Advancing Search Query Autocompletion Services with More and Better Suggestions

Dimitrios Kastrinakis[1] and Yannis Tzitzikas[1,2]

[1]Computer Science Department, University of Crete, GREECE, and
[2]Institute of Computer Science, FORTH-ICS, GREECE
Email: {kastrin | tzitzik}@csd.uoc.gr

Abstract. Autocompletion services help users in formulating queries by exploiting past queries. In this paper we propose methods for improving such services; specifically methods for increasing the number and the quality of the suggested "completions". In particular, we propose a novel method for partitioning the internal data structure that keeps the suggestions, making autocompletion services more scalable and faster. In addition we introduce a ranking method which promotes a suggestion that can lead to many other suggestions. The experimental and empirical results are promising.

1 Introduction

Search query autocompletion is the process of computing in real time and suggesting to the user words or phrases which can complete the query that the user has already typed, on the basis of user queries which have been submitted in the past. This feature is very useful and popular in several domains and systems [6, 15, 16, 5, 1]. Initially it was used in the Amazon web site [15], providing suggestions for products related to the product a user is searching for. It has also been applied to provide *thesaurus based suggestions* [3]. Nowadays, web browsers (e.g. Mozilla Firefox) use this feature when the user is typing a URL in the address bar or completing forms, according to recent history. The same applies for the "Tab" key when using a command line interpreter (i.e. sh or bash shell in Unix). Autocompletion has been proposed also for assisting the formulation of *faceted-search queries* [5] and *underspecified SQL Queries* [14]. *Semantic autocompletion* [11] interfaces have recently become prevalent in semantic search and annotation applications. Other fields where autocompletion is involved include e-mail clients, source code editors, word processors, etc. Another notable application of autocompletion is in the field of *mashups* [1]. However, the most common and widely known use of autocompletion is the autosuggest feature used by Google, Yahoo!, Bing or Ask search engines. In this case, a user can see the most popular searches starting with the string currently being typed.

In the context of a WSE (Web Search Engine), the suggested completions are useful because in many cases the user does not know what words to use or how to describe his information need. Apart from that, this feature allows the user to find out, without any additional effort from his side, what is popular among

the internet users given the current input string, as well as the number of hits that he will get if he submits each of the suggested queries. Our objective is to increase the number and the quality of suggested completions.

To increase the number of possible suggestions we need scalable data structures. Current techniques require loading in main memory a data structure based on the contents of the entire query log. However, for very large log files, this approach would occupy too much main memory space (or may not fit in main memory at all) and loading it would require much time. To tackle this problem, we propose a novel method for *partitioning* this data structure which allows loading only the fragment that is suitable for providing suggestions on the current input entered by a user. This approach is more scalable and it is faster since loading a fragment of the data structure requires less time.

Another key aspect for the success of autocompletion services is how the suggestions are ranked, since only a small number of the possible completions (usually less than 10) are prompted. In this paper we propose a ranking method which promotes those queries which are popular and are prefixes of other popular queries. We justify the advantages of this method by measurements over the query log of a real WSE.

In a nutshell, the key contribution of our work lies in: (a) showing how partitioning can enhance the scalability of autocompletion services and (b) proposing a new ranking method which promotes a query which is a prefix of other queries and if submitted, a relatively large number of results is retrieved. The rest of this paper is organized as follows. Section 2 discusses background. Section 3 elaborates on partitioning while Section 4 elaborates on ranking. Finally, Section 5 concludes and summarizes the advantages of the proposed methods.

2 Background and Related Work

Typically a WSE maintains a log file consisting of tuples of the form: $[ip\_addr, date, query, res\_num]$, where *ip_addr* is the IP address of the user who submitted the query, *date* is the date when the query was submitted, *query* is the query string submitted and *res_num* is the number of results this query yields. Of course, not every single query should be loaded from the log file. There are two main conditions that are usually adopted as filters for the queries (see Figure 1): (a) a query must not be more than D days old (e.g. a week) and (b) a query must not yield a null query result.

The queries (as well as their popularity scores) that satisfy the above conditions have to be inserted in a data structure with low complexity of retrieval, specifically $\mathcal{O}(n)$ for a string of n characters. The most appropriate data structure is a tree where every node contains a character. A query of n characters requires n nodes for insertion, one node in each level of the tree. This type of tree is called *trie*. An example of inserting queries in a trie is depicted at Figure 3a.

Let's now describe the on-line process. Consider a user who through a browser is typing a query in the query text box of a WSE which is equipped with an

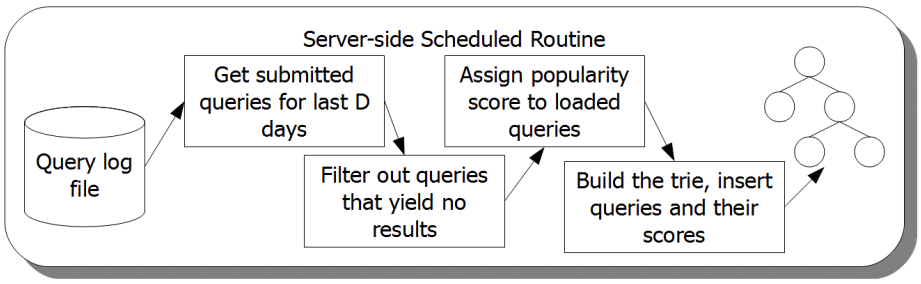


Fig. 1: Loading queries from the log file

autocompletion mechanism. Below we describe the chain of events that will occur. At first, the user types an input string *str* in the appropriate field (see Figure 2). The autocompletion client reads that string whenever a new keystroke occurs. Then, it is sent to the server that deploys the server side autocompletion component of the search engine, using asynchronous methods like *AJAX* [7]. Next, *str* is given as input for the trie, descending to the node containing its last character. A *Depth First Search* (see Figure 3b) is then applied below that node, collecting all past queries starting with *str*. The collected suggestions are ranked by their popularity score. Lastly, the server sends the *top-k* suggestions to the user.

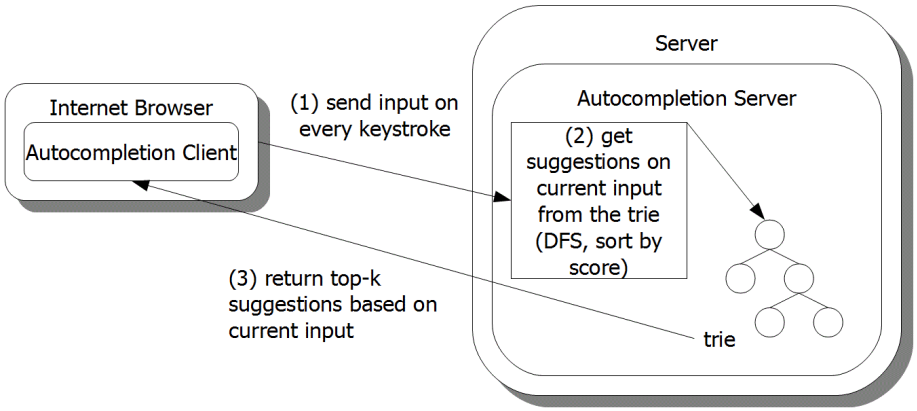
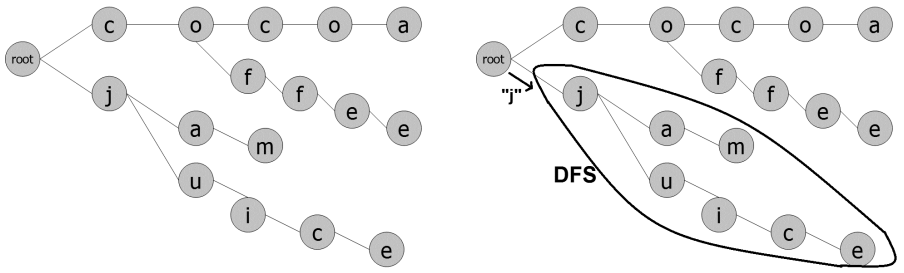


Fig. 2: Chain of events when auto completing a search query

3 Trie Partitioning

If the log file is very large, the trie containing the logged queries could be too big to fit in main memory. Even if the trie does fit in main memory, we may still want to reduce its main memory requirements in order to exploit the saved memory space for other reasons, e.g. for *result caching* [9] or *inverted list caching* [13]. Except for space requirements, loading a huge trie can take a significant amount of time. This is a major issue when dealing with autocompletion because a user wants to see suggestions *during* typing. This means that the total time interval



(a) Insertion of "cocoa", "coffee", "jam" and "juice" in a trie

(b) Retrieving suggestions "jam" and "juice" from current input"j"

Fig. 3: An example of inserting and retrieving data from a trie for autocompletion purposes

between a keystroke and the visualization of suggestions must be short (less than 1 sec). Besides, we have to take into account that loading a trie from the disc (if we have not already loaded it) is only one of the steps required (we have to be aware of the delays during packet transfer, server load, etc.). Therefore we should be able to keep the trie at a convenient size based on our space and time constraints.

3.1 The Proposed Solution

Instead of keeping suggestions in one trie, we propose partitioning it into two or more *subtries*. Eventually, the autocompletion mechanism will be using a forest of tries, each time loading the proper one based on the user input. The rising question is how should we partition the trie. At this point we should note that tree partitioning methods [12] have been applied mainly to indexing structures of database applications [2, 10, 8]. For instance, space-partitioning trees have been implemented and realized inside PostgreSQL [8], resulting in performance gains.

Partitioning by Starting Characters: Here we propose a partitioning method that is based on the starting characters (see Figure 4). Each subtrie contains queries whose starting characters belong to a specified set of characters assigned to this trie. For example, if we assume that the query log contains queries starting with latin characters only, then we can divide a trie into two subtries: one containing all queries str where $str[0] \in \{a, b, \ldots, m\}$ and another containing all strings str where $str[0] \in \{n, o, \ldots, z\}$. Let $A = \{a_1, \ldots, a_n\}$ be the set of the first characters of queries loaded from the log file, so the maximum number of tries is $|A|$. We can partition A to m ($m \leq |A|$) subsets $p_1, \ldots, p_m$, where $p_i \cap p_j = \emptyset$, $i \neq j$. Each p_i is used for determining which strings are to be inserted in a specific subtrie. Specifically, each p_i is associated with a trie that we will denote by tr_i. This association can be maintained by an index file that maps each partition p_i to a subtrie tr_i. Let $P = \{p_1, \ldots, p_m\}$ be the set that contains

these subsets and let $T = \{tr_1, \ldots, tr_m\}$ be the set of all such tries. In section 3.2 we will propose a generalization of this method for the first k characters, which provides additional advantages. We should mention at this point that related works on trie partitioning, such as those proposed in [4], are not applicable for autocompletion because the length of the query string is progressively increasing (it is not fixed or a priori-known as it is assumed in [4]).

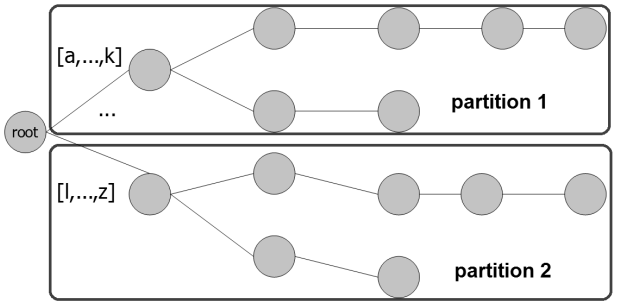


Fig. 4: Partitioning a trie based on the starting characters of inserted strings

3.2 Distributing the Starting Characters

The question that now arises is how we should form the partition P. We should not do this arbitrarily; imagine having 2 subtries and assigning: $\{a, b, \ldots, k\}$ to tr_1 and $\{l, m, \ldots, z\}$ to tr_2. Assume that the majority of queries in the log file start from the letters: 'm', 'n', 's', 't', 'o'. This means that tr_2 will be significantly larger than tr_1. This case is not better than keeping a single trie (because the size of tr_2 could be almost the size of the entire trie, and tr_2 could also be too big to host in main memory). In fact it is worse, because we may have to load the correct trie for a certain query, further delaying the appearance of suggestions to the user. Ultimately, a *distribution* of characters must be as smooth as possible, so that the tries eventually become of similar size.

Firstly, we analyze the query log file by counting the number of appearances of every starting character of each query. Specifically for every starting character c we compute its frequency $freq(c)$. Let $NTries$ be the number of subtries that we have decided to create, and $|Q|$ be the number of queries in the log file. To obtain a uniform distribution we would like $avg = \frac{|Q|}{NTries}$ queries to be inserted to each subtrie. One simple approach would be to assign n starting characters to a partition p_i until

$$\sum_{i=1}^{n}(freq(c_i)) \geq avg \ .$$

Figure 5 shows a simple example, where $Q = \{"blue", "bulb", "grape", "vial"\}$, $|Q| = 4$ and $NTries = 2$, so $avg = 4/2 = 2$. Partitions will be created as follows: $p_1 = \{b\}$ and $p_2 = \{g, v\}$. So $tr_1 = \{blue, bulb\}$ and $tr_2 = \{grape, vial\}$.

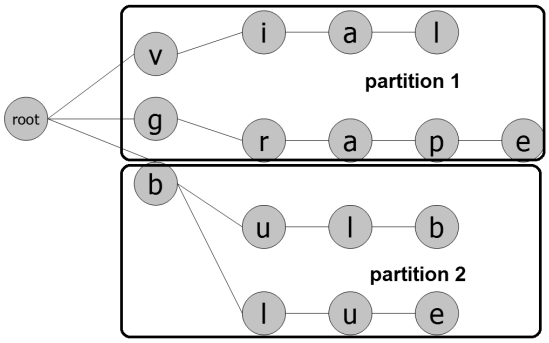
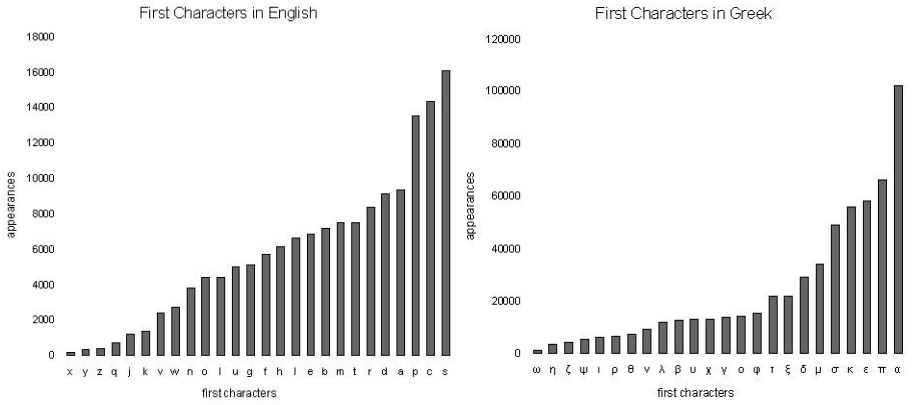


Fig. 5: Creating the partitions

However the above method is effective if the number of queries that start from a certain character are less than *avg*. Moreover the distribution of the first letters greatly affect the uniformity of the subtrie-sizes that we can achieve. To clarify this aspect we used natural language dictionaries to count the distribution of the first characters. For instance, Fig. 6a shows the distribution of the first characters for the English language. The letter 's' is the first in 16,104 out of 150,843 words contained in the dictionary, therefore 10.68% of English words start with 's'. Figure 6b shows the distribution of Greek words. There are 102,201 words starting with α out of 574,737 words contained in the dictionary, therefore 17.78% of Greek words start with α. Fig. 7 shows the distribution of starting characters in queries stored in a log file[1]. Again, 's' is the most frequent first letter, appearing in 8.2% of all the queries.



(a) Distribution of first letters in English (b) Distribution of first letters in Greek

Fig. 6: Distributions of first letters in English and Greek

[1] The log contained queries submitted to the Excite (March 13, 1997) search engine. URL: www.excite.com

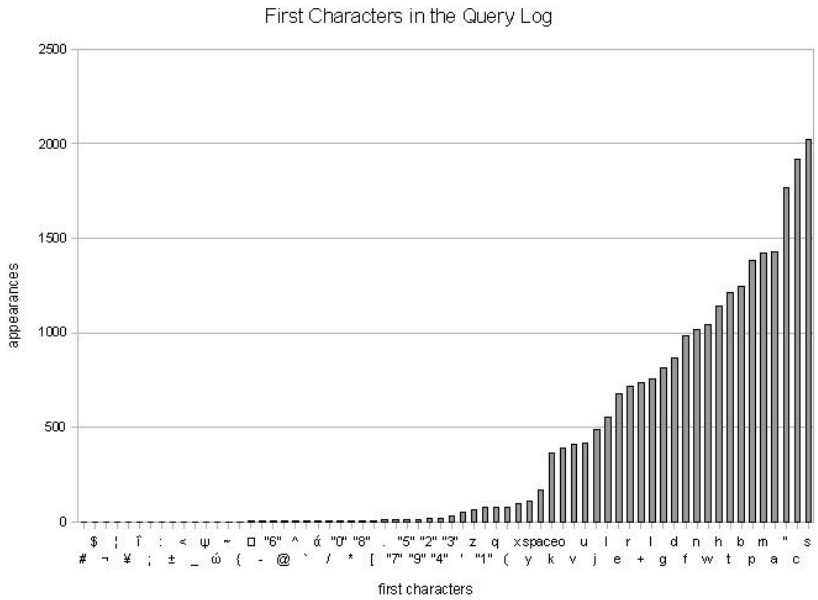


Fig. 7: Distributing queries by first characters, having used a query log from the Excite search engine

The above facts motivate an alternative method for partitioning: instead of distributing queries by their first characters only, we distribute them based on their first k characters. This allows achieving smoothness even if we have a character c such that $frec(c) > avg$. For example, instead of creating a single partition for 's', multiple partitions of 's' are created based on the next letter (if $k = 2$). Since each subtrie now corresponds to a prefix of k characters, no suggestions are computed for the first $k - 1$ characters of the user input.

The exact algorithm for partitioning is shown in Fig. 8. At first it collects all possible prefixes of k characters from the query log and counts their frequency. Then it creates a partition and keeps populating it until the total frequency becomes greater than the desired partition capacity. After that, a new partition is created and so on. Notice that A is sorted lexicographically and each $p_i \in P$ is characterized by the *range* of prefixes that have been associated to it. For example, suppose that $k = 2$ and p_i has been associated to "sa", "se" and "so". In this case only "sa" and "so" are kept, meaning that the queries that will be assigned to subtrie tr_i are those whose prefix is in the range $[\text{"}sa\text{"}, \ldots, \text{"}so\text{"}]$. Also notice that the algorithm does not require building and keeping the entire trie in main memory. After collecting the k-prefixes it reads the queries from the log file and distributes them to the appropriate partition. The above method of partitioning has to be repeated periodically.

Alg. *Distribute k First Characters to Partitions*
Input: k, *Capacity // the desired partition capacity in queries*,
QL *// log file with all submitted queries.*
Output: $P = \{p_1, \ldots, p_m\}$.
(1) $A = \emptyset$;
(2) for each query $q \in QL$ do
(3) $a = q[1 \ldots k]$; *// a holds the first k chars of q*
(4) if $a \in A$ then $a.frequency = a.frequency + 1$;
(5) else
(6) $a.frequency = 1$;
(7) $A = A \cup \{a\}$;
(8) end if;
(9) end for;
(10) sort(A); *// lexicographically*
(11) $i = j = 0$;
(12) while $i < |A|$
(13) $j++$;
(14) while $p_j.size < Capacity$
(15) $i++$;
(16) $p_j = p_j \cup \{a_i\}$;
(17) $p_j.size = p_j.size + a_i.frequency$;
(18) end while;
(19) $p_j.min = min_{lex}(p_j)$; *// the minimum lexicographically*
(20) $p_j.max = max_{lex}(p_j)$; *// the maximum lexicographically*
(21) $p_j.clear()$; *// we keep min and max only*
(22) end while;
(23) return P;

Fig. 8: Algorithm for distributing the k first characters of the logged queries to each p_i. Complexity: $\mathcal{O}(nk + mk \log m)$ where $n = |QL|$, $m = |A|$, and obviously $m \leq n$

3.3 Measurements

In this section we report experimental measurements[2]. We used the query log from the Excite search engine which contained $\sim 25,500$ distinct queries. For this data set we created various possible partitionings with $k = 1$ comprising 2, 3, 4, ..., 10 and 66 subtries (with $k = 1$ there are 66 distinct first characters in the log, i.e $|A| = 66$). Let LT be the time required to determine which trie to load, load that trie, and return the top 10 suggestions. For each case we computed the average LT of the generated tries as follows:

$$avg\_LT_{T_n} = \frac{\sum_{tr \in T_n} LT_{tr}(c)}{|T_n|} ,$$

where c is a single character given as input for the trie tr in order to get suggestions, $T_n = \{tr_1, \ldots, tr_n\}, n \in \{2, \ldots, 10, 66\}$. The measured times are shown in Fig. 9. Compared to non-partitioning, with 66 subtries and $k = 1$ we achieve a $Speedup = \frac{avg\_LT_{T_0}}{avg\_LT_{T_{66}}} = \frac{4000}{233} \simeq 17.1$ meaning that the partitioning is 17 times faster.

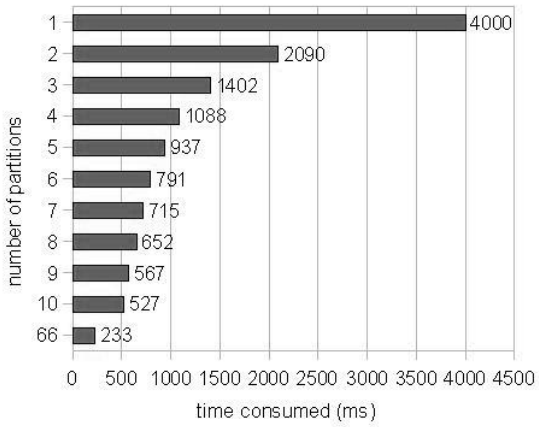
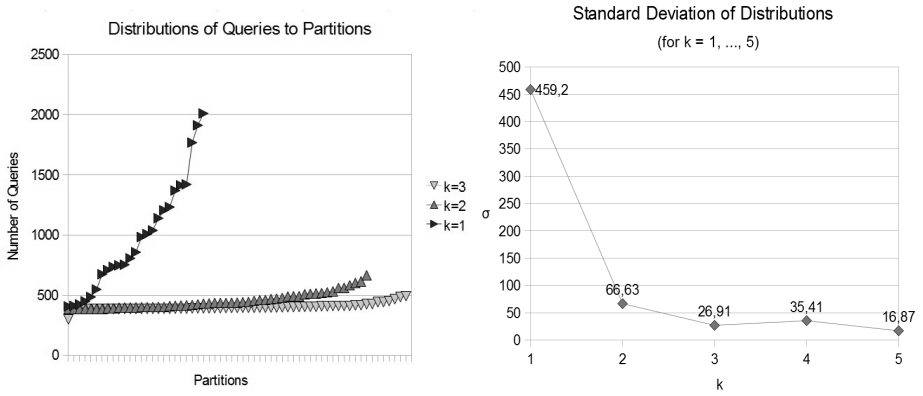


Fig. 9: Average LT for 1, 2, ..., 10 and 66 partitions and $k = 1$.

Fig. 10a shows how characters were distributed when partitioning was based on the first character only ($k = 1$), the first two characters ($k = 2$), and the first three characters ($k = 3$). The desired number of queries per partition was $avg = \frac{|Q|}{NTries} = \frac{25,500}{66} \simeq 386$, so the ideal distribution would have 386 queries per partition. Fig. 10b shows how "close" we are to the ideal distribution for $k = 1, \ldots, 5$, by plotting the standard deviation for each distribution, ($\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (x_i - \mu)^2}$, where $N = NTries$, each x_i represents the number of queries in partition i, and μ is the mean value of these numbers). It is obvious

[2] Programming Language: Java, Platform: CPU: Intel Core 2 Duo E6750 @ 2.66GHz, 4GB RAM, 2x 10,000RPM RAID0 disks, OS: Windows 7 x64.



(a) Distributing queries to each partition for $k = 1, 2, 3$

(b) Standard deviation of distributions for $k = 1, \ldots, 5$

Fig. 10: Distributing queries to partitions

from the plots, with $k \geq 2$, the distribution is dramatically smoother (closer to the ideal) than in the case of $k = 1$.

Synopsizing, we have seen that partitioning by the starting k-characters can lead to uniform in size subtries, which is crucial for respecting main memory constraints and reducing load times. The bigger k is, the smoother the distribution of queries to subtries becomes. The only drawback of a high k value is that we may not be able to compute suggestions for user inputs of size less than $k - 1$ characters. For example, suppose that $k = 2$ and assume that $P = \{[ab - az], [ba - bi], [bl - bz], ...\}$. If the user types "b" we cannot compute suggestions using one subtrie because b is distributed to the second and third trie, so we have to wait until the user types another character. Notice however, that if the user types "a" then we can compute suggestions since "a" is covered entirely by the first subtrie.

4 Ranking Suggestions

Another key aspect for the success of an autocompletion service is how the suggestions are ranked, since only a small number of the possible completions are prompted. One might think of several criteria that could be used (in isolation or in aggregation) for ranking suggestions, e.g. popularity, number of results, etc. In this section we propose ranking methods that are based on both popularity and number of results and also promote those queries which are popular and are prefixes of other popular queries.

The computation of popularity is based on the contents of the query log file. However note that just counting the submissions of a certain query from the log file is not sufficient; imagine a user of a search engine entering a certain

query 1,000 times. For this reason we count only those submissions coming from distinct sources only, specifically distinct IP addresses[3], therefore the query log contains IP addresses.

Let $q \rightsquigarrow q'$ denote that q is a prefix of q'. Let q_u denote the query the user has typed. We want to assign a score to each candidate completion q (where $q_u \rightsquigarrow q$). First we introduce a metric that takes into account popularity and answer size:

$$PopSize(q) = \frac{freq(q)}{MAX\_freq} \cdot \frac{res\_num(q)}{MAX\_res\_num} \tag{1}$$

where

- $freq(q)$ is the number of distinct IP addresses that have submitted q,
- MAX_freq is the maximum frequency of the queries in the log,
- $res\_num(q)$ is the number of results the query q yields,
- MAX_res_num is the maximum number of results of the queries.

The above formula assumes that all queries are independent. However some queries are prefixes of other queries and this observation should be taken into account. For example, consider the queries $q_1 =$ "$music$" $q_2 =$ "$music\ composers$", $q_3 =$ "$mammals$", $q_4 =$ "$mammals\ from\ Africa$", and assume that all of them have the same popularity. Now suppose that we have to compute the top-2 suggestions. The queries q_1 and q_3 are good candidates since each is a prefix of another popular query (of q_2 and q_4 respectively).

Let $Reach(q)$ be the set of all queries that have q as a prefix, i.e. $Reach(q) = \{\ q' \mid q \rightsquigarrow q'\}$. To exploit the above observation, here we propose another ranking formula, $PopSizeReach$ defined as:

$$PSR(q) = a * PopSize(q) + b * \frac{1}{|Reach(q)|} \sum_{q' \in Reach(q)} PopSize(q')\ , \tag{2}$$

where a and b are constants that range in (0,1) and $a + b = 1$. The second part of the formula is the average $PopSize$ of the queries that have q as prefix.

An alternative approach is to increase the score gained for queries having large $Reach(q)$ by removing $\frac{1}{|Reach(q)|}$:

$$PSR_2(q) = a * PopSize(q) + b * \sum_{q' \in Reach(q)} PopSize(q')\ , \tag{3}$$

A probabilistic approach is also possible. Here we want to assign a score to each q (where $q_u \rightsquigarrow q$) that reflects the probability that the user will select q if he has typed q_u. The estimation of the probability is again based on the log file. Specifically, we define

$$Score(q) = \frac{DeepFreq(q)}{\sum_{q_u \rightsquigarrow q'} DeepFreq(q')} \tag{4}$$

[3] Distinct IP addresses do not necessarily imply distinct users. It is just a simple mean of approaching the number of distinct users that entered a certain query.

where $DeepFreq(q) = freq(q) + \sum_{q \rightsquigarrow q'} freq(q')$.

Table 1 and 2 show the suggestions produced by the first two formulas when typing the query "books" and "news" respectively using the Excite query log. In this case, the computation of popularity ignored the number of results of each query because this information was not available in the query log. Because of this, (3) and (4) behave the same, producing identical suggestion rankings. Therefore only (4) is included in the tables. In Table 1, a notable change in ranking is "bookstore", which is 7th using (1) and first using (2). Similarly, in Table 2 "news" is 4th using (1), 3rd using (2) and 1st using (4). On the other hand, a long query has less probability of having many other queries that contain it as prefix. This fact is depicted in Table 2, where "newspaper vancouver washington" is 3rd using (1) but only 6th using (2).

Table 1: Actual suggestion rankings for query "books" using formulae (1), (2) and (4).

(1)	(2), $a = 0.5$, $b = 0.5$	(4)
bookstores and catalogue and 1-800	bookstore	books
books	bookstores and catalogue and 1-800	bookstore
books a million	books	bookstores and catalogue and 1-800
books and titles	books a million	books a million
books on tape	books and titles	books and titles
books: crime and punishment	books on tape	books on tape
bookstore	books: crime and punishment	books: crime and punishment

Table 2: Actual suggestion rankings for query "news" using (1), (2) and (4).

(1)	(2), $a = 0.5$, $b = 0.5$	(4)
newspaper clark county washington	newspaper clark county washington	news
newspapers	newspapers	newspaper
newspaper vancouver washington	news	newspaper clark county washington
news	newspaper	newspapers
news and server	newsgroups	newsgroups
news group	newspaper vancouver washington	newspaper vancouver washington
newsgroups	news and server	news and server
newspaper book reviews	news group	news group

However, the above examples are just indicative and they do now allow us to draw safe conclusions regarding these formulas. To evaluate a ranking method through a user study is a laborious and expensive task and often yields results which are not repeatable. In general we can say that there is not yet any formal methodology and method for evaluating ranking methods for query completions. For instance [3] mentions a user study but does no report any concrete results, while the majority of works on autocompletion (e.g. [5]) focus on efficiency and the quality of suggestions is by no means evaluated. For this reason, we introduce a metric that can measure the *predictive power* of a ranking method. The key point is that it requires as input only a query log. The main idea is the following: we use a prefix of a submitted query and then measure the rank of the whole query in the list of completions suggested (and ranked) by the ranking formula under evaluation. It follows that a ranking formula f is better than a ranking formula $\acute{f}$ if the ranks yielded by f are lower than those of $\acute{f}$.

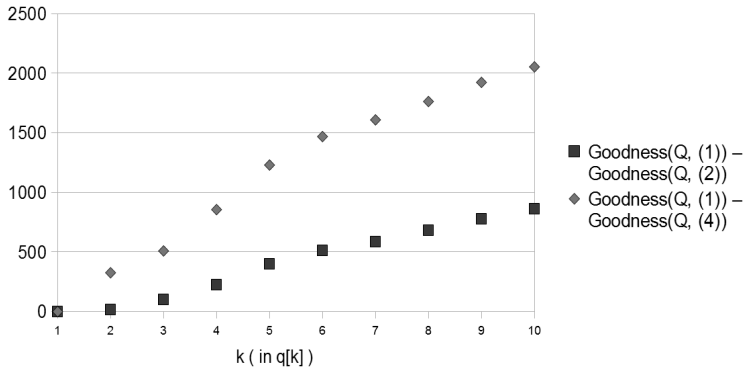


Fig. 11: Comparing the ranking methods using the difference of *Goodness* for each ranking method.

Let $q[k]$ denote the k-char prefix of a query q of the log file. We shall use $Rank_{(1)}(q, q[k])$ to denote the position of q in the suggestions produced by formula (1) if we sort them in descending order. For example, if $q =$ "*company*", $q[2] =$ "*co*" and the suggestions produced by formula (1) are $\langle$*core*, *computer*, *company*, *car*$\rangle$ then $Rank_{(1)}$("*company*","*co*") = 3. The less $Rank_f(q, q[k])$ is, the better formula f behaves assuming the given log file. To take into account all queries of a query log, we define the "goodness" of a scoring formula f over a query log Q as follows:

$$Goodness(Q, f) = \sum_{q \in Q} Rank_f(q, q[k])$$

As our objective is to comparatively evaluate scoring functions, it is not necessary to make any kind of normalization. A formula f is better than a formula $\acute{f}$ if $Goodness(Q, f) < Goodness(Q, \acute{f})$. Of course, one could generalize and consider all prefixes of q, not only one (i.e. the k-char prefix).

Returning to the problem at hand, we applied the above metric on formulas (1), (2), (3) and (4) for $k = 1, \ldots, 10$, using the Excite query log. Here, Q was accessed as a set that contained the distinct submitted queries, so duplicate queries were not included in ranking. Fig. 11 shows the plot of $Goodness(Q, (1)) - Goodness(Q, (2))$ and the plot of $Goodness(Q, (1)) - Goodness(Q, (4))$. Since the latter plot is higher than the first, it follows that (4) is better than (2), and this holds for all values of $k = 1, \ldots, 10$ (since both plots are positive, both (4) and (2) are better than (1)). The reason we did not include $Goodness(Q, (1)) - Goodness(Q, (3))$ in the plot was the unavailability of the number of results each query yields in the Excite log. Therefore (3) and (4) behave the same and produce the same *Goodness* results.

Another aspect of a ranking method is the amount of time required for computing the scores. Table 3 reports the times required for each ranking method using the Excite log (which contained $\sim 25,500$ distinct queries). Since we want to return suggestions to the end user in real time (otherwise the autocompletion feature would be useless), we pre-compute the scores for every query stored in the log and this is done off-line by the autocompletion server as shown in Fig. 1. In this way the computation of query completions is almost instant.

Table 3: Time consumed while computing scores for every ranking formula.

$PopSize$ (1)	$PopSizeReach$ (2)	$PopSizeReach_2$(3)	$DeepFreq$ (4)
$\sim 0.35s$	$\sim 60s$	$\sim 60s$	$\sim 54s$

5 Concluding Remarks

To make autocompletion services more scalable, we proposed a method for partitioning the trie of logged queries. This partitioning allows increasing the number of suggestions that can be hosted, and speeding up their computation at real time. As an example, consider an amount of main memory sufficient for hosting 25,500 different suggestions. By partitioning the trie with respect to the first character (i.e. $k = 1$), the same amount of memory can host 1,659,027 more suggestions, i.e. almost two orders of magnitude more, and the loading time is 17.1 times shorter. Since the first characters are not uniformly distributed in natural languages, we proposed a partitioning that is based on the first k characters which can be used for yielding uniform in size subtries.

Finally, we proposed a novel method for ranking suggestions, where the score of each suggestion depends on (a) its popularity (distinct submissions), (b) the number of results it yields if submitted, and (c) the suggestions that contain the current suggestion as prefix. To comparatively evaluate such ranking functions we introduced a metric measuring the predictive power of a ranking method and we identified the ranking method that prevails over a query log file of a real WSE. To the best of our knowledge no other work has elaborated on *index partitioning* or *structure-aware ranking*. We believe that these techniques can enhance autocompletion services in various applications.

References

1. Serge Abiteboul, Ohad Greenshpan, Tova Milo, and Neoklis Polyzotis. Matchup: Autocompletion for mashups. In *IEEE International Conference on Data Engineering*, pages 1479–1482, Shanghai, China, 2009.
2. Walid G. Aref and Ihab F. Ilyas. Sp-gist: An extensible database index for supporting space partitioning trees. *Journal of Intelligent Information Systems*, 17(2-3), December 2001.
3. Mario Arias, Jose M. Cantera, and Jesus Vegas. Context based personalization for mobile web search. In *VLDB 08*, August 2008.
4. Sanjay Baberwal and Ben Choi. Speeding up keyword search for search engines. In *3rd IASTED International Conference on Communications, Internet, and Information Technology*, pages 255–260, St. Thomas, US Virgin Islands, November 2004.
5. Holger Bast and Ingmar Weber. When you 're lost for words: Faceted search with autocompletion. In *SIGIR06 Workshop on Faceted Search*, Seattle, Washington, USA, August 2006.
6. Dwayne E. Bowman, Ruben E. Ortega, Michael L. Hamrick, Joel R. Spiegel, and Timothy R. Kohn. Refining search queries by the suggestion of correlated terms from prior searches. *Patent Number: 6,006,225*, December 1999.
7. Chrisina Draganova. Asynchronous javascript technology and xml (ajax). http://www.myacrobatpdf.com/6319/asynchronous-javascript-technology-and-xml-ajax.html.
8. M. Y. Eltabakh, R. Eltarras, and W. G. Aref. To trie or not to trie? realizing space-partitioning trees inside postgresql: Challenges, experiences and performance. In *Procs of the 31st VLDB Conference*, Trondheim, Norway, 2005.
9. T. Fagni, R. Perego, F. Silvestri, and S. Orlando. Boosting the performance of web search engines Caching and prefetching query results by exploiting historical usage data. In *ACM Transactions on Information Systems (TOIS)*, pages 51–78, 2006.
10. Thanaa M. Ghanem, Rahul Shah, Mohamed F. Mokbel, Walid G. Aref, and Jeffrey S. Vitter. Bulk operations for space-partitioning trees. In *20th International Conference on Data Engineering*, March 2004.
11. E. Hyvonen and E. Makela. Semantic autocompletion. In *Proceedings of the First Asia Semantic Web Conference, ASWC 2006*, Beijing, China, 2006.
12. Philippe Jacquet and Mireille Regnier. *Trie partitioning process: Limiting distributions*, volume 214. Springer Berlin / Heidelberg, 1986.
13. J.Zhang, X.Long, and T.Suel. Performance of compressed inverted list caching in search engines. In *Proceedings of the 17th international conference on World Wide Web*, Beijing, China, April 2008.
14. Terrence Mason and Ramon Lawrence. *Auto-completion of Underspecified SQL Queries*. Springer Berlin/Heidelberg, 2006.
15. Ruben E. Ortega, John W. Avery, and Robert Frederick. Search query autocompletion. *Patent Number: US 6,564,213 B1*, May 2003.
16. Ronald M. Whitman and Christofer L. Scofield. Search query refinement using related search phrases. *Patent Number: US 6,772,150 B1*, August 2004.