Journal of Web Engineering, Vol. 0, No. 0 (2003) 000–000
© Rinton Press

TYPE-AHEAD EXPLORATORY SEARCH THROUGH TYPO AND WORD ORDER TOLERANT AUTOCOMPLETION

PAVLOS FAFALIOS and YANNIS TZITZIKAS
Institute of Computer Science, Foundation for Research and Technology - Hellas (FORTH), and Computer Science Department, University of Crete, GREECE
{fafalios,tzitzik}@ics.forth.gr

Received (received date)
Revised (revised date)

There is an increasing interest on recommending to the user instantly (during typing characters) queries and query results. This is evidenced by the emergence of several systems that offer such functionalities, e.g. *Google Instant Search* for Web searching or *Facebook Search* for social searching. In this paper we consider showing more rich recommendations that show several other kinds of *supplementary information* that provide the user with a better *overview* of the search space. This supplementary information can be the result of various tasks (e.g. textual clustering or entity mining of the top search results), may have very large size and may cost a lot to be derived. The instant presentation of these recommendations (as the user types a query letter-by-letter) helps the user (a) to quickly discover what is popular among other users, (b) to decide fast which (of the suggested) query completions to use, and (c) to decide what hits of the returned answer to inspect. In this paper we focus on making this feasible (scalable) and flexible. Regarding scalability we elaborate on an approach based on precomputed information and we comparatively evaluate various *trie-based* index structures for making real-time interaction feasible, even if the size of the available memory space is limited. Specifically, we show how with modest hardware (like this of a mobile device) one can provide instant access to large amounts of data. Moreover, we propose and experimentally evaluate an incremental procedure for updating the index. For improving the throughput that can be served we analyze and experimentally evaluate various policies for *caching subtries*. With regard to flexibility, in order to reduce user's effort and to increase the exploitation of the precomputed information, we elaborate on how the recommendations can tolerate *different word orders* and *spelling errors*, assuming the proposed trie-based index structures. The experimental results revealed that such functionality significantly increases the number of recommendations especially for queries that contain several words. Finally, we propose an algorithm for computing the top-K suggestions that exploits the ranking information in order to reduce the trie traversals. An experimental evaluation proves that the proposed algorithm highly improves the retrieval time.

Keywords: type-ahead search, instant search, exploratory search, autocompletion, query suggestions, caching
Communicated by: to be filled by the Editorial

1 Introduction

There is an increasing interest on recommending to the user instantly (after each keystroke) queries and query results. This is evidenced by the emergence of several systems that offer such functionalities. Indicative examples include *Google Instant Search* for plain Web searching, *Facebook Search* which shows the more relative friends, pages, etc. as the user types in a query letter by letter, and *IMDB*[a] *Search* which shows the more relevant movies, actors, etc. together with a photo. These instant recommendations, apart from enabling the user to see what is popular among other users, they allow 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 (reformulations of prior queries are very common in Web searching [1]). In general, we can say that the user adapts his query on the fly until the results match what he wants. Such services save users time and effort also because people type slowly but read quickly, implying that the user can scan a results page while he types. For example, according to Google, *Instant Search* can save 2-5 seconds per search.[b] Moreover, the emergence of *smart phones* and tablets enforces the need to reduce the typing effort since typing in the virtual keyboard of a smart phone or tablet is much more laborious than typing in a common keyboard.

Apart from showing on-the-fly only the first page of results of the guessed query (as in Instant Search), [2] has proposed showing more rich recommendations that show several other kinds of *supplementary information* that provide the user with a better *overview* of the search space (that work introduced the term `IOS` standing for *Instant Overview Search*). This supplementary information can be the result of various tasks, e.g. *textual clustering* [3] or *entity mining* [4] of the top search results. Note that this information may have very large size and/or may cost a lot to be derived. For example, according to [4], performing entity mining in the *contents* of the top-50 results (i.e. web pages) that are returned by submitting a query to a web search engine requires more that one minute and large amount of main memory. From the technical point of view, [2] showed that with *partitioned trie-based indexes* we can achieve instant responses even if the precomputed information is too large to fit in main memory. Furthermore, [5] demonstrates various novel applications of instant overview search that exploit and recommend different kinds of supplementary information. Figure 1 summarizes the key points of the proposed approach. The figure shows the possible steps of a (meta) search process. If the user's intention (which is described by some query terms) has been indexed with the proposed structure, then all steps which are enclosed by the dashed line (which may cost many seconds) can be bypassed according to the `IOS` approach; they are just loaded from the index.

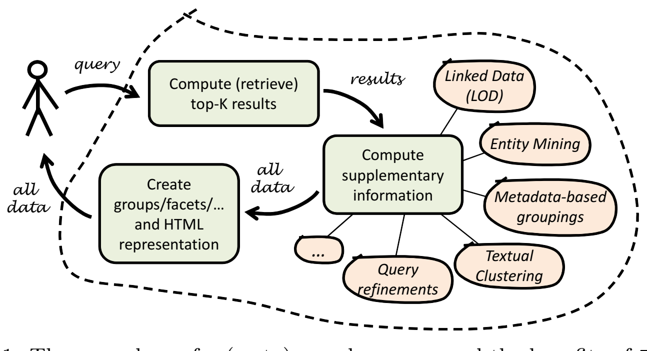


Fig. 1. The procedure of a (meta) search process and the benefits of `IOS`.

[a] `http://www.imdb.com/`
[b] `http://www.google.com/insidesearch/features/instant/about.html`

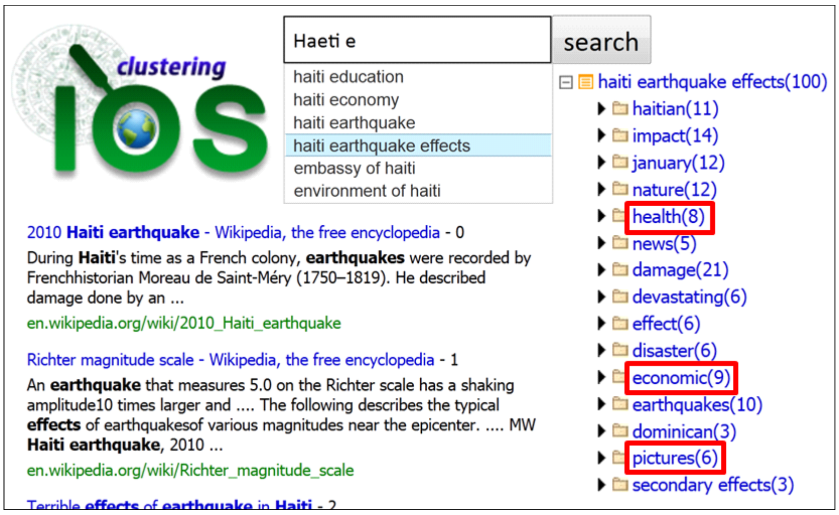
Regarding the value of the supplementary (overview) information to users in web searching, user studies [6] have shown that categorizing the search results improves the search speed and increases the accuracy of the selected results. Moreover, the user study in [7] showed that categories are successfully used as part of users' search habits. Specifically, users are able to access results that locate far in the rank order list and formulate simpler queries in order to find the needed results. In addition, the categories are beneficial when more than one result is needed like in an exploratory or undirected search task. Finally, according to [8] and [9], recall-oriented information can play an important role not only in understanding an information space, but also in helping users to select promising sub-topics for further exploration.

However, the instant provision of such services for large number of queries, big amounts of precomputed information and large number of *concurrent* users is challenging, especially in case we want to offer these services in a device with modest hardware (e.g. a smart phone or a tablet). In this paper we continue the line of research started with [2] and make the following contributions:

(a) We propose and evaluate various schemes for *caching subtries* (for Instant Overview Search) that exploit the available memory and, apart from speeding up these services, they increase the throughput that can be served.
(b) We show how the recommendation service can tolerate *different word orders*, *spelling errors* or *both*, assuming the proposed index structures (for reducing user's effort and increasing the exploitation of the precomputed information). Specifically, we analyze in detail several approaches and we reveal the *limits* for supporting these functionalities.
(c) We propose and evaluate an algorithm for retrieving the top-K suggestions that exploits the ranking information.
(d) We describe and evaluate an *incremental update* procedure of the index structures.

The motivation for focusing on scalability is obvious (without tackling this issue it is not feasible to provide such services in real applications), while the motivation for enhancing flexibility, and how it assists decision making, can be made evident from the following example.

Example 1: Consider a user that wants to write an article about the consequences (mainly the economic) of the earthquake that hit Haiti in 2010. For this reason he starts typing letter by letter the query `Haiti earthquake economic consequences`. With each keystroke, a list of suggestions appears instantly, which represents what is popular for the current input. Furthermore, for the top suggestion, the user is able to view the *top hits* and a *clustering* of the search space. After having typed the string `Haeti e` (notice that the word `Haiti` was misspelled), the user gets the suggestions `Haiti education`, `Haiti economy`, `Haiti earthquake`, `Haiti earthquake effects`, `Embassy of Haiti`, and `environment of Haiti`. Note that all these suggestions do not contain the user's input word `Haeti`, but the "similar" word `Haiti`. Furthermore the last two suggestions contain the word `Haiti` and a word that starts with "e" (`Embassy` and `environment`) in different word order than that of user's input. The user apprehends that the suggestion `Haiti earthquake effects` satisfies his search need. Thus, by pressing the down arrow key he navigates and selects the particular suggestion. At that time, the top hits of the selected suggestion and a clustering of the results appear *instantly* (note that user has not submitted the query). Afterwards, the user notices that there is a cluster with label economic(9), which actually contains 9 web pages that describe the economic damage of the earthquake in Haiti. The user clicks that label and instantly the results of this cluster appear on his screen. All these results are very likely relevant to his information need. Furthermore, the user notices that there are many other interesting cluster labels, such us health(8), nature(12) and pictures(6). So, he decides to include some pictures in his article and to write one more paragraph for the health problems that Haitian people faced because of the earthquake. Figure 2 depicts a screenshot of the aforementioned example. ⋄



Haeti e

haiti education
haiti economy
haiti earthquake
haiti earthquake effects
embassy of haiti
environment of haiti

2010 **Haiti earthquake** - Wikipedia, the free encyclopedia - 0
During **Haiti**'s time as a French colony, **earthquakes** were recorded by Frenchhistorian Moreau de Saint-Méry (1750–1819). He described damage done by an ...
en.wikipedia.org/wiki/2010_Haiti_earthquake

Richter magnitude scale - Wikipedia, the free encyclopedia - 1
An **earthquake** that measures 5.0 on the Richter scale has a shaking amplitude10 times larger and .... The following describes the typical **effects** of earthquakesof various magnitudes near the epicenter. .... MW **Haiti earthquake**, 2010 ...
en.wikipedia.org/wiki/Richter_magnitude_scale

Terrible **effects** of **earthquake** in **Haiti** - 2

haiti earthquake effects(100)
- haitian(11)
- impact(14)
- january(12)
- nature(12)
- health(8)
- news(5)
- damage(21)
- devastating(6)
- effect(6)
- disaster(6)
- economic(9)
- earthquakes(10)
- dominican(3)
- pictures(6)
- secondary effects(3)

Fig. 2. An indicative screenshot of IOS.

In general we can say that the instant presentation of such recommendations helps the user to quickly discover what is popular among other users (since the suggestions are based on log analysis), to decide fast which of the suggested query completions to use, and what hits of the returned answer to inspect. It is widely accepted that human decision making is a complex process [10] and this is true also in the context of information searching. In this context, i.e. during the search process, the user has to make two main kinds of decisions: the first concerns what query or queries to submit, the second is about what hits of the answer to inspect. We further analyze this process and discuss the impact of our work on decision making in Section 2.1.

The rest of this paper is organized as follows. Section 2 discusses decision making in the context of searching and related work. Section 3 introduces the trie-based index structures and details the main steps of trie construction and of an incremental method for refreshing them. Section 4 introduces the caching mechanism. Section 5 focuses on making *word-order independent* and *typo-tolerant* search feasible also over the proposed trie-based index structures. Section 6 discusses how to rank the (approximate) recommendations and proposes an algorithm for efficiently computing the top-K recommendations. Section 7 reports comparative experimental results for evaluating the index structures, the index update procedure and the caching schemes, demonstrating the feasibility of the flexibility features (and for quantifying their benefits in decision making), and finally for illustrating the effectiveness of the proposed top-K algorithm. Finally, Section 8 concludes and identifies issues that are worth further research. Extra material regarding benefits and speedup is given in the Appendix.

Various running prototypes that we have developed are available to use through `http://www.ics.forth.gr/isl/ios`.

2 Motivation and Related Work

In Section 2.1 we discuss the motivation from a decision making point of view, while in Section 2.2 we discuss related works from a technical point of view.

2.1 Human Decision Making during Web Searching

Human decision-making is defined as a cognitive process in which a preferred option or a course of action is chosen from among a set of alternatives based on certain information or considerations. In our context and during the search process, we can say that the user has to make two main kinds of decisions: *what query or queries to submit*, and *what hit(s) of the returned answers to inspect*.

With regard to what query to use, we should stress that there are several possible queries for expressing the same information need. This means that the user has to decide which one to type. The recommended query completions assist the user to *complete* what he has typed based on the popularity of the queries that other users have submitted in the past, and also enable him to identify possible *misspellings*. Furthermore, the supplementary information that is delivered instantly at each keystroke (corresponding either to a character or to an arrow button for shifting up and down in the list of suggested query completions) allows the user to *predict* the consequences of each choice and thus helps him to make a more "justified" and informed decision.

Concerning the second kind of decisions, i.e. what hits to inspect, the supplementary (overview) information can greatly assist him in locating hits that would be *very difficult and laborious to find* (i.e. those that are not in the first page of results). For instance, in the case of textual clustering of the top results [3], this can be achieved by clicking the label of a cluster that corresponds to a set of results that contain this label. Note also that users tend to look only the first page of results [11] not only because it is more time consuming and laborious to click, get and inspect the subsequent pages, but also because they feel that the hits in the first page are more reliable. As a consequence, users can be easily *biased* and they are vulnerable to web spamming as well as to the various search optimization techniques. We should also note that although search engines are programmed to rank web sites based on their popularity [12] and relevancy [13], empirical studies indicate various political, economic and social biases in the information they provide [14, 15]. Our thesis is that the provision of *overview information* alleviates also this problem since the users have the ability to see an interesting cluster label and by clicking on that label to end up to hits that could be low ranked (i.e. not in the first pages of results), practically to hits they would never inspect. Furthermore, the instant provision of supplementary information can assist the user in *refining*, or just differentiating, his information need. This can be realized either by focusing at a particular set of results (e.g. by clicking on a cluster name) or by starting the formulation of a new query. For instance, in our previous example (Example 1), although the user initially wanted to search for *economic effects*, the appearance of the supplementary information made him realize that there are also *health effects* and decide that he would like to get information about them.

Finally, we could say that in contrast to the typical recommender systems in which the available choices are usually less than millions (e.g. in the domain of movies, hotels, books, products), in the context of web searching the choices are billions. Moreover there are no fixed features (e.g. as in products), their number can be large and they are not homogeneous in an open environment like the Web (meaning that approaches like [16] cannot be applied).

2.2 Related Work

2.2.1 Search-as-you-Type

A Search-as-you-Type (also referred as Type-Ahead Search) system computes answers to keyword queries as the user types in keywords letter by letter. Specifically, at each keystroke of the user the browser sends (in AJAX style) the current string to the server, which in turn computes and returns to the browser a list of suggestions and the top hits of the top-suggested query. There are several works describing letter-by-letter searching for various kinds of sources, e.g. for relational databases [17, 18, 19] or documents [20, 21, 22].

Bast and Weber [20] study the problem of type-ahead search in documents and present a new indexing data structure (named HYB) which uses no more space than a compressed inverted index, and which can respond to autocompletion queries (as the user types in a query letter by letter) within a small fraction of a second. Ji et al. [17] formalize the problem of interactive fuzzy search on a *relational table* and propose various incremental search algorithms using previously precomputed and cached results in order to achieve an interactive speed. In particular, they use a trie to index the words in the relational table. Each leaf node of the trie has an inverted list of IDs of records that contain the corresponding word, with additional information such as the attribute in which the keyword appears and its position. Li et al. [21] study the problem of type-ahead search in XML data. Specifically, they use a trie to index the words of the XML data, and for each word they index not only the content nodes containing the word, but also those nodes whose descendants contain the word. Li et al. [18] study the problem of type-ahead search in relational tables and propose efficient index structures and algorithms for finding relevant answers on-the-fly by joining tuples in the database. They devise a partition-based method to improve query performance by grouping relevant tuples and pruning irrelevant tuples efficiently, and they also develop a technique to answer a query efficiently by predicting highly relevant complete queries. Wu et al. [19] propose a new search paradigm, called Seaform, to enable search-as-you-type in form-based interfaces. Seaform allows a user to specify keywords in multiple input boxes on a form and get the results instantly. Besides returning the matched records, Seaform also provide the matched attribute values as well. It incorporates trie-based index structures with inverted lists on the leaf nodes (together with two mapping tables) and search algorithms to support type-ahead search in multiple attributes. Li et al. [22] study how to answer top-k queries in type-ahead search, i.e. how to efficiently find the top-k best answers as a user types in a query letter by letter. In particular, their key question is how to access inverted lists on trie leaf nodes efficiently in order to answer top-k queries. In addition, they extend their techniques to support fuzzy type-ahead search which allows minor errors between the query keywords and the answers. Finally, Bast and Celikik [23] present new algorithms and index data structures for fuzzy type-ahead search that is robust against errors both on the side of the query and on the side of the documents. Specifically, they propose *fuzzy inverted lists* which correspond to a set of words and comprise the list of postings for that set of words.

However, the aforementioned works study the problem of finding and presenting instantly the documents/data that match some keywords (thus, their main competitor is the classical inverted index). On the contrary, we study the problem of finding and presenting instantly any related precomputed information regarding some keywords. Note that this information may have very large size and/or may cost a lot to be derived. Thus, to achieve very low response times and to avoid overloading the system, we index the corresponding information only for the (frequent) queries for which we want to offer type-ahead search, i.e. we do not index the underlying collection. The above approaches can be exploited in our problem only in case there is no suggestion for the query that the user is typing and in case we want to retrieve and present only the documents/data that match the query.

2.2.2 Caching

CompleteSearch [24], which uses HYB [20], caches the results of the query a user is typing excluding the last keyword. Ji et al. [17] also use cached results of earlier queries in order to achieve an interactive speed. Specifically, the idea is to cache only part of the results of a query. For subsequent queries, unfinished computation will be resumed if the previously cached results are not sufficient. Li et al. [21] cache the top-n nodes whose descendants contain a query's keyword. For each keyword, this approach first locates the corresponding node of the trie and then retrieves the top-n cached relevant elements. Li et al. [18] cache the trie vertices whose neighbors contain the keywords (as prefixes) and their corresponding Steiner trees. Seaform [19] caches the previous queries and their results, and when a query is submitted, the system first checks the cache to see whether the query can be answered from the cached results. Finally, Bast and Celikik [23] also use previously cached results in case an old query is a prefix of a newly typed query (while the user types the query letter by letter).

However, the aforementioned works cache previously computed results. On the contrary, we propose and evaluate various schemes for *caching subtries* (i.e. part of the index). We have not managed to find any work on type-ahead search that exploits the *available* main memory in order to cache a part of the index.

In addition, this problem resembles caching in web search engines. Specifically, a large body of work has been devoted to *results caching* and *posting lists caching*. Markatos [25] introduced caching query results as a means to reduce the response time of a search engine. Saraiva et al. [26] propose a dynamic caching system for caching query results and posting lists. Long and Suel [27] recommend a caching architecture that includes an intermediate level with on-disk caching of the intersections of the posting lists of frequently occurring pairs of terms. Fagni et al. [28] employ a hybrid results caching scheme where the available cache space is split into a static and a dynamic segment in order to capture both recent and frequent queries. Baeza-Yates et al. [29] explore the impact of results caching and posting lists caching in web search engines and show that posting lists caching offers higher hit ratios than results caching. Lempel and Moran [30] propose a cache replacement policy based on the probability of a result page being requested. Skobeltsyn et al. [31] propose an architecture that combines results caching with index pruning to reduce query processing load, while Gan and Suel [32] study weighted results caching techniques which consider both the frequency of the queries and their estimated execution costs. Finally, more recent works focus on the freshness of the cached results [33, 34].

Our case differs from the classic caching problem in the sense that we must offer *much faster* response time (of a few milliseconds) in order to present instantly the results and the supplementary information as the user types a query letter-by-letter (the query is not submitted). Furthermore, in our case we handle *bigger amounts* of information (e.g. the result of clustering or entity mining, images, etc.).

2.2.3 Typo-Tolerant and Word-Order Independent Query Suggestions

Ji et al. [17] study the problem of *fuzzy* type-ahead search, i.e. how to find records that include words similar to the keywords in the query even if they do not match exactly. Specifically, they have developed a caching-based algorithm for incrementally computing *active nodes* for a keyword as the user types it letter by letter. The leaf descendants of the active nodes are considered predicted keywords of the prefix. The idea is to use prefix-filtering: when a user types in one more letter, the previously computed active nodes can be used to compute the new active nodes. In case of queries with multiple words, they compute the active nodes of each word. However,this work indexes all the *words* of a relational table and finds the *records* that approximately match some query words. On the contrary, we index the frequent *queries* (that may have many words) and we find a set of *queries* that approximately match the input query (and for the top ranked query, we display the overview information). Nevertheless, we adopt a similar approach (i.e. we detect the active nodes) and we extend it in order to support both word-order independent and typo-tolerant query suggestions.

Chaudhuri and Kaushik [35] and Duan and Hsu [36] study the problem of online (while the user types a query letter by letter) spelling correction for query completions and both propose a trie-based approach. Chaudhuri and Kaushik [35] capture input typing errors via edit distance and propose two edit-tolerant autocompletion algorithms. The first is based on the state-of-the-art q-gram based edit distance matching algorithms and the second is a trie-based algorithm. We extend the trie-based algorithm in order to also support *word-order independent* typo-tolerant autocompletion. For example, if the user has typed in the partial query "mari**i**n mon" and the index contains the query "monroe mari**ly**n", the aforementioned work cannot suggest this query because the word order is different. Duan and Hsu [36] capture input typing errors via a n-gram transformation model and also adapt the A^* search algorithm with various pruning heuristics to dynamically expand the search space efficiency. However, as [35], this work cannot catch differences in the *word order*.

Li et al. [22] study the problem of fuzzy type-ahead search and use edit distance to measure the similarity between strings. They normalize the edit distance based on the query-word length in order to allow more errors for longer query keywords. However, as the work by Ji et al. [17], this approach indexes all the distinct *words* of the underlying collection and finds the *records* that approximately match some query words. On the contrary, we index the frequent *queries* and we find a set of *queries* that approximately match the input query.

Bast and Celikik [23] present new algorithms for fuzzy type-ahead search that is robust against errors both on the side of the query and on the side of the documents. As [17] and [22], this approach indexes the underlying collection and finds the *records* that approximately match some query words. However, it can also suggest full queries that approximately match the query that user is typing (based on the similar words that lie in the collection).

Finally, Hsu and Ottaviano [37] present three different *compressed* trie-based data structures that support efficient top-K completion queries and *briefly* discuss how their proposed approach can support fuzzy completions (however they do not test their assumptions).

To the best of our knowledge, our work is the first that elaborates on *word-order independent* suggestions. Note also that the above works consider that the index is not partitioned and also fits in main memory (only [23] proposes approaches in which the index may reside on disk). On the contrary, we elaborate also on *partitioned* trie-based indexes which are not in main memory, so we have to load the corresponding part of the index (i.e. subtrie) at real-time. In that case the instant behavior is more challenging. In addition, in each keystroke we have to locate and display bigger amounts of information.

2.2.4 Top-K Suggestions

Chaudhuri and Kaushik [35] shortly present how to extend the proposed algorithms in order to return only the top-K suggestions by exploiting a precomputed static score. In addition, Duan and Hsu [36] and Hsu and Ottaviano [37] apply a variation of the A^* search algorithm where each node in the trie is further annotated with the best score of all descendant queries.

We propose and experimentally evaluate a similar in spirit algorithm (appropriate for the proposed trie-based index structures) which however is based on a specific probabilistic ranking scheme. We also discuss how this algorithm can be adapted for supporting typo-tolerant and word-order independent top-K suggestions. A distinctive characteristic of the underlying ranking scheme is that the score of a node in the trie (that corresponds to a query suggestion q) is *by construction* bigger than the score of all its descendant nodes (i.e. suggestions that start with q), and expresses the probability that the user will select the corresponding suggestion after having typed a particular character sequence. This makes the evaluation of the top-K suggestions faster since we avoid traversing subtrees of big size. On the contrary, in [36] and [37] the underlying ranking scheme is not explicitly mentioned, meaning that (in some ranking schemes) a query q that is prefix of another query q' may has lower score than q', because for instance q' has more submissions than q. For example, for the partial query "`lio`", Google (for the time being) suggests "`lionel messi`" higher than "`lion`", although "`lion`" is prefix of "`lionel messi`". Consequently, by annotating each node in the trie with the best score of all descendant queries, we may eventually do not avoid traversing a big subtree. Thus, the underlying ranking scheme is very important in a top-K algorithm.

3 Trie-based Index Structures for Instant Overview Search

To tackle the requirements of `IOS`, we propose the adoption of trie-based index structures because they allow fast lookup with cost analogous to the length of the string that we search. Specifically, for each frequent query stored in the query log, we extend its node in the trie with two additional strings (as Figure 3 depicts). The first string contains the first page of results of the query, i.e. an HTML string containing for each hit its title, snippet and URL. The second string contains the supplementary information that we want to offer, for example the result of clustering or entity mining, etc. 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).

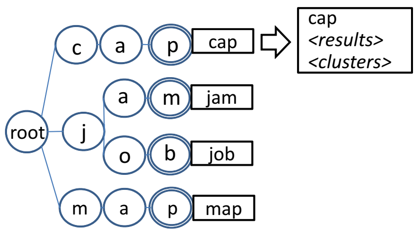


Fig. 3. Extending the query trie by two additional strings.

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

3.1 Trie Partitioning

The first idea is to adopt the *trie partitioning* method proposed in [38], i.e. to partition the enriched trie to a number of subtries based on the first character(s) of the queries. According to that method, only one "subtrie" of much smaller size is loaded in main memory for serving one request; the subtrie containing the queries whose starting characters are those that the user has typed so far. In particular, if $\mathcal{Q}$ is the set of all distinct queries in the log, let QT be the full trie over $\mathcal{Q}$ and $QSubtries = \{qst_1, \ldots, qst_N\}$ be the partition of QT to N subtries according to [38] which is based on the first k characters. Each subtrie qst_i contains a set of queries $queries(qst_i)$ whose starting characters belong to a specified set of characters. For example, if we assume that the query log contains queries starting with latin characters only and we decide to partition the trie based on the first character ($k = 1$), then we can divide a trie into two subtries: one containing all query strings q where $q[0] \in \{a, b, \ldots, m\}$ and another containing all q where $q[0] \in \{n, o, \ldots, z\}$. Note also that for $k \geq 2$, the distribution of queries to subtries is very smooth, close to the ideal (for more information about this approach the reader can refer to the corresponding paper [38]).

Since users seldom change their initial queries [39], 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.

3.2 Indexes to External Random Access Files

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. first hits and cluster label tree) in one or more different files. Consequently, for each query entry the trie 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 precomputed information. This approach greatly reduces the size of the trie, however we have to perform additional disk accesses for reading the pointing file, but only for that query whose supplementary information has to be displayed to the user's screen.

3.3 Combining Trie Partitioning and Indexes to External Files

We can 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 indexes 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 above choices, [2] comparatively evaluated the following approaches (depicted in Figure 4):

(a) **(SET) Single Enriched Trie.** Here each node of the query trie is extended with two strings: one for the first page of results and one for the supplementary information (e.g. results clustering). The entire enriched trie is kept in main memory (Figure 4a).

(b) **(STIE) Single Trie with Indices to External files**. Here each node of the query trie is enriched with pointers to external random access files, where a pointer consists of three numbers: file number, bytes to skip, bytes to read (Figure 4b). The single (query) trie is kept in main memory

(c) **(PET) Partitioned Enriched Trie.** Here the *enriched* trie, i.e. the one described in (a), is partitioned to several subtries, each stored in a separate file. The subtrie that is needed in order to serve a request is loaded in main memory at *request time.*

(d) **(PTIE) Partitioned Trie with Indices to External files.** Here the (query) trie is partitioned but each subtrie is enriched with pointers to external random access files (Figure 4d). Here also the subtrie is loaded in main memory at *request time.*

A synopsis of the experimental results is given in Section 7.1 (for reasons of self-containedness).

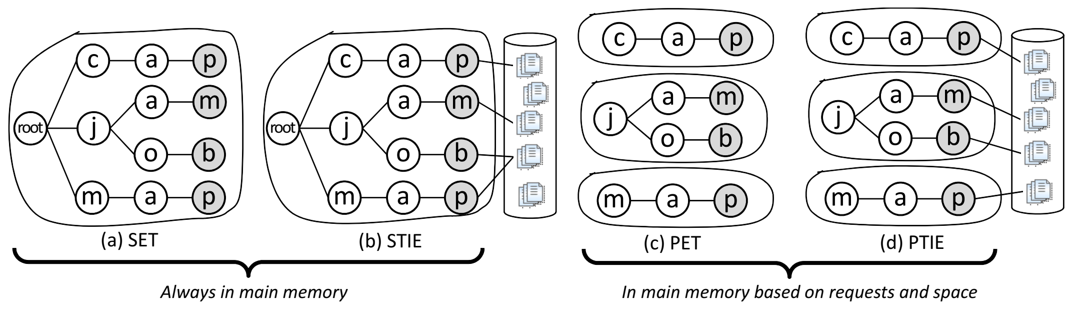


Fig. 4. Trie-based index structures for Instant Overview Search.

3.4 Index Construction and Update

For the construction of the trie (or subtries), the main tasks are (a) the analysis of the query log, (b) the execution of each distinct query in order to get the first page of results and the supplementary information (e.g. by batch query processing [40]), and (c) the creation of the trie file (or of the subtries). We precompute and store also the score of each distinct query, which is used by the autocompletion algorithm. Specifically, we use the method proposed in [38] that takes into account the frequency of the query and the frequencies of the queries that have this as prefix (the reader can find more information about the particular scoring function in Section 6). One can easily see that the time required by these tasks depends on the particular algorithms that are employed (for query evaluation, clustering, etc).

Note however 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 web search engine. One policy is to update the index daily. Since the construction of the trie from scratch is a time consuming task [2], it is worth providing an *incremental* method. We propose the following incremental update procedure: for each distinct query of the new query log we check if it exists in the index. If so, we just update i) its results, ii) its supplementary information, iii) its date and iv) its score. Since the proposed scoring method takes into account the frequency of the query and the frequencies of the queries that have this as prefix, we must update also the scores of its ancestor queries (since the score of a node in the trie is always bigger than the scores of its descendant nodes). If the query does not exist in the index we add it with all its data and update the scores of its ancestor queries.

The procedure is almost the same in case of STIE. While adding a new query in the index, we could either store its results and its supplementary information in a new random access file or append them in an existing one. Moreover, for updating the results and the supplementary information of an existing query, it is better to add the new data in a new random access file or append them at the end of an existing one and then update in the trie its file number, its bytes to skip and its bytes to read (otherwise we have to change the values of these three numbers for all the queries that follow that query in the external file).

In case we adopt a partitioned index structure (PET or PTIE) the update is a bit more complex. If the query exists in a subtrie, we have to detect the corresponding subtrie, load it and then follow the same procedure as in SET and STIE. If it does not exist, we check if a subtrie of its top-k characters exists (where k is the number of the first characters that determine trie partitioning [38]). If so, we add it in that (existing) subtrie, otherwise we add it in the subtrie with the smaller size (i.e. the subtrie that contains the fewer queries).

The major problem while updating each one of the four indexing approaches is that the size of the trie (or of a subtrie) may grow very much. In that case and in all approaches we can keep stable the size of the (sub)trie by removing the "old" queries according to a date threshold. This can be done by performing a trie traversal and removing those queries that are chronologically before the date threshold. Moreover, for each query that we remove, we have to update the scores of its ancestor queries. Alternatively, we can change the indexing approach to one with lower memory requirements. For example, if the size of SET has grown very much, we can adopt STIE, PET or PTIE. Respectively, in case of STIE and PET we can adopt PTIE. Furthermore, in case of PET and PTIE, we can split the one "big" subtrie to two different subtries or increase the number of the first characters that determine trie partitioning (i.e. the value of k).

Finally, the incremental update can be performed much faster if in each query submission the system stores the results and the supplementary information. This allows the system to use these results (although they are not the most "fresh") during the update of the trie, saving thereby considerable time.

In Section 7.1 we report indicative experimental results regarding the aforementioned incremental update procedure.

4 Throughput and Caching

Consider a scenario where a large number of users start typing queries at the same time. A rising question is how each index structure (either SET, STIE, PET or PTIE) reacts and in what cases the system will get overloaded. In the SET and STIE approaches, the trie is loaded only once at system's start-up and therefore, the number of the requests that the system can serve depends only on the server's request/session capacity. Note that in the STIE approach the usage of random access files limits its effect on main memory (there is only a small effect of buffer cache interaction). However, PET and PTIE require loading multiple subtries, i.e. the appropriate subtrie for each user's keystroke, and this could overload the system since multiple subtries will have to be hosted at main memory. This is true especially for PET whose average subtrie size is bigger than that of PTIE (because the subtries of PET contain the supplementary information, while those of PTIE do not).

The main question is how to exploit the available main memory in order to serve concurrently several users as fast as possible without overloading the system. Is it better to load in memory the appropriate subtrie at request time and remove it afterwards, or is it better to keep it in memory for a period of time? One general method to alleviate the throughput problem is to adopt a *caching mechanism*. Thereby, if the requests of two or more users require loading the same subtrie, the loading will be done only once. Obviously, the system has to look the cache before loading a subtrie.

4.1 Caching Schemes

One approach is to adopt a *dynamic* caching scheme, i.e. to start from an empty cache and put in it each requested subtrie. If a new subtrie has to be loaded and the cache is full, the system replaces an existing cached subtrie (e.g. the less frequent one that is not in use by a session) with the new one. It could also refresh the cache by removing the old subtries, i.e. the subtries that are not in use for a specific time threshold. In such a *dynamic* caching mechanism, the cache size can be set to be equal to the maximum number of loadable subtries that can fit in main memory at a given time. In a *static* approach, we could keep in cache the more frequently used subtries based on a past log analysis. Figure 5 shows the distribution of the top-20 first two characters in queries stored in a real query log of 1 million queries (submitted to Altavista[c] web search engine in 2003). We observe that the prefix "ho" is the most frequent prefix appearing in 2,69% of all queries. Moreover, the top-20 prefixes of size 2 appear in the 30% of all queries and this means that if we keep cached these subtries we expect a cache hit ratio of about 30%. Taking into account the above facts, we could keep always in memory the subtries of the most frequent first two characters. In an *hybrid* caching mechanism, we combine dynamic and static approach. In this way, we keep always in memory the subtries of the most frequent queries (static part) and keep an amount of memory for loading the subtries that are not in the cache (dynamic part). For example, if the available memory capacity can host MC subtries, we can split it into two parts: a static (with size S) and a dynamic (with size $D = MC - S$).

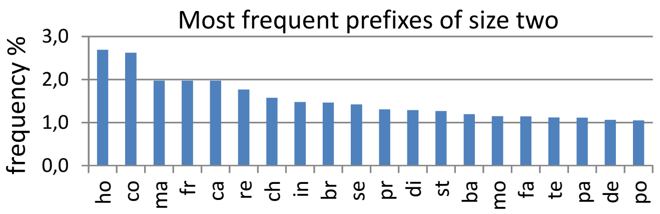


Fig. 5. Distributing queries by first two characters.

Algorithm 1 describes the above hybrid caching mechanism. *StaticMap* is a map for the static part of the cache and keeps information for each cached subtrie (e.g. frequency, last used time, subtrie's file, etc), while *DynMap* is the corresponding map for the dynamic

[c]http://en.wikipedia.org/wiki/AltaVista

part. We observe that in case we have a request for a subtrie that lies in the static part (line 3), the system can instantly retrieve the required data, otherwise it must check the dynamic part (line 5). In the dynamic part, if we have a request for a subtrie that is not in the cache (line 10) and the cache is full (line 15), we find (line 16) and remove (lines 18, 19) the less frequent subtrie that is not in use according to a time threshold TT, and then load the new subtrie (lines 20, 21). The function *refreshLastUsedTime(Trie subtrie)* sets the last used time of the corresponding subtrie to the current requested time, the function *refreshFrequency(Trie subtrie)* increases the frequency of the specific subtrie by one unit, and the function *getWorst(TT)* finds the less frequent subtrie that is not in use according to the time threshold TT. Finally, in case the dynamic cache is full and there is not a subtrie that is not in use (line 23), the system cannot serve instantly the user and thus the user must submit the query.

Algorithm 1 A hybrid caching mechanism.

```
Input: Current query $q$
Output: A subtrie's node which contains the required data
 1: $st \leftarrow$ subtrie($q$) //for the current query $q$, find the corresponding subtrie
 2: $S = StaticMap$.size()
 3: if $StaticMap$.contains($st$) then
 4:   return $StaticMap$.getNode($st$,$q$)
 5: else //the requested subtrie is not in the static part
 6:   if $DynMap$.contains($st$) then
 7:     refreshLastUsedTime($st$)
 8:     refreshFrequency($st$)
 9:     return $DynMap$.getNode($st$,$q$)
10:   else //the requested subtrie is not in the dynamic part
11:     if $DynMap$.size() $< MC - S$ then
12:       load($st$, frequency=1)
13:       $DynMap$.add($st$)
14:       return $DynMap$.getNode($st$,$q$)
15:     else //dynamic cache is full, remove the less frequent subtrie that is not in use
16:       $s \leftarrow DynMap$.getWorst($TT$)
17:       if $s$ then
18:         unload($s$)
19:         $DynMap$.remove($s$)
20:         load($st$, frequency=1)
21:         $DynMap$.add($st$)
22:         return $DynMap$.getNode($st$,$q$)
23:       else //the system cannot serve instantly the user, user has to submit a full query
24:         print Memory is full and in use! Please submit the query!
25:         return NULL
26:       end if
27:     end if
28:   end if
29: end if
```

The problem of deciding how to partition the available memory in a static and a dynamic part depends on the characteristics of the expected workload and the intended behavior of the system. In general we could say that it is a good choice to have a dynamic part in order to offer fast responses for recent and frequent queries and adapt to emerging temporal trends (e.g. breaking news about an earthquake will cause a significant number of new queries on the subject). In [25, 41, 42], the authors illustrate that the majority of the repeated queries are referenced again within short time intervals, i.e. queries have significant temporal locality. Nevertheless, there remains an important portion of queries that are repeated within relatively longer time intervals. Lempel and Moran [30] observed that query popularity follows an inverse power-law distribution and that the most popular queries submitted to search engines do not change very frequently. This is a strong rationale for introducing a static part. But which is the better way to partition the memory? Markatos [25] mentions that although dynamic caching (most recently accessed queries) can use large caches more effectively, static caching (caching of the most popular queries) can perform better for (very) small caches. Furthermore, the experiments in [28] illustrate that for all the replacement policies, the best way to partition the memory is to give more space to the static part. Specifically, giving 60% to 80% of the memory to the static part, we can achieve the highest hit ratio.

In Section 7.2, we experimentally evaluate various caching policies for our setting in order to reveal the most appropriate for our case.

5 On "Flexible" Recommendations

Even if we use a very large query log, our index may not contain queries that start with a particular user's input string. In that case, the system cannot suggest any queries and overview information. However, our index may contain a query that is *very close* to what the user has typed. For instance, a query in the index may contain all the words of user's input in *different order*, or the user may have typed a word with *typo(s)*. It would be nice if in such cases the system could avoid returning an empty list of suggestions. Below in Section 5.1 we discuss how we can tolerate different *word orders*, in Section 5.2 we elaborate on *typo-tolerant* search, while in Section 5.3 we detail how we can efficiently exploit *both* of the above functionalities. Note that we target on offering these functionalities assuming the proposed index structures and without building new indexes.

5.1 On Relaxing the Word Order

Consider a user that starts typing `avensis toyota`. Suppose that the query trie (or subtrie) contains the query `toyota avensis` but not the query `avensis toyota`. After having typed `avensis`, no `toyota` is suggested however the user types "t" hoping that `toyota` will be suggested. Suppose that the trie does not contain any completion for `avensis t`, so the system cannot offer any suggestion.

To tackle this problem the system can also load the suggestions starting from "t". Moreover, the system can exploit the fact that the user has already typed `avensis`. For this reason, instead of suggesting the top rating completions for the prefix "t", it should search the trie for finding (and ranking) all completions that start from "t" and include the word `avensis` as second or third (and so on) word. This requires traversing the (sub)trie after "t" (which in any case is traversed for ranking the possible query suggestions) and keeping (scoring with a non zero value) only those completions that contain `avensis`. This may yield suggestions like `toyota avensis`, `technical characteristics avensis`, `test drive avensis`, etc. If the user does not select any of the above and instead types "i" this process will be repeated leading to suggestions like `tires avensis`, etc. Below we discuss the most common case of two-word queries and then approaches for offering this functionality in case of queries with more than two words.

Two-Word Queries. For two-word queries (which is the majority of queries in web search engines) the system can check both orders (at the time the user starts typing the second word). During looking up the permuted query, the system exploits the knowledge of the first word to restrict the possible suggestions. Finally the system ranks the suggestions yielded by each word order and derives a single list of recommended query suggestions. Regarding efficiency, if the two permutations belong to the same trie, then only one more subtrie traversal is needed (in order to collect the suggestions that start with the second word). Furthermore, the system has to perform $|Q_w|$ string comparisons (where Q_w is the set of queries that start with the character sequence w), in order to check if the queries that start with the second word (or part of word if user has not finished typing) contain the first word. The cost for checking if a string is contained in an other string has complexity $O(n)$, where n is the length of the biggest string (Boyer-Moore string search algorithm [43]). However, if the two permutations do not belong to the same subtrie (in case of PET and PTIE), the system has to load the new required subtrie which costs a bit more if the particular subtrie is not in the cache (in case we adopt a caching mechanism).

Queries with Many Words. The case of queries with more than two words is more expensive since a m-words query has $m!$ possible permutations. However we should note that long queries are not frequent. Nevertheless, a system can adopt one of the following approaches:

(a) `(ALL_PERMUT)` *Check all possible* $m!$ *permutations* (where m is the number of words of user's input string). This approach is very expensive (especially for queries with many words), as it requires $m!$ trie traversals.

(b) `(WORD_CONTAIN)` *Check for queries that start from a word of user's input string and contain at least one of the remaining words.* The more of the remaining words the query contains, the higher rank it receives. In this approach, a query in the index does not have to contain all the words of user's input in order to be added in suggestions list, limiting thereby the probability of returning an empty list. For a m-word query, this approach has to perform m trie traversals and $\sum_{i=1}^{m} |Q_i|(m-1)$ string comparisons, where Q_i is the set of queries that start with the i-th word of the query. Algorithm 2 describes this approach. For each word of user's input string (line 3), we find all the queries of the log that start with this word (line 5). Then for each query (line 6), we check if it contains one or more of input's words (lines 8, 9). The more of the remaining words the query contains, the higher rank it receives (line 10). Note that this rank is not the final rank of the suggestion. It actually represents the number of words in query that are contained in each suggestion and can be taken into account in the final ranking of both normal and different word-order suggestions. Finally, if it contains at least one word, we add this query to the list of suggestions (lines 11, 12).

Table 1 summarizes the two approaches. The second approach requires few trie traversals (proportional to the number of query's distinct words), however it has to perform many string comparisons. Nevertheless, it limits the probability of returning an empty list of suggestions

Algorithm 2 Finding suggestions without considering the word order.

Input: Current user's input q, the query log *trie*
Output: A list of *suggestions* each accompanied by its *score* which is actually the number of words in q that are contained in each suggestion.

```
1: $words \leftarrow$ getWords($q$) //for the current input $q$, find and return its distinct words
2: $suggestions \leftarrow$new $list()$
3: for all $word$ in $words$ do
4:     $trie$.descendTo($word$) //$trie$ traversal
5:     $queries \leftarrow trie$.findDescendantQueries() //get queries starting with $word$
6:     for all $query$ in $queries$ do
7:         $score = 0$; //initialization of suggestion's score
8:         for all $cur\_word$ in $words - \{word\}$ do
9:             if $query$.contains($cur\_word$) then
10:                $score$++
11:            end if
12:        end for
13:        if $score > 0$ then //the query contains at least one of input's words
14:            $suggestions$.add($query$, $score$)
15:        end if
16:    end for
17: end for
18: return $suggestions$
```

Table 1: Synopsis of the implementation approaches of word-order independent suggestions.

Cost and Functionality	`ALL_PERMUT`	`WORD_CONTAIN`
Num of (sub)trie traversals	$m!$	m
Num of string comparisons	$\varnothing$	$\sum_{i=1}^{m} \lvert Q_i \rvert (m-1)$
Max number of subtrie loadings (for PTIE)	m	m
Suggestion must contains all the words of user's input	`YES`	`NO`

Note: m is the number of words of user's input and Q_i is the set of queries that start with the i-th word of the query.

(a suggestion does not have to contain all the words of user's input). In addition, in both approaches and in case we adopt PET or PTIE, the system has to load up to m distinct subtries (one subtrie for each distinct word of user's input) and the user must type at least k letters of the last word in order to check also for queries that start with this word (where k is the number of the first characters that determine trie partitioning).

Incremental Suggestions. Note that while the user is typing a word, if the new input is part of the old input, i.e. user has not changed what he has already typed (which is the common case), then the new suggestions are subset of the previous suggestions. For example, if the user has typed `toy` and for this input he gets the suggestions `toyota`, `toyota avensis`, `toyota corolla`, `toys` and `toy story`, then by pressing "o" he gets the suggestions `toyota`, `toyota avensis` and `toyota corolla` which are a subset of the previous set of suggestions. Thereby, once the system has retrieved a list of suggestions for a particular word and user has not changed what he has already typed, we can avoid trie traversals and consequently subtrie loadings in case of PET and PTIE. This is also important in word-order independent search for queries with more than one word, in particular for the second approach (`WORD_CONTAIN`). For example, if user has typed `toyota` and for this input he gets the suggestions `toyota`, `toyota avensis` and `toyota corolla` then by continue typing `toyota av`, we can first filter the existing suggestions by removing those that do not contain `av`, and then traverse the trie for getting the new suggestions that start with `av` and contain `toyota`. Once we have retrieved the new suggestions and user continues typing the second word, we just filter the new suggestions according to what user is typing, i.e. there is no need for new trie traversals. Consequently, in this case the costs in Table 1 do not represent the costs of one user's keystroke, but the total costs of the whole query (e.g. user copies and pastes his query). However, if user types his query too fast or changes his initial input (by pressing the backspace button or using the mouse), we cannot exploit the incremental suggestions.

Synopsis. Relaxing the word order of the suggestion system offers two benefits, one for the user's side and one for the server's side. Obviously, user's satisfaction grows since the system returns suggestions that it would not find without this functionality. On the other hand, the system does not need to index queries that contain exactly the same words, reducing in this way the size of the trie. However, if we cannot exploit the incremental approach, it increases system's response time since the system has to perform more trie traversals, many string comparisons and maybe to load and access other subtries (in case of PET and PTIE). In Section 7.3 we report experimental results that demonstrate the feasibility of this functionality regarding the proposed index structures.

5.2 Typo-Tolerant Search

Now we describe approaches for offering typo-tolerant query suggestions and the challenges that arise by adopting them in the proposed trie-based indexes.

Motivation. Consider a user typing a query who is not sure for the spelling of a word, e.g. he types `merilyn`, but actually he would like to type `marilyn`. Suppose that the query trie (or subtrie) contains the queries `marilyn`, `marilyn monroe` and `marilyn manson`, but not the query `merilyn`. The user (after having typed the first 2 characters) will never get the suggestion `marilyn` and consequently the other two suggestions `marilyn monroe` and `marilyn manson`. Note that according to Cucerzan and Brill [44], more than 10% of search engines queries are misspelled. This is even more severe for tail queries of which more than 20% are misspelled [45]. Typos occur for a variety of reasons. For example, when typing quickly, users may add or drop letters unintentionally. Moreover, accidentally hitting an adjacent key on the keyboard, also known as the fat-finger syndrome, is common especially on mobile devices with small virtual keyboards.

Implementation Approaches. (a) (`DETECT_ALL`) One approach is to load also the suggestions that their *beginning substring* is "similar" to the query that the user is typing. For example, the system can compute the Edit (Levenshtein) distance [46] between user input and the beginning substring of each full query in the log.[d] If this number is lower than a threshold,

[d]Note that the complexity of a common edit distance algorithm, e.g. a dynamic programming algorithm, is $O(nm)$, where n and m are the strings length (in our case m is the length of the user query and n is the sum of the lengths of all queries in the query log).

the system can add the corresponding queries to the list of suggestions and rank them as if no edit distance was employed. Specifically, in each user's keystroke we can detect the *active* nodes of the trie. A node is considered *active* if the edit distance between its corresponding string and user's input is lower than a threshold. For example, in the trie of Figure 6, for the user input `meri` and *edit distance* = 1, the active nodes are the nodes that correspond to the character sequences `ceri` and `mari`. Thereby, the system can suggest the queries `cerise`, `cerium`, `maria`, and `marilyn`. For choosing the right edit distance threshold, we can take into account the length of user's current input. For instance, we can allow one edit operation (insertion, deletion or substitution) per three characters, i.e. $threshold = input\_length/3$. Moreover, we start searching for similar queries when user has typed at least the third character (it has no sense to find typos if user has typed too few characters).

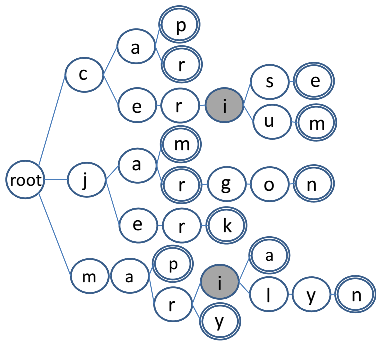


Fig. 6. Trie's active nodes for input `meri` and edit distance = 1.

(b) (`IGNORE_FIRST`) Another approach is to *ignore possible typo in the first character* of user query. Note that if we want to find suggestions including typo in the first character, the system has to compute the edit distance between the user input and the beginning substring of many queries in the log (in order to find the active nodes). Specifically, in the worst case the system has to visit the nodes that their corresponding string has maximum length equal to the input length plus the edit distance threshold. For example, for the input "meri" and in case we allow one edit operation per three characters the system has to check the nodes with maximum string length equal to five characters (i.e. the nodes of level ≤ 5). However, this costs a lot and is proportional to the total number of distinct queries that lie in the index. If we would like to handle this efficiently we would have to create a character-based suffix tree of all queries in the log but that would increase the space requirements. Furthermore, in case of PET and PTIE, the system has to load and access all the subtries, overloading thereby the system.

Thus, one method to reduce this cost is to find the active nodes only for the part of the trie that contains queries starting with the letter that the user input starts (e.g. if user has typed `merilyn`, find all similar queries that start with "m"). For SET and STIE, this requires one more trie traversal of the part of the trie that its queries start with a specific character (in order to detect the active nodes), and the computation of the edit distance between the corresponding string of each node and user's current input. This costs about $|N_c|(\frac{n+edt}{2})n$, where n is the length of the user input string, edt is the edit distance threshold for the current input length, and N_c is the set of all nodes of the trie with level less than or equal to $n + edt$

Table 2: Synopsis of the implementation approaches of typo-tolerant suggestions.

Cost	`DETECT_ALL`	`IGNORE_FIRST`
Num of (sub)trie traversals	Traverses a part of the trie (of level $\leq n + edt$)	Traverses a part of the trie (of level $\leq n + edt$, starting with input's first char)
Num of edit distance computations	$\lvert N\rvert(\frac{n+edt}{2})n$	$\lvert N_c\rvert(\frac{n+edt}{2})n$
Max number of subtrie loadings (for PET and PTIE)	All the subtries	Depends on k

Note: k is the number of the first characters that determine trie partitioning, n is the length of user's input string, edt is the edit distance threshold for the current input length, N is the set of all nodes of the trie with level $\leq n + edt$, and N_c is the set of all nodes of the trie with level $\leq n + edt$ that their corresponding string starts with input's first character.

that their corresponding string starts with input's first character. In PET and PTIE and for $k > 1$ (where k is the number of the first characters that determine trie partitioning), the system may need to access more subtries. For example, for input "merilyn" and $k = 2$, the system has to access the subtries that correspond to the character sequences "ma", "mb", "mc", etc. Thus, in this case we must check only the subtries that lie in the cache.

Table 2 summarizes the approaches for typo-tolerant search. In comparison to the general approach, ignoring typo in the first character reduces the trie traversals, the edit distance computations and the maximum number of subtrie loadings (for PET and PTIE). Especially for PET and PTIE, we must always ignore typo in the first character and traverse only the subtries that lie in the cache (for $k > 1$). In Section 7.3 we report indicative results regarding this functionality.

5.3 Combining Typos and Different Word Orders

Here we elaborate on how we can support *both* typo-tolerant and word-order independent suggestions.

Consider a user typing the query `monroe merilyn` and suppose that our index does not contain any query that starts with `monroe`, however it contains the queries `marilyn monroe` and `marilyn monroe filmography`. Unfortunately these queries will not be suggested neither by typo-tolerant nor by word-order independent search. Below we describe two possible approaches.

(a) (`DETECT_ALL`) To tackle this problem the system can detect the *active nodes* of *each word* and then restrict the possible suggestions exploiting the knowledge of the remaining words. Specifically, we can use the incremental approach for catching different word orders. If the new input is part of the old input and if the user continues typing the same word (i.e. no space character is detected) then we just update the active nodes of the last word. When we detect that user is typing a new word, we "lock" the active nodes of the previous words (i.e. there is no need to update anymore their active nodes) and try to find the active nodes of only the new word. Furthermore, we filter the suggestions that derive by each active node to those that contain at least one of the remaining words or one of the corresponding strings of the active nodes of each remaining word (in order to catch possible typos). In this way we can capture both different word orders and typos.

In each keystroke, the computation of the active nodes for a word (or a part of a word if the user has not finished typing it) costs about $|N|(\frac{w+edt}{2})w$, where w is the length of the word, edt is the edit distance threshold for the current word length, and N is the set of all nodes of the trie with level less than or equal to $w + edt$. If the user has changed his initial input (or he starts typing very fast), then we must find the active nodes of each word (which costs $m|N|(\frac{w+edt}{2})w$, where m is the number of words of user's input). Furthermore, the filtering of the suggestions that derive from each active node to those that contain at least one of the remaining words or one of the corresponding strings of the active nodes of each remaining word, requires $\sum_{i=1}^{l} |A_i|(l-1)$ string comparisons, where l is the total number of active nodes and A_i is the set of suggestions that derive from the i-th active node.

(b) (`IGNORE_FIRST`) To improve efficiency, we can ignore typo in the first character, as we saw at the second approach of typo-tolerant search. Obviously, this approach reduces the trie traversals, the edit distance computations and the number of subtrie loadings for PET and PTIE.

6 On Ranking the (Approximate) Recommendations

The ranking of recommendations is important since only a small number of the possible completions are prompted.

Various ranking methods were introduced and analyzed in [38]. One of the methods that was proved effective (and has a clear probabilistic interpretation) is described next. Let $q \rightsquigarrow q'$ denote that q is a prefix of q'. Let q_u denote the query the user has typed. We assign a score to each candidate completion 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 based on the log file and it is defined as:

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

where $DeepFreq(q) = freq(q) + \sum_{q \rightsquigarrow q'} freq(q')$, and $freq(q)$ is the frequency of q in the query log. We have to stress that this ranking does not affect the instant behavior because we precompute the scores of all queries and we store them in the (sub)tries.

6.1 Exploiting Ranking Information for Efficiently Computing the Top-K Suggestions

We can exploit the ranking information for reducing the (sub)trie traversals. For instance, suppose that we are interesting in suggesting the top-K (e.g. $K = 10$) queries. According to the aforementioned ranking method, if $q \rightsquigarrow q'$ then $Score(q) \geq Score(q')$. Actually the score of a query suggestion q corresponds to the probability that the user will select q after after having typed a particular character sequence. In that case, we can keep an array with the K highest scores of completions encountered so far, and before continuing a trie traversal from a node n, we check if the score of n is greater than the minimum score stored in the array. Only in that case it is worth proceeding to the children of n. This is actually an adaptation of the *Threshold Algorithm* [47] for our setting. This optimization is especially useful for the case of word order and typo tolerant recommendations which require making several (sub)trie traversals.

Algorithm 3 describes the procedure for computing the top-K suggestions. We first descend to the tree node that corresponds to the input string (line 1) and start traversing the subtree under that node (line 3). If the score of the current node is lower that the minimum score of the suggestions in the top-K array (line 4) we ignore the subtree under that node and continue the (sub)trie traversal (line 5). Note that in each node (even if it does not contain a complete query) we have stored the maximum score of the suggestions that are under that node. If the score is bigger than the minimum score (line 6) we check if the node contains a complete query, i.e it corresponds to a suggestion (line 7). If so, we check if the array with the top-K suggestions is full (line 8). If not, we just add the corresponding suggestion to the array (line 9), otherwise (line 10) we have to remove the suggestion with the minimum score (line 11), add the new suggestion (line 12), and finally assign the new minimum score (line 13).

Algorithm 3 Efficiently computing the top-K suggestions.

```
Input: Current input $q$, the query log $trie$, $K$ desired number of suggestions.
Output: The top-K suggestions ($topKarray$).
1: $trie$.descendTo($q$)
2: $minScore = -1$
3: for all $node$ in depthFirstTrieTraversal($trie$) do //start traversing the descendant nodes
4:     if $node$.getRank $<$ $minScore$ then //the node has lower score than the min score
5:         continue //ignore the subtree under $node$ and continue
6:     else
7:         if $node$.hasCompleteQuery() then //the node corresponds to a suggestion
8:             if $topKarray$.size() $<$ $K$ then //the top-K array is not full
9:                 $topKarray$.add($node$.getQuery())
10:            else //the top-K array is full
11:                $topKarray$.removeMinSuggestion() //remove suggestion with min score
12:                $topKarray$.add($node$.getQuery()) //add the new suggestion
13:                $minScore$ = $topKarray$.getMinScore() //assign the new minimum score
14:            end if
15:        end if
16:    end if
17: end for
18: return $topKarray$
```

The procedure is almost the same in case we want to offer word order independent and typo tolerant suggestions. Regarding the former, we descend to each *distinct word* of user's input string and perform a trie traversal ignoring the subtrees under the nodes with score lower than the minimum score of the suggestions in the top-K array. Regarding the latter, we follow exactly the same procedure for each *active node*.

In case we adopt a partitioned index (PET or PTIE) we can avoid loading a subtrie if we store (and keep in memory) the maximum score of the suggestions that it contains. Thereby, we can avoid loading a subtrie with low ranked suggestions. Moreover, in case of word order independent and typo tolerant suggestions, we can start traversing the subtries with the higher rank and stop when we have filled the top-K array, saving thereby considerable time.

Supporting a Time Threshold Policy. A search system may also have a *"maximum time" policy* for computing the top-K suggestions (e.g. 100 ms). In that case, we have to start measuring the time when the algorithm starts and return the top-K array if we pass a given time threshold. In case we want to stop retrieving suggestions (because we have not more time), we have two cases: (i) only one subtree traversal remains, (ii) n subtree traversals remain. In the first case we know which of the top suggestions are surely the top (those with score greater than the score of the remaining node). In the second case, the top suggestions are surely the top if their score is greater than the max score of the n remaining nodes. Thereby we are able to know the percentage of suggestions that are surely the top.

6.2 Ranking of Approximate Recommendations

For ranking the approximate recommendations, one approach is to rank the suggestions that correspond to different word orders or to similar strings as if they were "normal" suggestions. An alternative approach is to penalize the "approximate" suggestions, i.e. to promote those suggestions that correspond exactly to the user's input string, and penalize the rest taking into account also their difference to the user's input string. This can be realized also by a simple mechanism. For instance, suppose that the system policy is to prompt the top-10 suggestions. A simple policy is to show the approximate matches always at the last positions of the 10 suggestions, i.e. at the positions 6 to 10. With regard to the ranking of the approximate recommendations, one approach is to rely solely on their original score. Alternatively we can reduce their score based on their distance with the user input string, e.g. taking into account the number of common words, the edit distance, or combinations of these.[e]

7 Experimental Evaluation

In this section we first summarize the main experimental results (from [2] and [5]) regarding the proposed index structures (for reasons of self-containedness), we describe criteria that should be used and in the right order for selecting the most appropriate index, and we report experimental results regarding the update of the index (Section 7.1). Subsequently we experimentally evaluate the caching schemes described in Section 4.1 for understanding how they affect the throughput that can be served and the response time (Section 7.2). We also report indicative results regarding word-order independent and typo-tolerant search, and we quantify the benefits of this flexibility in terms of how many more suggestions users receive (Section 7.3). Finally, we experimentally evaluate the proposed top-K algorithm described in Section 6.1 for efficiently retrieving the top-K suggestions (Section 7.4).

In all experiments we used four query logs of different sizes. One with 100,000 queries, one with 500,000 queries, one with 1 million queries, and one with 1 million *distinct* queries. The first three logs are subsets of a random *real* query log sample from Altavista web search engine (of year 2003). The last one is synthetic but it retains the main features of a real query log. The logs that were used in the experiments are available to download.[f] Table 3 reports their main features. 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 the system was in Java 1.6 (J2EE platform).

[e] A comparative evaluation of the above ranking choices as well as their personalization are beyond the scope of this work but certainly important topics for future research.
[f] `http://www.ics.forth.gr/~fafalios/iosDatasets.zip`

Table 3: The query logs used in the experiments.

Num of queries in the log	Num of unique queries	Avg num of words per query	Num of distinct words	Avg num of chars per query
100,000 (Altavista 2003)	72,075	3.1	64,904	21
500,000 (Altavista 2003)	293,156	3.1	153,593	21
1m (Altavista 2003)	513,275	3.1	219,178	21
1m (synthetic)	1m	3.1	369,661	21

7.1 Evaluation of Index Structures

[2] presents the results of a comparative experimental evaluation of the four indexing approaches (SET, STIE, PET, PTIE) that were described in Section 3. Regarding the *size of the index*, SET requires the most 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. Finally, PET requires less space than STIE only for very large query logs and according to the way that the index has been partitioned. As regards the *average response time*, SET approach is much more efficient than all the other approaches. However, as mentioned above, for large query logs 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 (the reason is that PET has to load at request time a subtrie of usually big size).

[5] performed a *large scale experimental evaluation* of PTIE (which is the more scalable approach) using a very large query log and large amounts of precomputed information. In particular, it was used a synthetic query log of *one million distinct* queries and synthetic precomputed information of *one terabyte*. It measured the average time for retrieving the suggestions, the results and the clusters for a random input string without using any cache scheme. The average retrieval time was about *135 ms*, proving that PTIE is efficient even for very large query logs and precomputed information. At last we should mention that with a *modest* personal computer the other indexing approaches cannot be used in such a big number of queries and precomputed information. PTIE is very scalable because the data that it has to load at request time has small size, its total cost of main memory is low and depends on the number of requests, and also it has low response time.

Selecting the Right Index. The main conclusion is that the proposed functionality is feasible for real time interaction even if both the log file and the preprocessed information have very large 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 and can be used even with very large query logs. 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, as we saw in Section 4.

Note that it is always useful to have an efficient method that best utilizes the existing resources and does not require more hardware. Regarding memory, there is always a hierarchy of memories,[g] meaning that we can assume that we have two kinds of memory: one smaller faster and more expensive, and one bigger but less fast. Approaches that exploit the faster and smaller memory are always useful. Nevertheless, the proposed functionality can be exploited also in a *distributed* system. Specifically, in SET, we can load the enriched trie at each node and apply a load balancing technique [48, 49, 50], e.g. round robin dispatching of requests. In STIE, we can distribute not only the trie, but also the random access files at several nodes. In PET and PTIE, we can load each subtrie in one or more nodes according to the frequency of use of each subtrie. In that case, the mediator forwards the request to the appropriate node and applies the round robin dispatch in case there are more than one nodes with the same subtrie.

Index Update Time. Here we evaluate the proposed index update procedure described in Section 3.4. For the experiments, we used two real query logs, one with 500,000 queries and one with 100,000 queries (Table 3 reports their main features). The big log was used for building the index and the small one for updating it. We measured the *index update time* of STIE and PTIE since they are considered the most scalable indexing approaches. Specifically, we measured the time for a) analyzing the new query log (and computing the scores of the queries according to the formula described in Section 6), b) loading the existing trie (for STIE) or the required subtries (for PTIE), c) updating the data of the queries that already lie in the index, d) adding the new queries with all their data, e) updating the scores of the ancestor queries (which is required by the scoring formula), and f) saving the new (updated) trie (or subtries in case of PTIE). For c) and d) we stored the data in new random access files. Note that we do not measured the time for deriving the top search results and the overview information because that time highly depends on the particular algorithms that are employed and the underlying sources (moreover, the system may have stored that information while evaluating it).

Table 4 reports the results. We notice that analyzing the query log and computing the scores of the queries is not a time consuming task (about 1.5 second). For updating STIE, saving the update trie is the most time consuming task (about 43 seconds). This happens because the size of STIE is very large (specifically, in our setting the updated trie was about 600 mb). For the same reason, loading the existing (large) trie is time consuming (about 28 seconds). Moreover, adding the new queries in the index consumes about 26% of the overall time, while updating the existing queries and the scores of the ancestors is not time consuming.

For updating PTIE, loading the subtries is the most time consuming task. This is a predictable result because for each distinct query in the new query log we have to load its

[g] `http://en.wikipedia.org/wiki/Memory_hierarchy`

Table 4: The main index update tasks and the time for updating an index of 500,000 queries using a query log of 100,000 queries.

Task	Time in STIE	Time in PTIE
Analyze the query log and compute the scores	1,500 ms (1.4%)	1,500 ms (1.4%)
Load the existing trie (STIE) or the required subtries (PTIE)	27,922 ms (25.3%)	58,913 ms (55.5%)
Update the data of the queries that already lie in the index	8,054 ms (7.3%)	4,583 ms (4.3%)
Add the new queries with all their data	28,582 ms (25.9%)	8,351 ms (7.9%)
Update the scores of the ancestor queries	876 ms (0.8%)	154 ms (0.15%)
Save the new (updated) trie (or subtries in case of PTIE)	43,267 ms (39.3%)	32,573 ms (30.7%)
Total time	110,201 ms	106,074 ms

corresponding subtrie (if it is not already in the main memory). Note also that we did not use any caching scheme. Furthermore, saving the updated subtries consumes about 30% of the overall time. We notice also that in PTIE, adding the new queries or updating the existing ones consumes much less time that in STIE. This happens because in PTIE the subtrie that we update for each distinct query has much smaller size than the single big trie of STIE.

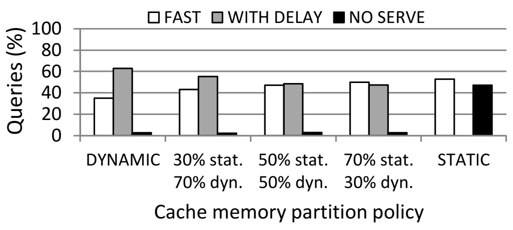
7.2 Evaluation of Caching Schemes

To evaluate the benefits of caching in our setting, we performed a comparative evaluation of the following choices:

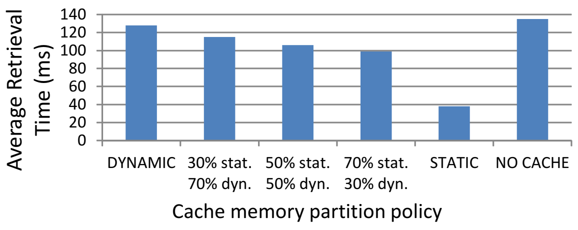
(1) Static cache
(2) Dynamic cache
(3) Hybrid (static: 30%, dynamic: 70%)
(4) Hybrid (static: 50%, dynamic: 50%)
(5) Hybrid (static: 70%, dynamic: 30%)
(6) No cache

We used a synthetic query log of *one million distinct* queries (see Table 3 for its features). We created 344 subtries using the PTIE approach with trie partitioning based on the first two characters and with minimum 1,000 entries per subtrie. In a loop without any "sleep" period, we run 10,000 random queries selected from the log file. The query rate was about 8 queries per second. We chose to set the memory capacity (MC) to 60 subtries, i.e. 17.44% of all subtries can fit in main memory at the same time. The time threshold (TT) that a subtrie is considered in use (and thus it cannot be removed) was set to 10 seconds, i.e. about $10 * 8 = 80$ queries have to be served at the same time. As we will see later, the values of these three factors (*query rate*, *percentage of cached subtries* and *time threshold*) highly affect the experimental results. As far as the static cache is concerned, we load the more frequent subtries after a query log analysis.

Served Queries. Figure 7a reports a) the percentage of queries that were served from the cache without delay (fast response since the request can be served instantly from the cache), b) the percentage of queries that were served from the cache with delay (in order to remove the less frequent subtrie and load the requested one), and c) the percentage of queries that could not be served (because of memory overloading). We see how the results change starting from a full dynamic cache and resulting in a full static cache. We observe that as the cache becomes more static, more queries are served fast and less are served with delay. However,



(a) Served Queries.



(b) Average response time.

Fig. 7

in a full static cache, almost half of the queries cannot be served. On the contrary, this percentage is very low in the other policies (lower than 5%).

Average Response Time. Figure 7b illustrates the average response times for all approaches. We notice that as the cache becomes more static, the average response time gets faster, since more queries can be served instantly from the static cache. On the other hand, in a full static cache there is no space in order to serve the less frequent queries and as a result the system gets overloaded and many requests cannot be served. In case we do not use any cache policy, i.e. the system loads the requested subtrie in each user's request (keystroke) and removes it when the user has been served, the average response time is slightly higher than that of a dynamic cache. However, if many users request a subtrie at the same time, the system can easily get overloaded. Specifically, in our experiments the main memory capacity is set to 60 subtries, i.e. the system can serve at most 60 queries at the same time. Thus, since we have 80 queries that have to be served at the same time period, 20 of them will not be served (25%).

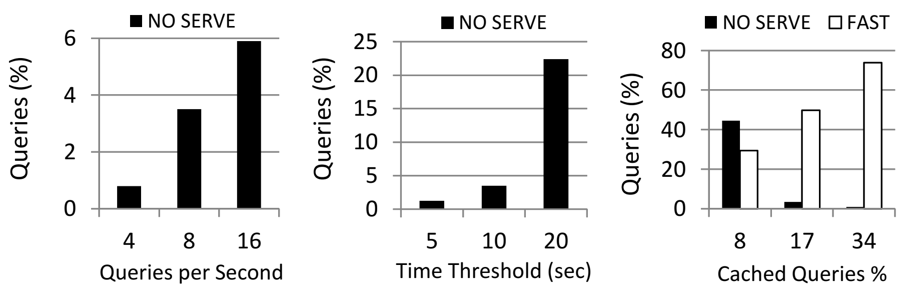
Discussion. We should clarify that the above results concern the case where each user (i.e. each iteration in the loop) requests only one subtrie, in particular the subtrie that contains the randomly selected query. The results do not take into account the common case in which the user continues typing a query and he is served instantly by the same subtrie. For this reason in realistic workloads the results are expected to be better.

As we mentioned before, beyond the aforementioned experiments, results and conclusions, we can say that there are three main factors that affect the experimental results:

(1) Number of queries per second (*Query Rate*).
(2) Time Threshold (*TT*), the time (seconds) that a subtrie is considered in use and thus it cannot be removed from the cache.
(3) Percentage of cached subtries (*MC / Total number of subtries*).

To understand how these factors impact the behavior of the system, we conducted additional experiments assuming a hybrid caching policy with 70% static and 30% dynamic cache. Figure 8 illustrates how each of the above factors affects the results keeping constant the other two factors. We observe that the smaller the query rate is, the less requests cannot be served (Figure 8a). As far as the time threshold is concerned, we can see that less threshold time results in less requests that cannot be served (Figure 8b). Finally, for bigger percentage of cached queries we expect better results. i.e. more fast responses and less requests that cannot be served (Figure 8c).

Synopsis. Concluding the above results, we could stress that: (i) the best caching policy is the hybrid with 70% static and 30% dynamic cache, as it combines low percentage of requests



(a) (b) (c)

Fig. 8: (a) How the number of queries per second affects the results, with 10 seconds time threshold and 17% cached queries. (b) How the time threshold affects the results, with 8 queries per second and 17% cached results. (c) How the percentage of cached queries affects the results, with 10 seconds time threshold and 8 queries per second.

that cannot be served, high percentage of requests that are served fast and low average response time. In particular, regarding the settings of the aforementioned experiment, we found out that the particular caching scheme offered about 80% better throughput (since less than 5% of the queries could not be served, contrary to the 25% of no-cache case), and about 25% speedup for queries that lie in the index. (ii) The more percentage of cached subtries we have (that means more main memory capacity), the better results we expect. (iii) We cannot know in advance the number of queries per second, however we expect that for small numbers, the requests that cannot be served are few. (iv) The time threshold that a subtrie is considered in use is constant and thus we know in advance how this number affects the behavior of our system.

To sum up we saw that a caching scheme can significantly increase the throughput and alleviate overloading problems. By partitioning the cache into a static and a dynamic part, and giving more space to the static part, we can increase the throughput that can be served and speed up `IOS` by offering lower response times.

7.3 Evaluation of Word-Order Independent and Typo-Tolerant Search

7.3.1 Retrieval Time

We decided to measure the average retrieval type of STIE and PTIE since they are the most scalable. Specifically, using a real query log of one million queries (see Table 3 for its main features), we run 1,000 random queries selected from the log of various number of words (for evaluating the word-order independent search) and various number of characters (for evaluating the typo-tolerant search). Furthermore, we consider the worst case in which we cannot retrieve incremental suggestions, we do not use any caching scheme (for PTIE) and we do not exploit the top-K algorithm for retrieving the suggestions.

Word-order independent suggestions. Regarding word-order independent suggestions, we run the experiment for 2-word, 4-word, 8-word and 12-word queries and we measure the time for retrieving suggestions that start from a word and contain at least one of the remaining words (approach `WORD_CONTAIN`). For each query we keep only the first two characters from

Table 5: Average retrieval time.

(a) Word-order independent.

Query length	STIE	PTIE
2-word	29ms	182ms
4-word	37ms	492ms
8-word	48ms	829ms
12-word	58ms	1,054ms

(b) Typo-tolerant in STIE.

Query length	`DETECT ALL`	`IGNORE FIRST`
4-char	96ms	28ms
8-char	142ms	39ms
12-char	225ms	36ms
16-char	305ms	32ms

the last word. Table 5a reports the results. We notice that STIE can efficiently support this functionality with average retrieval time lower than 60 ms, even for queries with many words. PTIE is efficient for 2-word queries (which is the common case) with average retrieval time about 180 ms, but for queries with many words the time is very high. Consequently, for queries with more than two words and if the system cannot offer incremental suggestions, PTIE must search only the subtries that lie in the cache.

Typo-tolerant suggestions. With regard to typo-tolerant suggestions, we run the experiment keeping from each query the first-4, first-8, first-12 and first-16 characters. Table 5b reports the results of STIE for the common case (approach `DETECT_ALL`), and when ignoring typo in the first character (approach `IGNORE_FIRST`). We notice that STIE can efficiently support the `DETECT_ALL` approach if the query does not have too many characters (about less than 15 chars), and very efficiently when ignoring typo in the first character (with average retrieval time lower than 40 ms). Since we do not use any caching scheme, we do not report results of PTIE because PTIE has to load a lot of subtries which is very time consuming. For this reason, PTIE must offer this functionality only by ignoring typo in the first character. Furthermore, if the trie has not been partitioned based on the *first character* ($k > 1$), we must check only the subtries that lie in the cache.

Tolerating both typos and different word orders. In this case we run the experiment for 2-word, 4-word, 8-word and 12-word queries and we measure the time for a) detecting the active nodes of each word *ignoring typo in the first character*, b) retrieving the suggestions of each active node, and c) filtering the suggestions exploiting the knowledge of the remaining (active) words. The results revealed that STIE can efficiently support the combination of both functionalities for queries with no more than 4 words (which is the common case). Furthermore, PTIE must offer this functionality incrementally or by checking only the subtries that lie in the cache.

7.3.2 Number of Additional Suggestions

In order to quantify the effect of the offered flexibility, we measure the average number of additional suggestions we get when we tolerate different word orders and spelling errors. Specifically, using a *real* query log of one million queries (see Table 3 for its main features), we performed the experiments described below.

Word-order independent suggestions. For 1,000 random (distinct) queries in the log that have two words and each word contains at least four characters, we keep only the first two characters of the last word and find its suggestions (initially with the same word order and then without considering the order). For example, for the query `marilyn monroe` we find all queries that start with `marilyn mo` and then the suggestions of the same query without

Table 6: Average number of additional suggestions per query when tolerating different word orders.

Query length	Having typed the first 2 chars of the last word	Having typed the first 3 chars of the last word	Having typed the first 4 chars of the last word
2-word	23.6	3.5	1.9
3-word	69.3	28.1	22.2
4-word	117.5	58.4	47.8

Table 7: Average number of additional suggestions per query when tolerating spelling errors.

Approach	Having typed 4 chars (edt_dist=1)	Having typed 8 chars (edt_dist=2)	Having typed 12 chars (edt_dist=4)	Having typed 16 chars (edt_dist=5)
`DETECT_ALL`	975	55.8	72	34
`IGNORE_FIRST`	792	52	50	27

considering the word order (approach `WORD_CONTAIN`). We do the same for all queries in the log and we compute the average number of the *additional* suggestions we get (*new* − *old*) when we tolerate different word orders. We also run the same experiment keeping the first three and the first four characters of the last word, and using 3-word and 4-word queries.

Table 6 shows the results. Notice the number of suggestions always increases and for queries with more than two words, even if we have typed many characters of the last word, the additional suggestions are significantly more.

Typo-tolerant suggestions. For 1,000 random (distinct) queries in the log that have at least four characters, we keep only the first four characters and compute its suggestions. For example, for the query `marilyn monroe` we find all queries that start with `mari`. Afterwards, we find the active nodes of that query (approach `DETECT_ALL` with edit distance threshold equal to one) and get their corresponding suggestions. Following the same procedure for all distinct queries in the log, we find the average number of the *additional* suggestions we get when we tolerate spelling errors. We run the same experiment keeping the first 8, first 12, and first 16 characters, and allowing one edit operation per three characters.

Table 7 (first row) reports the average number of additional suggestions in each case. Note that the results highly depend on the selected edit distance threshold and the distinct number of queries in the index. We notice that (in our setting) we get much more suggestions in all cases. We run the same experiments ignoring typo in the first character (approach `INGORE_FIRST`), e.g. for the query `marilyn` we found the similar queries that start with “m”. Table 7 (second row) reports the results. Notice that also here the suggestions are much more and the difference with the first approach is small. If we also consider that many of the suggestions may be false (we are not able to know user’s intent), this is a strong rationale for ignoring typo in the first character.

7.3.3 Conclusion

Undoubtedly, word-order independent and typo-tolerant search increase flexibility (and thus user’s satisfaction), and the degree of exploitation of the precomputed information. Furthermore, they increase the number of queries for which the system can assist the corresponding decision making process. However, both functionalities increase system’s response time since the system has to perform more trie traversals, several string comparisons and maybe (in case

Table 8: Efficient support of *word-order independent search*, *typo-tolerant search* and their *combination* over the proposed index structures.

Functionality	Approach	STIE	PTIE
Word-order indep.	`ALL_PERMUT`	✓	Only for cached subtries
	`WORD_CONTAIN`	✓	Only incrementally or for cached subtries
Typo-tolerant	`DETECT_ALL`	Only for input length<15 chars	X
	`IGNORE_FIRST`	✓	Only for $k = 1$ or cached subtries
Combination	`DETECT_ALL`	X	X
	`IGNORE_FIRST`	Only for $m \leq 4$ or incrementally	Only incrementally and for $k = 1$, or for cached subtries

Note: k is the number of the first characters that determine trie partitioning and m is the number of words of user's input.

of PET and PTIE) to load and access other subtries. Table 8 illustrates the applicability of word-order independent search, typo-tolerant search and their combination, over STIE and PTIE (since these are the most scalable indexes). STIE can efficiently support word order independent and typo-tolerant search since the whole trie is in main memory. Only in case of large input strings we must ignore typo in the first character in order to efficiently tolerate typos. On the other hand, PTIE can efficiently support word-order independent search either incrementally or for the subtries that lie in the cache, and typo-tolerant search by always ignoring typo in the first character. For offering the combination of both functionalities, we must always ignore typo in the first character. In addition, in all approaches we can highly improve the response time if we adopt the top-K algorithm or a maximum time policy (as described in Section 6.1). These techniques are beneficial especially for PTIE since it may need to load and access many subtries.

Finally, these functionalities are worth offering only when there are *few* or *no* suggestions of the current input string, meaning that system's response time is not affected if there are many suggestions, since there is no need to search for typos or other word orders.

7.4 Evaluation of the top-K algorithm

Here we evaluate the effectiveness of the top-K algorithm described in Section 6.1. Using a *real* query log of one million queries (see Table 3 for its main features) and adopting the PTIE indexing approach, we measure the average time for retrieving the top-10 suggestions for various first-2 chars (the time does not include the time for loading the subtrie and the time for presenting the results to the user). Specifically, we run 1,000 iterations asking for suggestions starting with "th", 1,000 iterations asking for suggestions starting with "co", and the same for "mo", "je" and "ct". Note that each sequence has different number of suggestions (i.e. queries in the log that start with the corresponding first-2 chars). Table 9 reports the number of queries starting with each sequence (second line), the average time for retrieving the top-10 suggestions *without* adopting the proposed top-K algorithm (third line), and the average time for retrieving the top-10 suggestions *adopting* the proposed top-K algorithm (fourth line).

We notice that the improvement is impressing; three orders of magnitude. Without adopting the top-K algorithm, the system has to retrieve all the suggestions starting with a particular character sequence, rank them and return the top-10. Instead, by adopting the proposed

Table 9: The effectiveness of the top-K algorithm.

First-2 chars	th	co	mo	je
Queries starting with the first-2 chars	27,186	15,587	11,201	5,992
Retrieving top-10 suggestions **without** adopting the top-K alg.	1,150 ms	700 ms	300 ms	50 ms
Retrieving top-10 suggestions **adopting** the top-K alg.	$<$ 5ms	$\approx$ 0 ms	$\approx$ 0 ms	$\approx$ 0 ms

top-K algorithm, we avoid traversing parts of the tree that contain low ranked suggestions. This can greatly reduce the response time, especially in case of word order independent and typo tolerant suggestions which require to perform many trie traversals and string comparisons, and maybe (in case of PET and PTIE) to load and access other subtries.

8 Concluding Remarks

In this paper we elaborated on methods for enabling the instant provision of informative query recommendations that provide the user with a better *overview* of the search space. The supplementary information can be the result of various (time-consuming) tasks, e.g. textual clustering or entity mining over the contents of the top search results. These methods are generic in the sense that they are independent of the precomputed information, i.e. independent of the ranking, clustering, entity mining, etc. method. It is also important to clarify that the performance of these methods 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 the frequent distinct queries).

These informative recommendations give the users an overview of the information space and allow them to quickly discover what is popular among other users, to decide fast which of the suggested query completions to use, and what hits of the returned answer to inspect (alleviating the users' bias in the first page of results or the possible biases of the ranking method of the search system). Essentially, the interactive mechanism that we propose can assist the user in all steps of the search process. Specifically, in a keystroke basis, the user is getting information about the available *options* (recommended query completions), aid for *choosing an option* (appearance of supplementary information), aid for *acting* (no need to type the entire query so user actions become less laborious), aid for *evaluating* the choices (the user can see the number of results, the clusters, the entities, etc.), and aid for changing his *situation* (mental state may accompanied by a refined or different information need).

We paid special attention to the perspectives of *scalability* and *flexibility*. Regarding *scalability* we considered an approach based on precomputed information and we comparatively evaluated four trie-based index structures. The experimental results showed that a partitioned trie-based index structure can efficiently support recommendations for millions of distinct queries even with modest hardware. Essentially, we showed how one can provide instant access to large amount of data, utilizing the existing resources and without requiring more hardware. Then we described criteria that should be used and in the right order for selecting the most appropriate index according to the available main memory and desired performance. 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 web search system, we described an *incremental* approach for updating the index. For improving the throughput that can be served we analyzed experimentally various policies for *caching subtries*. According to our results a *hybrid* (70% static and 30% dynamic) caching policy seems to be the more appropriate choice yielding a throughput increment of around 80% and a 25% speedup. For reducing users effort and increasing the exploitation of the precomputed information, we provided algorithms (appropriate for the proposed trie-based index structures) which can tolerate different *word orders* and *spelling errors*. The experimental results, apart from revealing the *limits* for supporting these functionalities, showed that they significantly increase the number of recommendations especially for queries that contain several words (the latter usually have none or very few suggested completions). Furthermore, we proposed a top-K algorithm for retrieving the top-K suggestions which exploits the ranking information for reducing the (sub)trie traversals. The experimental results proved that the proposed algorithm highly improves the average retrieval time. This optimization is especially useful for the case of word order and typo tolerant recommendations which require making several (sub)trie traversals.

A direction for further research is to study how space-efficient (trie-based) data structures ([51, 37]) can be applied in our problem. Another direction is to analyze how exactly users exploit the precomputed information that appears instantly. This requires very fast eye-tracking equipment for measuring how many times (and under what conditions) the user glances at the displayed precomputed information, and methods appropriate for analyzing the gathered information. That will also assist deciding *where* [52, 53] and *how* [54] it is worth displaying the recommended information.

Acknowledgements

The authors would like to thank MUMIA COST action (IC1002, 2010-2014) for supporting a Short-Term Scientific Mission (STSM) of the first author in Vienna (December 2011), giving them valuable feedback for improving and extending this work.

References

1. F. Silvestri, "Mining query logs: Turning search usage data into knowledge," *Foundations and Trends in Information Retrieval*, vol. 4, no. 12, pp. 1–174, 2010.
2. P. Fafalios and Y. Tzitzikas, "Exploiting available memory and disk for scalable instant overview search," in *Web Information System Engineering–WISE 2011*, pp. 101–115, Springer, 2011.
3. S. Kopidaki, P. Papadakos, and Y. Tzitzikas, "STC+ and NM-STC: Two novel online results clustering methods for web searching," in *WISE '09: Proceedings of the 10th International Conference on Web Information Systems Engineering*, October 2009.
4. P. Fafalios, I. Kitsos, Y. Marketakis, C. Baldassarre, M. Salampasis, and Y. Tzitzikas, "Web searching with entity mining at query time," in *Multidisciplinary Information Retrieval*, pp. 73–88, Springer, 2012.
5. P. Fafalios, I. Kitsos, and Y. Tzitzikas, "Scalable, flexible and generic instant overview search," in *Proceedings of the 21st International Conference on World Wide Web*, pp. 333–336, ACM, 2012.
6. M. Käki and A. Aula, "Findex: improving search result use through automatic filtering categories," *Interacting with Computers*, vol. 17, no. 2, pp. 187–206, 2005.
7. M. Käki, "Findex: search result categories help users when document ranking fails," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 131–140, ACM, 2005.
8. B. Kules, R. Capra, M. Banta, and T. Sierra, "What do exploratory searchers look at in a faceted search interface?," in *Proceedings of the 9th ACM/IEEE-CS joint conference on Digital libraries*, pp. 313–322, ACM, 2009.
9. M. Wilson *et al.*, "A longitudinal study of exploratory and keyword search," in *Proceedings of the 8th ACM/IEEE-CS joint conference on Digital libraries*, pp. 52–56, ACM, 2008.
10. S. Teddy, F. Yap, C. Quek, and E. Lai, "A neurocognitive approach to decision making for the reconstruction of the metabolic insulin profile of a healthy person," *Handbook on Decision Making*,

pp. 497–532, 2010.

11. A. Spink, B. Jansen, D. Wolfram, and T. Saracevic, "From e-sex to e-commerce: Web search changes," *Computer*, vol. 35, no. 3, pp. 107–109, 2002.
12. L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: bringing order to the web.," 1999.
13. G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Information processing & management*, vol. 24, no. 5, pp. 513–523, 1988.
14. E. Segev, *Google and the Digital Divide: The Biases of Online Knowledge.* Chandos Publishing (Oxford), 2009.
15. L. Vaughan and M. Thelwall, "Search engine coverage bias: evidence and possible causes," *Information Processing & Management*, vol. 40, no. 4, pp. 693–707, 2004.
16. D. Bridge and F. Ricci, "Supporting product selection with query editing recommendations," in *Proceedings of the 2007 ACM conference on Recommender systems*, pp. 65–72, ACM, 2007.
17. S. Ji, G. Li, C. Li, and J. Feng, "Efficient interactive fuzzy keyword search," in *Proceedings of the 18th international conference on World wide web*, WWW '09, (New York, USA), pp. 371–380, ACM, 2009.
18. G. Li, S. Ji, C. Li, and J. Feng, "Efficient type-ahead search on relational data: a tastier approach," in *Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD '09, (New York, NY, USA), pp. 695–706, ACM, 2009.
19. H. Wu, G. Li, C. Li, and L. Zhou, "Seaform: Search-as-you-type in forms," in *Proceedings of the 36th International Conference on Very Large Data Bases*, VLDB '10, 2010.
20. H. Bast and I. Weber, "Type less, find more: fast autocompletion search with a succinct index," in *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '06, (New York, NY, USA), pp. 364–371, ACM, 2006.
21. G. Li, J. Feng, and L. Zhou, "Interactive search in xml data," in *Proceedings of the 18th international conference on World wide web*, WWW '09, (New York, NY, USA), pp. 1063–1064, ACM, 2009.
22. G. Li, J. Wang, C. Li, and J. Feng, "Supporting efficient top-k queries in type-ahead search," in *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval*, pp. 355–364, ACM, 2012.
23. H. Bast and M. Celikik, "Efficient fuzzy search in large text collections," *ACM Transactions on Information Systems (TOIS)*, vol. 31, no. 2, p. 10, 2013.
24. H. Bast and I. Weber, "The completesearch engine: Interactive, efficient, and towards ir& db integration.," in *CIDR*, vol. 7, pp. 88–95, 2007.
25. E. Markatos, "On caching search engine query results," *Computer Communications*, vol. 24, no. 2, pp. 137–143, 2001.
26. P. Saraiva, E. Silva de Moura, N. Ziviani, W. Meira, R. Fonseca, and B. Riberio-Neto, "Rank-preserving two-level caching for scalable search engines," in *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 51–58, ACM, 2001.
27. X. Long and T. Suel, "Three-level caching for efficient query processing in large web search engines," *World Wide Web*, vol. 9, no. 4, pp. 369–395, 2006.
28. 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," *ACM Transactions on Information Systems (TOIS)*, vol. 24, no. 1, pp. 51–78, 2006.
29. R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri, "The impact of caching on search engines," in *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 183–190, ACM, 2007.
30. R. Lempel and S. Moran, "Predictive caching and prefetching of query results in search engines," in *Proceedings of the 12th international conference on World Wide Web*, pp. 19–28, ACM, 2003.
31. G. Skobeltsyn, F. Junqueira, V. Plachouras, and R. Baeza-Yates, "Resin: a combination of results caching and index pruning for high-performance web search engines," in *Proceedings of the*

31st annual international ACM SIGIR conference on Research and development in information retrieval, pp. 131–138, ACM, 2008.
32. Q. Gan and T. Suel, "Improved techniques for result caching in web search engines," in *Proceedings of the 18th international conference on World wide web*, pp. 431–440, ACM, 2009.
33. B. Cambazoglu, F. Junqueira, V. Plachouras, S. Banachowski, B. Cui, S. Lim, and B. Bridge, "A refreshing perspective of search engine caching," in *Proceedings of the 19th international conference on World wide web*, pp. 181–190, ACM, 2010.
34. R. Blanco, E. Bortnikov, F. Junqueira, R. Lempel, L. Telloli, and H. Zaragoza, "Caching search engine results over incremental indices," in *Proceeding of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '10, (New York, NY, USA), pp. 82–89, ACM, 2010.
35. S. Chaudhuri and R. Kaushik, "Extending autocompletion to tolerate errors," *SIGMOD*, 2009.
36. H. Duan and B. Hsu, "Online spelling correction for query completion," *WWW*, 2011.
37. B.-J. P. Hsu and G. Ottaviano, "Space-efficient data structures for top-k completion," in *Proceedings of the 22nd international conference on World Wide Web*, pp. 583–594, International World Wide Web Conferences Steering Committee, 2013.
38. D. Kastrinakis and Y. Tzitzikas, "Advancing search query autocompletion services with more and better suggestions," in *Proceedings of the 10th international conference on Web engineering*, pp. 35–49, Springer, 2010.
39. C. Silverstein, H. Marais, M. Henzinger, and M. Moricz, "Analysis of a very large web search engine query log," *SIGIR Forum*, vol. 33, pp. 6–12, September 1999.
40. S. Ding, J. Attenberg, R. Baeza-Yates, and T. Suel, "Batch query processing for web search engines," in *Proceedings of the fourth ACM international conference on Web search and data mining*, pp. 137–146, ACM, 2011.
41. Y. Xie and D. O'Hallaron, "Locality in search engine queries and its implications for caching," in *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 3, 2002.
42. J. Teevan, E. Adar, R. Jones, and M. Potts, "Information re-retrieval: repeat queries in yahoo's logs," in *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 151–158, ACM, 2007.
43. R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, 1977.
44. S. Cucerzan and E. Brill, "Spelling correction as an iterative process that exploits the collective knowledge of web users," in *Proceedings of EMNLP*, vol. 4, pp. 293–300, 2004.
45. A. Broder, P. Ciccolo, E. Gabrilovich, V. Josifovski, D. Metzler, L. Riedel, and J. Yuan, "Online expansion of rare queries for sponsored search," in *Proceedings of the 18th international conference on World wide web*, pp. 511–520, ACM, 2009.
46. G. Navarro, "A Guided Tour to Approximate String Matching," *ACM computing surveys (CSUR)*, vol. 33, no. 1, pp. 31–88, 2001.
47. R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," in *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 102–113, ACM, 2001.
48. V. Cardellini, M. Colajanni, and P. Yu, "Dynamic load balancing on web-server systems," *Internet Computing, IEEE*, vol. 3, no. 3, pp. 28 –39, 1999.
49. V. Ungureanu, B. Melamed, and M. Katehakis, "Effective load balancing for cluster-based servers employing job preemption," *Performance Evaluation*, vol. 65, no. 8, pp. 606–622, 2008.
50. T. Chieu, A. Mohindra, A. Karve, and A. Segal, "Dynamic scaling of web applications in a virtualized cloud computing environment," in *e-Business Engineering, 2009. ICEBE '09. IEEE International Conference on*, pp. 281–286, oct. 2009.
51. R. Grossi and G. Ottaviano, "Fast compressed tries through path decompositions," *CoRR*, vol. abs/1111.5220, 2011.
52. R. Moore, E. Churchill, and R. Kantamneni, "Three sequential positions of query repair in inter-

actions with internet search engines," in *Proceedings of the ACM 2011 conference on Computer supported cooperative work*, pp. 415–424, ACM, 2011.
53. G. Buscher, A. Dengel, R. Biedert, and L. V. Elst, "Attentive documents: Eye tracking as implicit feedback for information retrieval and beyond," *ACM Trans. Interact. Intell. Syst.*, vol. 1, no. 2, 2012.
54. L. Chen and H. Tsoi, "Users' decision behavior in recommender interfaces: Impact of layout design," *RecSys'11 Workshop on Human Decision Making in Recommender Systems*, 2011.
55. R. Baeza-Yates, B. Ribeiro-Neto, *et al.*, *Modern information retrieval*, vol. 463. ACM press New York, 1999.

Appendix A: Benefits for the Server's Side

Less incoming queries, reduced computational cost per received query. Apart from the benefit for the users side, our approach is beneficial also for the server. In particular, we could point that an `IOS` functionality (a) *reduces the number of incoming queries* which are not really useful for the end users, since it assists them in avoiding wrongly typed queries (user adapts his query on the fly until the results match what he wants), and (b) reduces the computational cost because the *same precomputed information is exploited in several requests* and thus the engine has to evaluate less tasks at run time. The combination of the above increases the throughput of the server since the number of incoming queries is much smaller and the response time is much less. This is true even if the user types the right query (for his information need) from the beginning. For instance, and for the case of clustering, in `IOS` for serving one request we have to load (if not already loaded in memory) and show (send via HTTP) the cluster label tree. Without `IOS` (e.g. in plain `StellaSearch` [3]), for serving one request we have to evaluate the query (either locally or send it to a remote engine), fetch the top-L snippets, apply the clustering algorithm `NM-STC` to derive the cluster label tree, and finally send the outcome over HTTP to the end user. The case is analogous for the other kinds of precomputed information (e.g. entity mining over the top hits, etc.). In general we can say that the more expensive the "overview information" is to derive, the more beneficial for the server side our approach and indexes are.

In the context of real time results clustering, we could also say that the indexes of `IOS` can be considered as a form of cache for speeding up `StellaSearch`. The trie over the query strings is actually the data structure allowing fast lookups in that cache. Under this perspective, we can say that in our work we exploit that cache not only after the user query has been submitted (i.e. after the user has pressed the *submit* button), but while the user types his query letter by letter, i.e. during typing. Also note that the size of that cache is large therefore one part of it is in main memory, the rest in secondary.

Less monetary cost (at meta search level). In a meta-search setting, the engine has to connect to the underlying search engines in order to get the results of a query. Such services are not for free and sometimes they are billed according to the number of served queries. An `IOS` approach reduces the queries sent to the underlying engines and thus can save money.

Less network connections (at meta search level). If the construction of the "overview of the results" requires connecting to several external sources (e.g. web pages, LOD SPARQL endpoints, etc), then an `IOS` approach reduces the number of connections which are required.

Appendix B: Expected Speedup

Since past works (i.e. those mentioned in the introduction) have proved that the supplementary (overview) information is useful for the users during searching, what is worth evaluating here is the *instant* behavior, i.e. the speedup that a search system offers to its users by adopting the proposed functionality.

The *speedup* that we expect is proportional to the percentage of the incoming queries that can be served from our index. For example, if for every 1,000 incoming queries, about 200 are served from `IOS` (i.e. they are not submitted to the server), then the $S = 20\%$ of incoming queries are served instantly (e.g. in time less than 200ms) and the 80% are served with delay (the exact delay depends on the underlying search system, the overview information, the particular algorithms that are employed, etc.). However, the more queries we precompute the bigger the speedup is. The number of the precomputed queries depends on the system's available main and secondary memory. In particular, the only real price to pay is actually the space required for storing the precomputed information.

In a typical standalone web search engine, if $\mathcal{V}$ is the vocabulary of the dataset, then the size of the inverted file is $O(|\mathcal{V}|^2)$, i.e. the square of dataset's distinct words (Heap's Law [55]). Considering that about 60% of queries submitted to search engines has up to 2 words [11], then the maximum number of two-word queries is $|\mathcal{V}|^2$, i.e. in the magnitude of the size of the inverted file. Thus, with a disk space proportional to $O(|\mathcal{V}|^2)$, we can have precomputed information for 60% of queries. Moreover, note that contrary to the inverted file whose size grows according to the size of the collection, our index is affected only by the size of the query log and in particular by the number of distinct queries.

The required main memory depends on the caching mechanism and the index that we use. For instance, in PTIE, a subtrie of 1,000 queries has size about 1 MB, i.e. a machine with 1 GB available main memory can keep in memory at the same time about 1,000 subtries of 1,000,000 total distinct queries. Thus, with 100 MB (10%) more main memory, we can have instant search for 100,000 more incoming queries (10%).