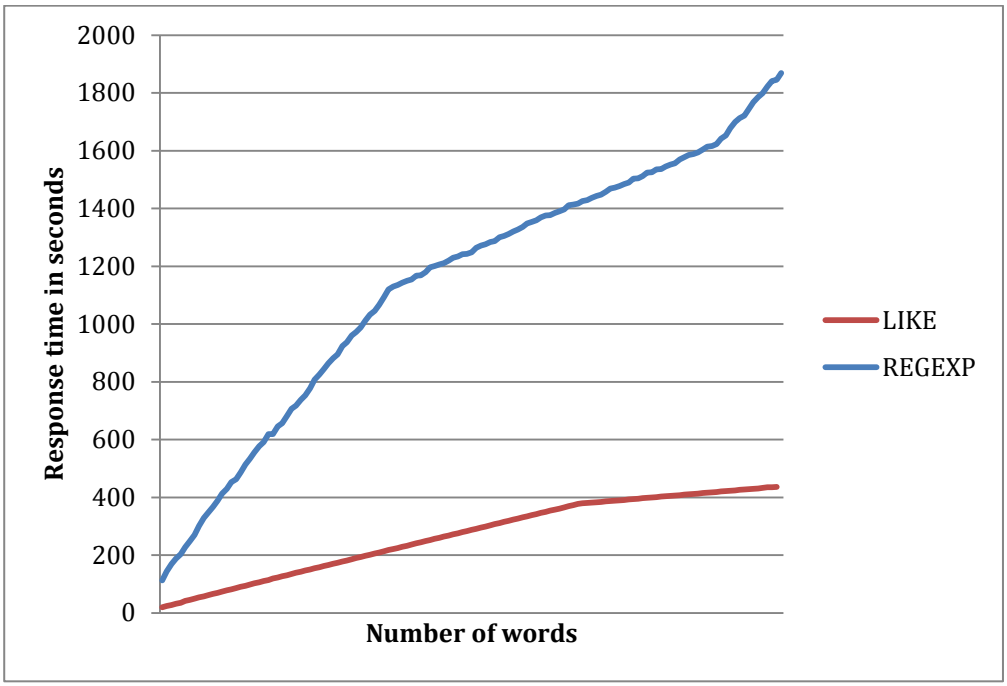
These tests proved not to be a mismatch since there were hits on randomized letter combinations. The first match occurred with the combination "Ty", later also the word "Cup" was introduced by the random string generator. There were also other combinations of words that resulted in a match.

With no match of the strings the response time in both cases were 0.0 seconds. Since I have measured response time where I exclude the first matching sequence there have to be a match for a reading to occur.

However in the case of both LIKE and REGEXP with random words the general response time was higher compared to the array of words used. For example the response time of 121 matching tuples had a response time of 19.7 seconds with LIKE and 112.2 seconds with REGEXP compared to the 19.2 seconds for LIKE and 121.4 seconds with REGEXP when the hit count was 8211 matching tuples.

In this experiment the amount of words were larger than in the previous experiments it is more visible where the response time starts to level out. But in the case

of REGEXP there is a clear increment of the response time again that is not visible in the LIKE experiments (graph 4).



Graph 4 Comparison of response time with LIKE and REGEXP with random strings.

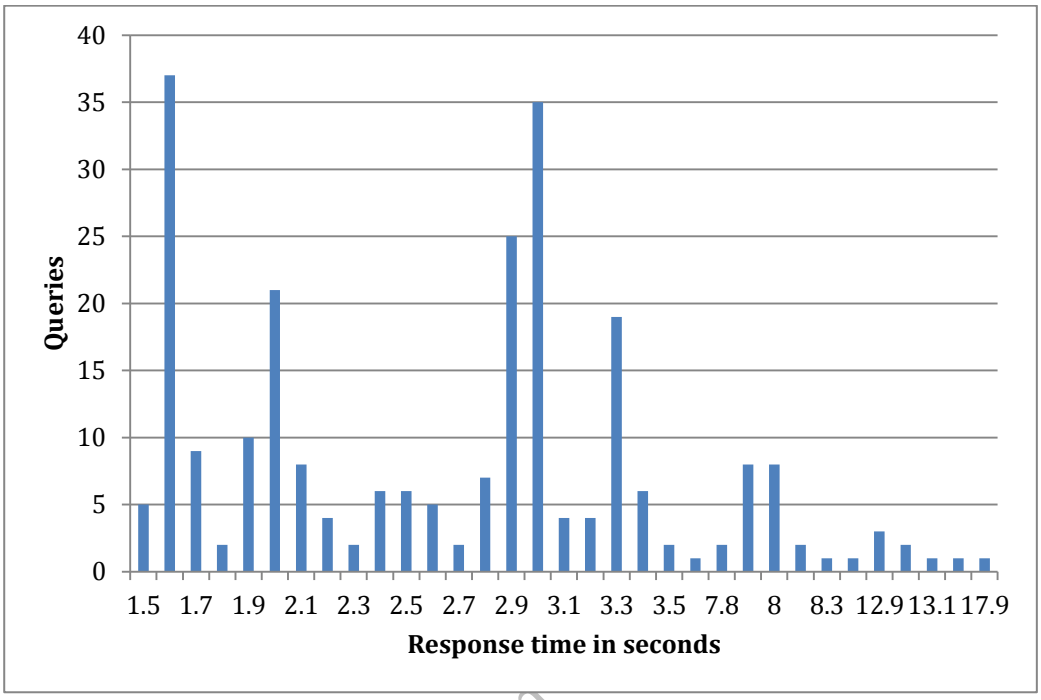
Common two letter words

When performing tests with the common two letter words found in the dataset the results were less differentiated than with the previous tests. Out of the four types of experiments made here only one had a clear difference from the other in response time.

Using LIKE

When using the first kind of query where the individual words are separated with an "OR" and the LIKE statement is used there are two major spikes in response time (graph 5). There is a large spike at 1.6 seconds and one at 3.0 seconds, the majority of the response times are situated in between these spikes. There are some times that are extremely high though. The highest response time was registered at 17.9 seconds. The span of which the queries response time can be found

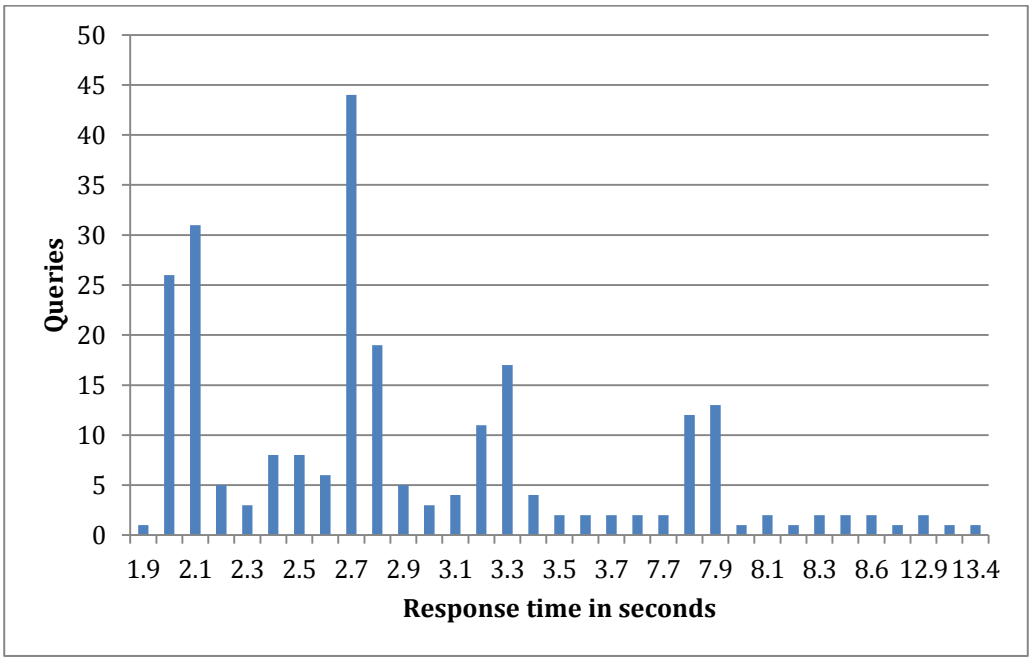
lies between 1.5 seconds and 17.9 seconds. The average response time was 3.3 seconds. The LIKE expression also had the widest range in its time span.



Graph 5 Spread of response time of the common two letter words using LIKE.

Using REGEXP with OR

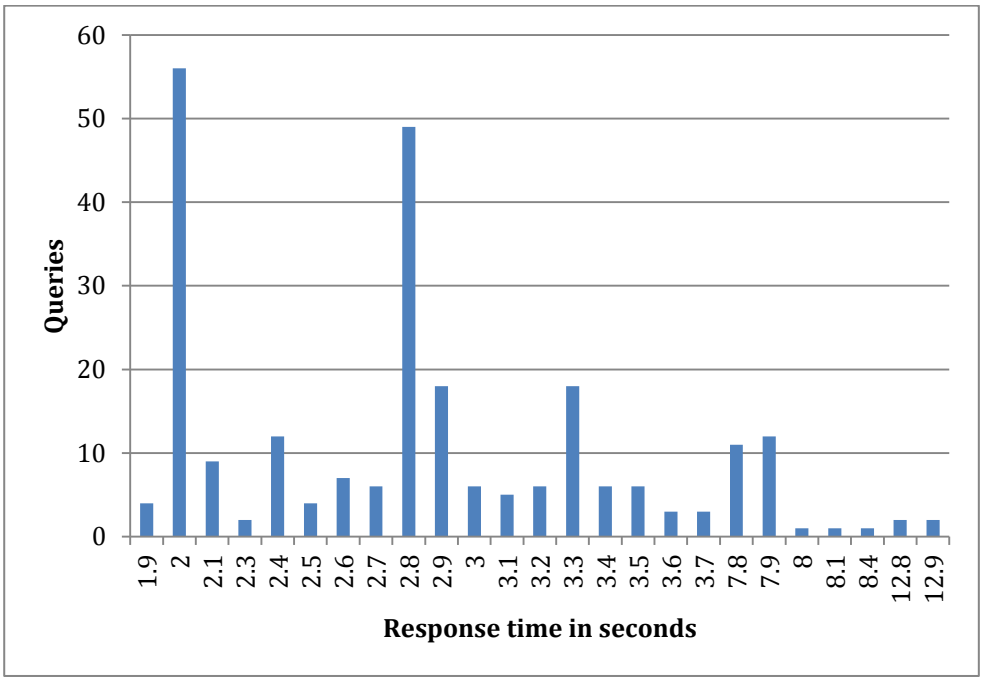
In this experiment regular expressions were used in a more complex way but with the SQL separator "OR" between different REGEXP statements (graph 6). Here there were three notable spikes. One of them at 2 seconds, the second spike at 2.1 seconds, and the third spike at 2.7 seconds. In difference to the LIKE statement there was not a containment of the majority in between two spikes. The span of the lowest spike at two seconds and the largest spike at 2.7 seconds are less than the span of the LIKE statement. But the majority of all hits are within a similar timespan of 1.4 seconds. In general the time span is smaller going from 1.9 seconds to 13.4 seconds compared to the LIKE query. The average response time was 3.7 seconds in this experiment.



Graph 6 Spread of response time of the common two letter words using REGEXP and the SQL OR statement.

Using REGEXP with regular expression separator

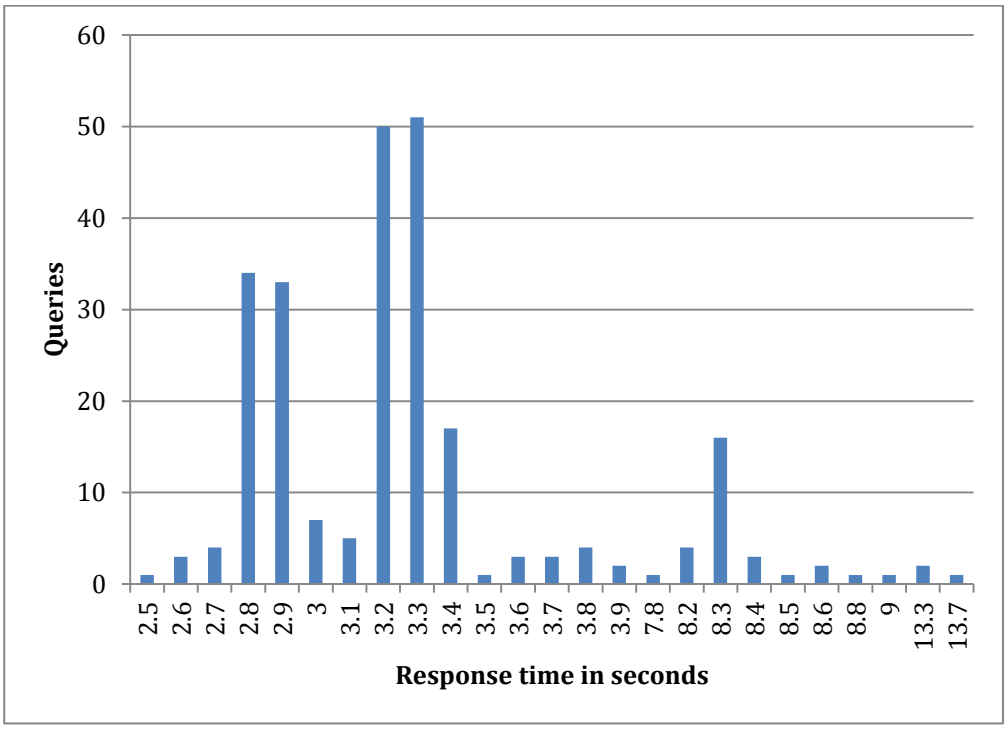
Once again there are clear spikes in the response time, with 56 response times at 2 seconds and 49 at 2.8 seconds. But as with the query using the SQL "OR" separator there is no large cluster of response times in between these spikes. The general dispersement of all response times is fairly even. This experiment had the shortest time span of responses that was situated between 1.9 seconds and 12.9 seconds. The average response time was 3.4 seconds (graph 7).



Graph 7 Spread of response times with common two letter words using REGEXP and using the regular expression or separator.

Using REGEXP with full words and regular expression separator

When using REGEXP with full words and separating them with the regular expression separator bar (|) there is a clear cluster. This is situated between 2.8 seconds and 3.4 seconds. There are a high representation of 3.2 seconds and 3.3 seconds. The fastest time was higher than with the other experiments of 2.5 seconds but the highest measured time was more like the other REGEXP queries than with the LIKE query of 13.7 seconds (graph 8).



Graph 8 Spread of response times with common two letter words using REGEXP with full words and the regular expression or separator.

Potential error factors

Since there are few earlier experiments made with a similar topic as this I have made a few errors that I noted at the end of the ten week period I had to conclude this paper in.

Since these experiments were performed on a virtual server, the overall workload on the host is unknown. Some of the times might have been influenced by heavy usage of that host computer. This could very well explain why there is in some cases an extreme peak that does not seem to fit into the other response times.

The way I chose to time my results was not optimal when it came to measure time when no tuples could be found. Then the response time was 0.0 seconds. The method to measure time did however work when matching tuples were found. Sadly I did not realize this until after more than 75% of the experiment time had passed and the experiments could not be remade with this error fix.

Conclusion

When using regular expressions to perform searches in a MySQL database the most efficient ones were those that were more optimized expressions. When the POSIX representations, i.e. [[.space.]], are used the LIKE statement is clearly more efficient than REGEXP. The experiments performed by me did not use POSIX representations on the LIKE version of the same search term. The POSIX representations evidently created a longer response time. This could depend on either that POSIX representations in general might be more time consuming to convert or that MySQL have an inefficient way of interpreting them.

When using a regular expression without POSIX representations, the tests are inconclusive. It is hard to see which of REGEXP or LIKE is more efficient. Both contain some peaks that are about the same in response time, but both have a majority of response times below 3.5 seconds. Considering the amount of tuples returned, 250 135, in the experiments not using POSIX that could be viewed as a fairly fast response time.

However, if an implementation would be made via automation of code, it would be easier for the programmer to simply use the input of a user and not change the white spaces to the correct POSIX equivalent counterparts. But in an implementation in this way the LIKE statement would probably be easier to use to automatically compose search terms.

When using the regular expression bar (|) separator the search terms are easier to read for a human when used to separate complete words. This however seem to increase the response time quite much. The response time was increased with almost a full second.

Since automatically comparing words to create an expression that is efficient with the REGEXP statement would be a fairly complex algorithm and the execution time of performing an operation like this is unknown to me at this point. It would be very interesting to create an algorithm and try using this with search terms and compare it with LIKE statements.

Why the spikes in execution time occur is unknown to me. They seem to be at a very random rate and completely dependent on how the processor work at that

time, and since the experiments were performed on a virtual machine the payload at that time might have been higher on the entire server. It is also probably influenced by how much of the cache memory is used on the server etc. This would be interesting to see the performance on a server that is better configured to be used as a MySQL server by someone who has more knowledge and experience in configuring these than me.

To conclude: With a correctly formed regular expression where the expression is designed to be optimal there is no visible difference between LIKE and REGEXP. Also POSIX representations should be avoided to increase the response time.

When the readability of the search term in the query is needed the REGEXP statement is to be preferred with the regular expression separator.

It would be interesting to recreate these experiments without using POSIX representations. It would also be highly interesting to try and create an algorithm that would be able to transform search keywords into an optimized regular expression. I would also like to see how different database engines would perform compared to each other. It would also be interesting to see how i.e. MySQL would perform if the regular expression engine was exchanged to a traditional NFA engine.

Sources

[1] J. Friedl, "A Casual Stroll Across the Regex Landscape," in *Mastering Regular Expressions*, Sebastpool, O'Reilly Media, 2006, p. 85.

[2] K. Thompson, "Programming Techniques: Regular expression search algorithm," *Communications of the ACM,* vol. 11, no. 6, pp. 419-422, 1968.

[3] The Open Group, "Posix Certified," 17 May 2006. [Online]. Available: http://get.posixcertified.ieee.org/docs/POSIX_Certification_Guide.html. [Accessed 8 May 2012].

[4] J. E. F. Friedl, "Understanding Benchmarks in This Chapter," in *Mastering Regular Expressions*, Sebastopol, O'Reilly Media, 2006, p. 249.

[5] "MySQL :: MySQL 5.1 Reference Manual :: 12.5.2 Regular Expressions," Oracle, [Online]. Available: http://dev.mysql.com/doc/refman/5.1/en/regexp.html. [Accessed 21 May 2012].

Appendix

Data logs

D1:1 - Full log and graphs of automated incremented strings with real words.

http://upload.b-zeal.net/paper/d1.1.graphs_full_log_automated_strings.xlsx

D1:2 - Full log and graphs over randomized strings.

http://upload.b-zeal.net/paper/d1.2.graphs_full_log_random_strings.xlsx

D1:3 - Partial logs (first 250 runs) and graphs of common two letter words experiments.

http://upload.b-zeal.net/paper/d1.3.graphs_partial_log_common_words.xlsx

L1:1 - Full log of all individual queries unedited.

http://upload.b-zeal.net/paper/l1.1.full_log_individual.xls

L1:2 - Unedited log of keyword searches with automated calculations.

http://upload.b-zeal.net/paper/l1.2.log_comparison_increasing_keyword_list.xls

SQL data dumps

SD1:1 - Full dump of dataset.

http://upload.b-zeal.net/paper/SD1.1.full_dataset_dump

SD1:2 - Full dump of all logs.

http://upload.b-zeal.net/paper/SD1.2.full_log_dump.sql

Source code

Debris

http://upload.b-zeal.net/paper/ConsoleGUIDebris.rar

Linnæus University
School of Computer Science, Physics and Mathematics

351 95 Växjö / 391 82 Kalmar
Tel 0772-28 80 00
dfm@lnu.se
Lnu.se