-This is an electronic version of an Article published in:
12th International Conference on Web Information System Engineering,
Sydney, Australia, 2011
-The final publication is available at:
http://link.springer.com/chapter/10.1007/978-3-642-24434-6_8

Exploiting Available Memory and Disk for Scalable Instant Overview Search

Pavlos Fafalios and Yannis Tzitzikas

Institute of Computer Science, FORTH-ICS, GREECE, and
Computer Science Department, University of Crete, GREECE
{fafalios,tzitzik}@csd.uoc.gr

Abstract. *Search-As-You-Type* (or *Instant Search)* is a recently introduced functionality which shows predictive results while the user types a query letter by letter. In this paper we generalize and propose an extension of this technique which apart from showing on-the-fly the first page of results, it shows various other kinds of information, e.g. the outcome of results clustering techniques, or metadata-based groupings of the results. Although this functionality is more informative than the classic search-as-you type, since it combines *Autocompletion*, *Search-As-You-Type*, and *Results Clustering*, the provision of real-time interaction is more challenging. To tackle this issue we propose an approach based on pre-computed information and we comparatively evaluate various index structures for making real-time interaction feasible, even if the size of the available memory space is limited. This comparison reveals the memory/performance trade-off and allows deciding which index structure to use according to the available main memory and desired performance. Furthermore we show that an incremental algorithm can be used to keep the index structure fresh.

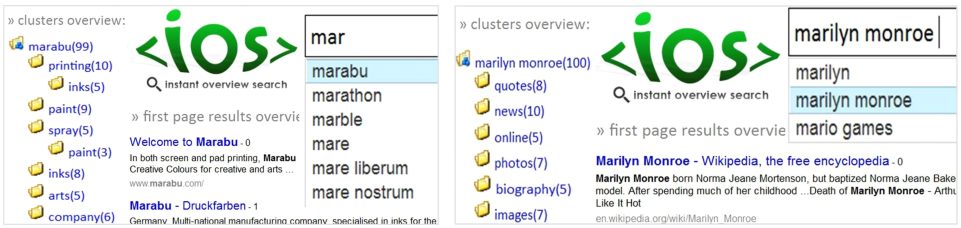
1 Introduction

Autocompletion services help users in formulating queries by exploiting past (and logged) queries. Recently, Google adopted *Instant Search*, a new *Search-As-You-Type* (for short *SAYT*) enhancement that apart from allowing the user to find out what is popular given the current input string, it also shows an overview of the top hits (first page of results) as the user types a query. In particular, one or more possible matches (actually "completions") of the typed query are computed and immediately their results are presented to the user. This immediate feedback allows the user to stop typing if the desired results have already been recommended. If on the other hand the sought results are not there, the user can continue typing or change what he has typed so far. In general, we can say that the user adapts his query on the fly until the results match what he wants. Although the usefulness of this functionality has not been measured or proved (we did not manage to find any work that reports the results of a user study), it is true that people type slowly but read quickly (a glance at another part of the

page typically takes only a few milliseconds[1]), implying that the user can scan a results page while he types. Furthermore, we observe an increasing interest on providing such functionality evidenced by the emergence of several systems e.g. *EasySearch*[2], *Keyboardr*[3], or *Wiki Instant*[4].

Apart from the benefit for the user's side, SAYT is also beneficial for the WSE (Web Search Engine), in the sense that the suggested answers have been pre-computed, thus the engine has to evaluate less queries at run time.

In this paper we generalize and propose a more powerful search-as-you-type functionality which apart from showing on-the-fly only the first page of results of the guessed query, it can show several other kinds of supplementary information that provide the user with a better overview of the search space. We shall call this paradigm of searching *Instant Overview Searching*, for short `IOS`. Regarding the supplementary information, we focus on *results clustering* since it can provide informative overviews of larger parts of the answer. Such *"Cluster-As-You-Type"* functionality, can be considered as an instance of `IOS`. To grasp the idea, Fig. 1 shows an indicative screen dump. Consider a user who wants to find information about the biography of *Marilyn Monroe* and for this reason he starts typing the query "marilyn monroe biography" letter by letter. After having typed the first three letters, i.e. the string "mar", a set of query completions are prompted as shown in Fig. 1a. At the same time, the first page of results and the cluster label tree that correspond to the top suggestion "marabu" appear instantly (at the main area and the left bar of the screen respectively). At that point the user can either continue typing or select a suggestion from the list. If he selects one of the suggestions the first page and the clustering of the results appear immediately. Moreover, the user is able to click on a cluster's name and get the results of only that cluster.



(a) Start typing a query. (b) Select the suggestion "marilyn monroe".

Fig. 1: IOS indicative screen dumps.

Suppose the user continues typing and presses the key *"i"*. The list of suggestions is refreshed according to the new input "mari", the top suggestion is now different (i.e. "marilyn") and thus the first page and the cluster label tree

[1] http://www.google.com/instant/
[2] http://easysearch.webs.com/home.htm
[3] http://keyboardr.com/
[4] http://wikinstant.com/

change according to the new top suggestion. Moreover, since there is the suggestion "marilyn monroe" that matches what he would type, the user selects it (Fig. 1b) and then the cluster label tree and the first page of the query "marilyn monroe" appears immediately. We observe that the cluster label tree contains a cluster with label "biography (5)". The user clicks on that cluster and gets the results that are relevant to his information need. Notice that the user typed only 4 letters and made only 2 clicks in total. Furthermore, he have seen how the results about Marilyn Monroe are clustered.

It is not hard to see that efficiency and real-time interaction is challenging. To tackle this challenge, we propose enriching the trie structure [13] which is used for query autocompletion with the various pre-computed supplementary information. For example, and for the case of clustering, for each query in the trie we keep the outcome of the results clustering algorithm, specifically the cluster label tree (the cluster label tree is a tree of strings, each being the label, readable name, for a cluster). This choice greatly reduces the required computation at interaction time, however it greatly increases the space requirements of the trie. For this reason in this paper we describe and comparatively evaluate various index structures. Essentially each index structure is actually a method for partitioning the information between main and secondary memory. We have conducted a detailed performance evaluation according to various aspects such as system's response time and main memory cost, which reveals the main performance trade-off and assists deciding which index structure to use according to the available main memory and desired performance. Furthermore we show that an incremental algorithm can be used to keep the index structure fresh in affordable time.

Finally we should clarify that the overall effectiveness of the provided functionality, apart from its efficiency, depends on the quality of the pre-computed information, i.e. on the quality of (a) the ranking method that determines query suggestions, (b) the hits in the first page of results, and (c) the labels returned by the clustering. However we should stress that the index structures that we introduce and their analysis can be used with *any* autocompletion, ranking and clustering method, making our approach widely applicable.

The rest of this paper is organized as follows. Section 2 motivates `IOS` by sketching applications. Section 3 discusses related works. Section 4 introduces the index structures and Section 5 reports extensive comparative experimental results. Finally, Section 6 concludes and identifies issues that are worth further research.

2 Applications of IOS

In this section we describe in brief two main forms of `IOS`; over a *meta* web search engine (MSE), and over a *standalone* web search engine (WSE).

For instance, [8] describes a MSE over Google that clusters at real time the retrieved results according to the words that appear in the title and the snippet of each result. Such a system could pre-compute and store the cluster

label tree and the top-k results for each query of the log file. In this way the system could provide an overview of the results allowing the user to adapt his query while he is typing. Suppose that a user searches for "apple iphone" and starts typing that string. After having typed "apple", he notices that there is a cluster with name *apple iphone* with 15 results. Instantly, he can click on that cluster label and view the results. If the results do not satisfy his information need, he can continue typing. However we should note that in the context of a MSE the delivered results depend on the results of the selected underlying search engines. Since the results of the same query may be different at different points in time, the pre-computed and stored cluster label tree of the query results may be different from the cluster label tree of the current query results. This means that a particular stored term/label of a cluster may not exist in the cluster label tree of a future search of the same query. Therefore, when a user clicks on a cluster's term, the system may not be able to perform the query and then focus on the selected cluster unless the results of this cluster (i.e. its top-10 hits) have been also stored beforehand. Of course that approach increases the amount of pre-computed information that has to be stored and fetched. As we shall see, the index structures that we will introduce can be used to provide real-time interaction also in such cases.

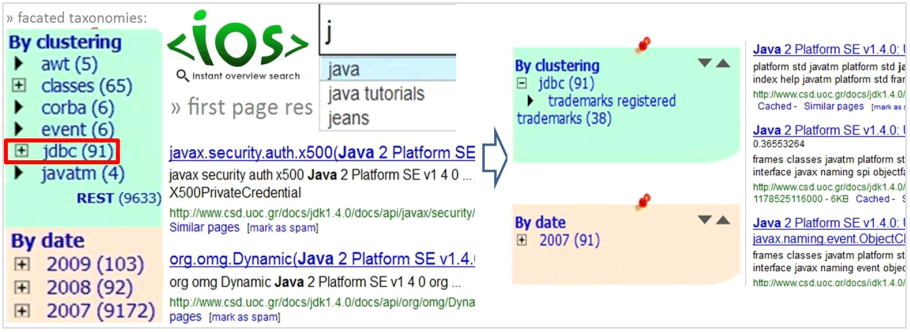


Fig. 2: `IOS` over a faceted search engine.

In the context of a standalone WSE, the engine can pre-compute and store not only the cluster label tree, but also the facets (metadata-based groupings) of the top-k results of each logged query. For example, in [12] the engine characterizes the top-k results according to five facets: *by clustering*, *by domain*, *by date*, *by filetype* and *by language*. The provision of such information during typing can accelerate the search process. For instance, consider a user seeking for SIGIR 2010 papers. While he types this query letter by letter, he notices that for the top suggestion "SIGIR", there is the term *2010* below the facet *By date* with 50 results. This means that he can directly click on that term and get the corresponding results (on user click the engine executes the top suggested query and directly focuses, i.e. restricts the answer on the clicked term/facet). Two screendumps of a prototype application we have developed are given at Fig. 2.

It follows that IOS can be used for accelerating the performance of multifaceted browsing interfaces [4, 3, 6].

3 Background and Related Work

A SAYT system computes answers to keyword queries as the user types in keywords letter by letter. Fig. 3 illustrates the architecture of such a system. At each keystroke of the user the browser sends (in AJAX style) the current string to the server, which in turn computes the top suggestions and returns to the browswer the best answers ranked by their relevancy to the top-suggested query.

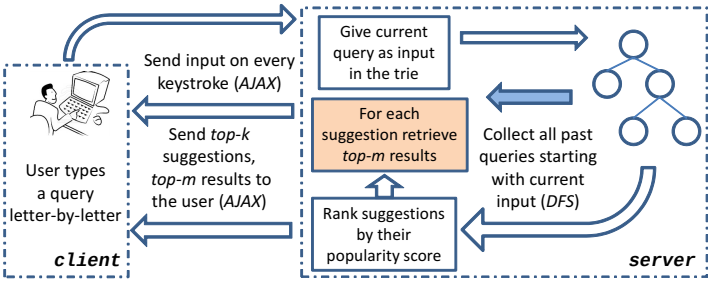


Fig. 3: A Search-As-You-Type (SAYT) System.

There are several works describing letter-by-letter searching for various kinds of sources, e.g. for relational databases [5, 15, 10, 11], or documents [2, 9]. To cope with the efficiency problem, most use pre-computed results and in-memory indexes (i.e. the entire index is loaded in main memory). The only work that does not load everything in main memory is the semantic search engine *CompleteSearch*[5]) [1]. That work differs from ours in the following aspects: (1) CompleteSearch is not a generic search engine, but is based on specific known datasets (like Wikipedia and DBLP) with predefined semantic categories, (2) it's suggestion system is word based, not query based, i.e. it suggests only words that match user's current input, not whole queries, (3) it focuses on compression of index structures, especially in disk-based settings, while IOS uses partial in-memory indexes, (4) in general CompleteSearch helps users to formulate a good semantic query while IOS helps users to locate directly and fast what they are looking for.

4 Index Structures

To tackle the requirements of IOS, we propose enriching the trie that is used for autocompletion with the results of the pre-processing steps. Specifically, and for the scenario of Fig. 1, for each query stored in the query log file, we extend its entry in the trie by two additional strings. The first string contains the first

[5] http://search.mpi-inf.mpg.de

page of results of the guessed query (i.e. an HTML string containing for each hit its title, snippet and URL). The second string contains the *faceted dynamic taxonomy*, for short facets (or only the cluster label tree in the case of a MSE). Note that we prefer to store both strings in HTML format in order to save time while presenting the results to the user (no need for any post-processing).

Obviously, such enrichment significantly increases the size of the trie, since for each query we have to save two additional long strings. Specifically, in the original trie for each logged query of n characters, the trie keeps a node of about n bytes (usually $n = 16$). However, the string that represents the cluster label tree (or the facets) can be about 30,000 bytes and the string that represents the first page of results is about 40,000 bytes (including the characters of the HTML code for both strings). Below we propose methods and index structures for managing this increased amount of data.

The first idea is to adopt the *trie partitioning* method proposed in [7]. According to that method, only one "subtrie" (of much smaller size) is loaded in main memory. Since users seldom change their initial queries [14], dividing the trie in this way implies that once the appropriate subtrie has been loaded (during the initial user's keystrokes), the system can compute the completions of the subsequent requests using the same subtrie. Now suppose the case where we do not have enough main memory to load the enriched trie (or subtrie). In such case we can save the results of the preprocessing steps (e.g. facets, cluster label tree, first hits), in one or more different files. Consequently, the trie for each query entry has to keep only three numbers: (a) one for the file, (b) one for the bytes to skip and (c) one for the bytes to read. Obviously, one should use *random access files* for having fast access to the pre-computed information. This approach greatly reduces the size of the trie, however we have to perform additional disk accesses for reading the pointing file.

Note that this approach could be used in the context of a MSE, in order to store also the results of each cluster (as we discussed previously). We could further reduce the size of the trie that is loaded in the main memory by combining the last two approaches (trie partitioning and trie with indices to external files). Such approach requires very small amount of main memory, however it requires more time for loading the appropriate subtrie in the main memory and reading the data from the pointing file.

To clarify the pros and cons of the aforementioned approaches, we decided to comparatively evaluate the following approaches (depicted at Fig. 4):

(a) **(SET) Single Enriched Trie.** The entire enriched trie in main memory.
(b) **(STIE) Single Trie with Indices to External files**. Single (query) trie in main memory with pointers to external random access files where a pointer consists of 3 numbers (file number, bytes to skip, and bytes to read).
(c) **(PET) Partitioned Enriched Trie.** The *enriched* trie is partitioned to several subtries using the partitioning proposed in [7].
(d) **(PTIE) Partitioned Trie with Indices to External files.** The (query) trie is partitioned ([7]) and each subtrie contains pointers to external random access files.

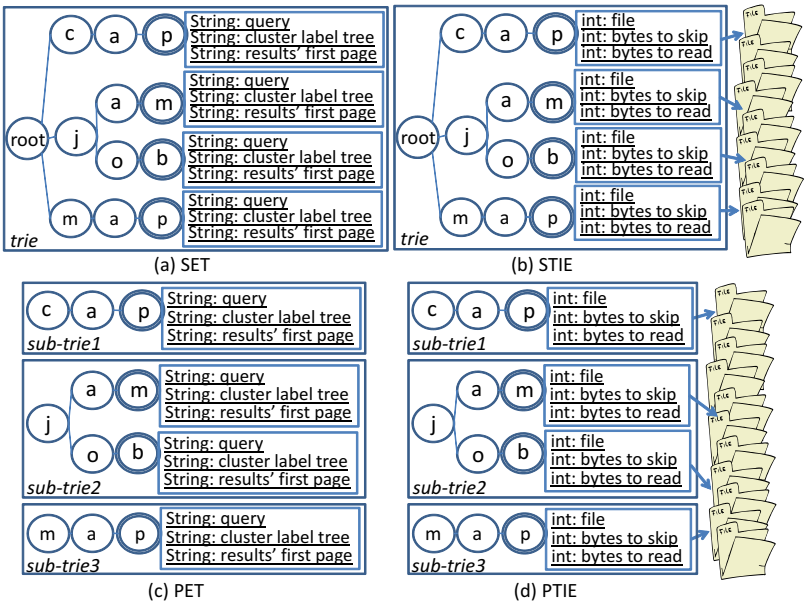


Fig. 4: Index implementation approaches.

5 Experimental Evaluation

The objective is to comparatively evaluate the previous approaches, to identify the more scalable one(s) and to clarify the main performance trade-offs.

Evaluation Aspects We evaluated the above approaches according to the following aspects:

[*Trie Size to be loaded in main memory*] We will measure the size of the trie that has to be loaded in main memory. The key requirement for those approaches which are based on a single trie, is to fit in memory, since loading is done only once (at deployment time), so the loading time of the trie does not affect the performance at user requests. Instead, the loading time is very important for those approaches relying on multiple subtries, since each user request may require loading the appropriate subtrie from the disk.

[*Average retrieval time*] We will measure the average time for retrieving the suggestions, the results and the clusters/facets for a specific current user's query. This time does not contain the network's delay time and the javascript running time.

[*Construction Time and Update Time*] We will measure the time required to process the query log and construct the corresponding index. This task has to be done once, however it should be redone periodically since the log file changes. We should clarify at this point that in contrast to the problem of inverted file construction, in our case we do not have main memory problems at construction time, specifically the most space consuming part of the pre-computed information (i.e. the outcome of results clustering and the first page

of hits) can be directly stored in the external file(s) at construction time, thus there is no need for creating and merging partial indices (as it is the case of inverted file construction). We will also investigate the time required to *update* the index structures (after query log change) instead of constructing them from scratch.

Num. of log's queries	Num. of unique queries	Avg num. of words per query	Num. of distinct words	Avg num. of chars per query
1,000	578	2.23	950	15.5
10,000	5,341	2.3	6,225	16
20,000	10,518	2.34	10,526	16.2
40,000	20,184	2.35	17,179	16.2

Table 1: Query Log Files

Data Sets and Setup We used 4 query log files of different sizes. One with 1,000, one with 10,000, one with 20,000 and one with 40,000 queries. Each file is a subset of a random log sample of about 45,000 fully anonymized queries from a *real* query log file (from Excite[6] WSE). Table 1 illustrates some features of these files. We should note that it is not necessary to have a very big set of distinct queries for an `IOS` functionality. Note that in WSE query logs the query repetition frequency follows a Zipf distribution [16], i.e. there are few queries that are repeated frequently and a very large number of queries which have very small frequency. Obviously the latter are not very useful to be logged and used as suggestions for an `IOS` functionality in the sense that they will not be suggested as completions (their rank will be very low). Regarding trie partitioning, we created subtries each corresponding to the first two characters ($k = 2$) of the queries. As shown in [7], for $k = 2$ the partitioning yields subtries whose sizes are very close (close to ideal). All experiments were carried out in an ordinary laptop with processor Intel Core 2 Duo P7370 @ 2.00Ghz CPU, 3GB RAM and running Windows 7 (64 bit). The implementation of all approaches was in Java 1.6 (J2EE platform), using Apache Tomcat 6 and Ajax JSP Tag Library.

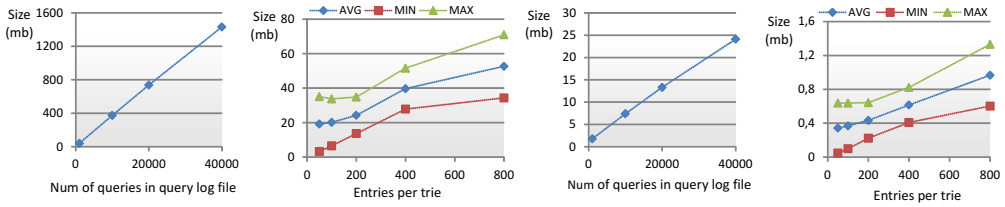
Results on Trie's sizes Fig. 5 illustrates how the size of the trie that is loaded in the main memory grows in each approach.
In the SET approach (Fig. 5a), the size of the trie increases linearly with the query log size and can get quite high. In particular, we observe that SET requires about 40 MB for storing the results and the clusters of a query log file of 1000 queries (578 distinct queries), i.e. it needs about 70 KB per query.
In the PET approach (Fig. 5b), we examine how the size of a subtrie grows as a function of the entries (results and clusters of a query), we decide to store in it. Since the subtries do not have the same size, the diagram presents the worst case (max size), the best case (min size) and the mean case (average size) of a subtrie's size (for a query log file of 40,000 queries). However, the smaller this number is, the more subtries have to be created. For example, selecting to store 50 entries in each subtrie, 170 tries have to be created (note that this also

[6] http://www.excite.com

depends on the number of distinct substrings of size k, for more see [7]). On the other hand, selecting to store 800 entries per subtrie, only 24 subtries are created.

In the STIE approach (Fig. 5c), the number of entries we select to store in each separate file does not affect the size of the trie that is loaded in the main memory. We observe that this approach leads to a very small trie's size. In particular, STIE requires about 1,8 MB for a query log file of 1,000 queries (578 distinct queries), i.e. it needs only about 3 KB per query. For a log file of 40,000 queries, selecting to store 50 entries in each file leads to the creation of 404 external files (with average size of about 3.5 MB each one). At the same time, selecting to store 400 entries in each file, 51 files have to be created (with average size of about 28 MB each one).

In the PTIE approach (Fig. 5d), we combine trie partitioning and trie with indices to external files in order to further reduce the size of the trie. As in PET, we examine how the number of the entries we select to store in each subtrie affects its size (as we mentioned above, the number of entries we select to store in each separate file does not affect the size of the trie). Since the subtries have not the same size, the diagram presents the worst case (max size), the best case (min size) and the mean case (average size) of a subtrie's size (for a query log file of 40,000 queries and selecting to store 400 entries in each external file). We observe that even for the worst case, the size of a subtrie is extremely low.



(a) SET (b) PET (c) STIE (d) PTIE

Fig. 5: Size of the Trie

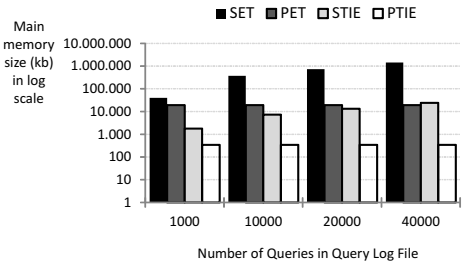
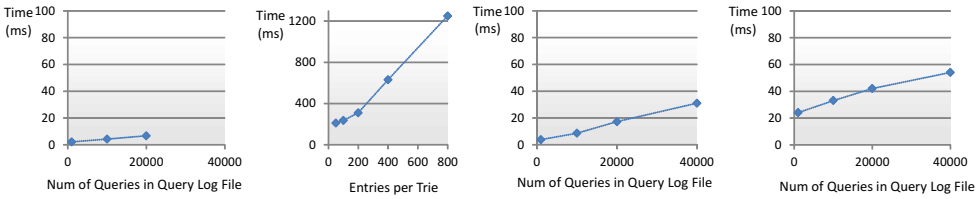


Fig. 6: Comparison of the trie's size.

Conclusion. Fig. 6 compares the four approaches where the y-axis is in logarithm scale. For the PET and PTIE approach, we choose to depict the

best case (50 entries per trie), so the average size of a subtrie is constant and independent of the query log size. As expected, SET requires the more main memory space, which can be very big for large query logs. The best approach (regarding only the size of the trie) is PTIE with great difference from the others. STIE follows but for query logs of smaller than 40,000 queries. Finally, PET requires less space than STIE only for very large query logs files (more than 40,000 queries).

Average Retrieval Time Fig. 7 depicts the average retrieval time for each implementation approach (for the PET approach, the corresponding diagram concerns a query log file of 40,000 queries).

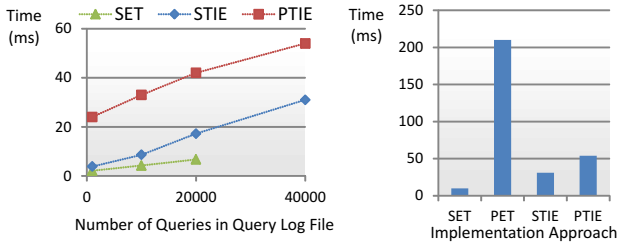


(a) SET (b) PET (c) STIE (d) PTIE

Fig. 7: Average retrieval time.

We can see that SET has very low average retrieval time (almost constant), taking only a few milliseconds to retrieve the required data even for very large query log files (Fig. 7a). However, it requires that the entire trie fits the main memory. In PET, the average retrieval time depends on the size of the appropriate subtrie that needs to be loaded in the main memory, i.e. the number of entries per subtrie. We observe that for all cases, PET is much slower than SET (Fig. 7b). The STIE approach was implemented using *random access files* and therefore the average retrieval time does not depend on the size of the external file. We observe that this approach is very efficient even for large query log files with average retrieval time lower than 40 ms (Fig. 7c). Finally, as in STIE approach, PTIE was implemented using random access files. For this reason, its average retrieval time depend mainly on the size of the query log file (Fig. 7d). We observe that this approach is a bit worse than STIE. The reason is that it requires one more disk access in order to find and load the appropriate subtrie.

Synopsis. Fig. 8b compares the average retrieval time of all approaches (for PET approach, we consider the best case of 50 entries per subtrie). It is obvious that the SET approach is much more efficient than all the other approaches. However, as mentioned above, for large query log files its size is huge and it does not fit in main memory. For this reason, one may argue that the best approach is STIE, as it combines low trie size and very fast retrieval time. Moreover, PTIE is a very good approach as it offers very small trie size and efficient retrieval time. Finally, PET approach is the worst although its retrieval time is not unacceptable (about 200 ms). Fig. 8a compares only the SET, STIE and PTIE approaches, excluding PET (as it is independent on the size of the query log file). We observe



(a) PET vs. STIE vs. PTIE (b) All indexes

Fig. 8: Comparison of the average retrieval time.

that in all approaches, the average retrieval time is very low. SET is the more efficient approach, followed by STIE, and finally by PTIE.

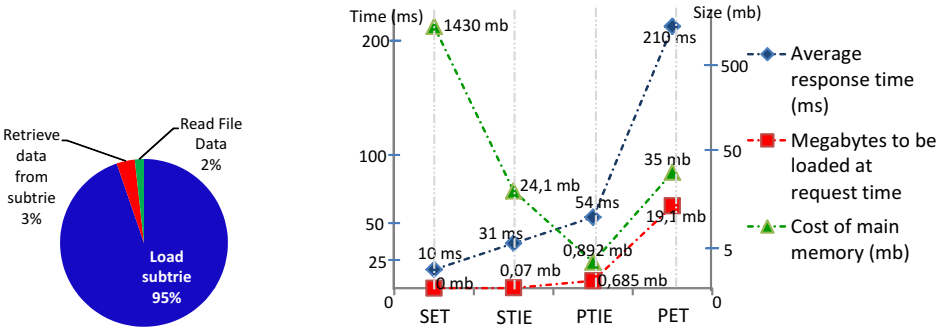
At last we should mention that experiments over bigger synthetic query logs yield the same conclusions[7].

5.1 Selecting the Right Index

The main conclusion is that the proposed functionality is feasible for real time interaction even if the log file and the preprocessed information have considerably high size. The selection of the implementation approach depends on the available main memory, the size of the log file, and the size of the preprocessed information. Below we describe criteria that should be used and in the right order.
1/ If the entire SET fits in memory then this is the faster choice since no loading has to be done during user's requests.
2/ If SET does not fit in memory then, the next choice to follow is STIE since it offers faster retrieval time in comparison to PET and PTIE. However note that STIE is feasible only if the trie of the query log fits in main memory (which is usually the case), if not then PTIE approach has to be used.
3/ Finally, we could say that the more scalable approach is PTIE, since it can be adopted even if the available main memory has very small size. Furthermore, the experiments showed that PTIE is very efficient (with retrieval time lower than 60 ms) and can be used even with very large query log files. Fig. 9a analyzes the retrieval time of PTIE's main tasks. However, more information has to be loaded to main memory at request time in comparison to SET and STIE. This limits the throughput that is feasible to achieve. This problem can be alleviated by adopting a caching scheme, an issue for future research.

Fig. 9b summarizes the results and illustrates the trade-off between (a) response time, (b) amount of bytes to be loaded in main memory for serving one request, and (c) cost (amount) of main memory that should be available. The values concern a query log of 40,000 queries (20,184 unique queries). For the PET approach we consider the best case of 50 entries per subtrie.

[7] For example over a synthetic query log with 230,000 distinct queries STIE gave average resp. time 80ms while PTIE slightly higher: 80-90ms.



(a) Retrieval time analysis of PTIE.

(b) Summarized results for all approaches.

Fig. 9

We observe that SET does not load anything at request time, while STIE loads only the bytes that has to read from the external file (about 70 KB). For this reason, these two approaches achieve very fast response time. PTIE needs about 54 ms for loading the appropriate subtrie (of average size about 622 KB) and reading the results from the external file, while PET spends about 210 ms to load the appropriate subtrie (of average size about 19,1 MB) that contains all the needed information. On the other side, SET requires vast amount of main memory contrary to the other three approaches that require much less.

5.2 Construction and Update

Construction For the construction of the trie, the main tasks are (a) the analysis of the query log file, (b) the execution of each distinct query in order to get the first page and the cluster label tree of the results and (c) the creation of the trie's file. Of course the time required by these tasks depend on the particular query evaluation or clustering method that it is employed. Table 2 reports the average times of these tasks for various sizes of the query log.

Number of log's queries	Query log file analysis time (ms)	Results and clusters retrieval time (ms)	Trie creation time (ms)	Total time (sec)
1,000	4	592,515	1,259	594
10,000	9	5,415,150	10,156	5,425
20,000	12	10,802,970	19,950	10,823
40,000	16	21,105,780	34,760	21,141

Table 2: Trie's Construction Time.

The results are almost the same for all index approaches and for this reason we present only the results of the SET approach (for the STIE and PTIE approach, the creation of the external files has very small time impact and for the PET and PTIE approach, the time for creating all the subtries is almost equal to the time SET requires to create a single big trie).

We observe that the task requiring the most time is the retrieval of the results and their clustering. For example, for the query log of 1,000 queries (578 distinct queries), we can see that the retrieval of the results and their clustering takes about 592 seconds, i.e. around 1 second per query.

Update Note that the index should be updated periodically (based on the contents of the constantly changing query log and the new fresh results of the underlying WSE). One policy is to update the index daily. Since the construction takes some hours (as we saw earlier) it is worth providing an *incremental* method. An incremental approach is to create the trie of the new query log file and then merge the old "big" trie with the new one which is much smaller. If an entry of the new trie does not exist in the initial trie then we just add the new query with all its data to the initial trie. If however an entry of the new trie exists in the initial then we have to update its results, its clusters and it's popularity (which is used by the autocompletion algorithm). As we have seen earlier, the time for creating a trie depends on the size of the query log file (about 1 second for each query in our setting). For example, if the new query log has 250 distinct queries then it's trie creation time is about 4 minutes.

For testing the time required for merging two tries, we run an experiment with an initial trie of about 11,000 distinct queries (767 MB) and a new trie of 250 distinct queries (14 MB). The execution time was about 2 minutes. We can see that the incremental approach is much more efficient (6 minutes versus about 3 hours).

6 Conclusion

In this paper we introduced a generalized form of search-as-you-type functionality which apart from suggesting query completions and showing fast the first page of results, it shows various other kinds of supplementary information for giving the users an overview of the information space. The effectiveness of this functionality depends on the quality of the pre-computed information (e.g the quality of the ranking method for the query suggestions, the quality of the hits in the first page of results, the quality of the labels returned by the clustering). In any case, the provision of such services at real time significantly increases the amount of information that has to be stored and fetched at run time. For this reason we focused on the problem of efficiency, and we comparatively evaluated various index structures which partition the pre-computed information in various ways. This comparison reveals the main performance trade-off and allows deciding which index structure to use according to the available main memory and desired performance. Contrary to past works, the proposed structure (PTIE) partitions the index and can be adapted to a system's main memory capacity. This means that one can exploit large amounts of pre-computed information (even images produced by visualization algorithms). We should also note that the performance is independent of the size of the collection; it is affected only by the size of the query log (in particular by the number of distinct queries), which

often has a limited size as we discussed earlier. Furthermore, and since the index should be updated periodically based on the contents of the constantly changing query log and the new fresh results of the underlying WSE, we described an incremental approach for updating the index. Finally we should clarify that the proposed implementation method can be used with any ranking, clustering or autocompletion method. Issues which are worth further research include: (a) *caching* mechanisms for further reducing the information that has to be loaded at user request time and thus increasing the throughput that can be served, and (b) methods for *automatically* selecting the more beneficial index according to details of the particular setting (e.g. size of cluster label tree, size of first page of results, etc), the available main memory, and desired performance.

Acknowledgements This work was partially supported by the EU project SCIDIP-ES (FP7-283401).

References

1. H. Bast, A. Chitea, F. Suchanek, and I. Weber. Ester: efficient search on text, entities, and relations. In *SIGIR 2007*, pages 671–678. ACM, 2007.
2. H. Bast and I. Weber. Type less, find more: fast autocompletion search with a succinct index. In *SIGIR 2006*, pages 364–371. ACM, 2006.
3. S. Basu Roy, H. Wang, G. Das, U. Nambiar, and M. Mohania. Minimum-effort driven dynamic faceted search in structured databases. In *CIKM 2008*. ACM.
4. W. Dakka, P. Ipeirotis, and K. Wood. Automatic construction of multifaceted browsing interfaces. In *CIKM 2005*, pages 768–775. ACM, 2005.
5. S. Ji, G. Li, C. Li, and J. Feng. Efficient interactive fuzzy keyword search. In *WWW 2009*, pages 371–380. ACM, 2009.
6. A. Kashyap, V. Hristidis, and M. Petropoulos. Facetor: cost-driven exploration of faceted query results. In *CIKM 2010*, pages 719–728. ACM, 2010.
7. D. Kastrinakis and Y. Tzitzikas. Advancing search query autocompletion services with more and better suggestions. In *ICWE 2010*. Springer, 2010.
8. S. Kopidaki, P. Papadakos, and Y. Tzitzikas. Stc+ and nm-stc: Two novel online results clustering methods for web searching. *WISE 2009*, pages 523–537, 2009.
9. G. Li, J. Feng, and L. Zhou. Interactive search in xml data. In *WWW 2009*, pages 1063–1064. ACM, 2009.
10. G. Li, S. Ji, C. Li, and J. Feng. Efficient type-ahead search on relational data: a tastier approach. In *SIGMOD 2009*, pages 695–706. ACM, 2009.
11. S. Li, W. Yu, X. Gu, H. Jiang, and C. Fang. Efficient interactive smart keyword search. *WISE 2010*, pages 204–215, 2010.
12. P. Papadakos, S. Kopidaki, N. Armenatzoglou, and Y. Tzitzikas. Exploratory web searching with dynamic taxonomies and results clustering. In *ECDL 2009*. Springer, 2009.
13. H. Shang and T. Merrettal. Tries for approximate string matching. *Knowledge and Data Engineering, IEEE*, pages 540–547, 1996.
14. C. Silverstein, H. Marais, M. Henzinger, and M. Moricz. Analysis of a very large web search engine query log. In *SIGIR 1999*, pages 6–12. ACM, 1999.
15. H. Wu, G. Li, C. Li, and L. Zhou. Seaform: search-as-you-type in forms. *VLDB 2010*, pages 1565–1568, 2010.
16. Y. Xie and D. O'Hallaron. Locality in search engine queries and its implications for caching. In *INFOCOM 2002, IEEE*, pages 1238–1247. IEEE, 2002.