Scalable, Flexible and Generic Instant Overview Search

Pavlos Fafalios, Ioannis Kitsos and Yannis Tzitzikas
Institute of Computer Science, FORTH-ICS, GREECE, and
Computer Science Department, University of Crete, GREECE
{fafalios,kitsos,tzitzik}@ics.forth.gr

ABSTRACT

The last years there is an increasing interest on providing the top search results while the user types a query letter by letter. In this paper we present and demonstrate a *family* of instant search applications which apart from showing instantly only the top search results, they can show various other kinds of *precomputed aggregated information.* This paradigm is more helpful for the end user (in comparison to the classic search-as-you-type), since it can combine *autocompletion*, *search-as-you-type*, *results clustering*, *faceted search*, *entity mining*, etc. Furthermore, apart from being helpful for the end user, it is also beneficial for the server's side. 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. We demonstrate how this can be achieved using very modest hardware. Our approach relies on (a) a *partitioned trie-based index* that exploits the available main memory and disk, and (b) dedicated *caching techniques.* We report performance results over a server running on a modest personal computer (with 3 GB main memory) that provides instant services for *millions* of distinct queries and *terabytes* of precomputed information. Furthermore these services are tolerant to user *typos* and different *word orders.*

1. INTRODUCTION

Query autocompletion services help users in formulating queries by exploiting past (and logged) queries. *Instant search* (or search-as-you-type) is an enhancement which also shows the first page of results of the top query suggestion as the user types. We observe an increasing interest on providing such functionality evidenced by the emergence of several systems: e.g. Google[1] for plain web searching, Facebook[2] for showing the more relative friends (pages, etc), IMDB[3] for showing the more relevant movies (actors, etc) together with a photo, and so on.

In general, such immediate feedback allows the user to stop typing if the desired results have already been recommended. If on the other hand the sought results are not there, the user can continue typing or change what he has typed so far. In general, we can say that the user adapts his query on the fly until the results match what he wants.

Apart from showing instantly only the top hits of the guessed query, [6] proposed showing several other kinds of supplementary information that provide the user with a better overview of the search space (the term `IOS`, standing for *instant overview search*, was introduced). Essentially that work 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. Here we extend these ideas and techniques, specifically: a) we introduce and evaluate a *caching scheme* that apart from *speeding up* these services it *increases* the *throughput* of the service provider, b) we demonstrate the efficiency of the approach in supporting much bigger number of queries and precomputed information, c) we extend these services so that to be tolerant to *user typos* and independent of the typed *word order*, and d) we demonstrate various novel applications of instant overview search (notably one that enhances web searching with *entity mining*).

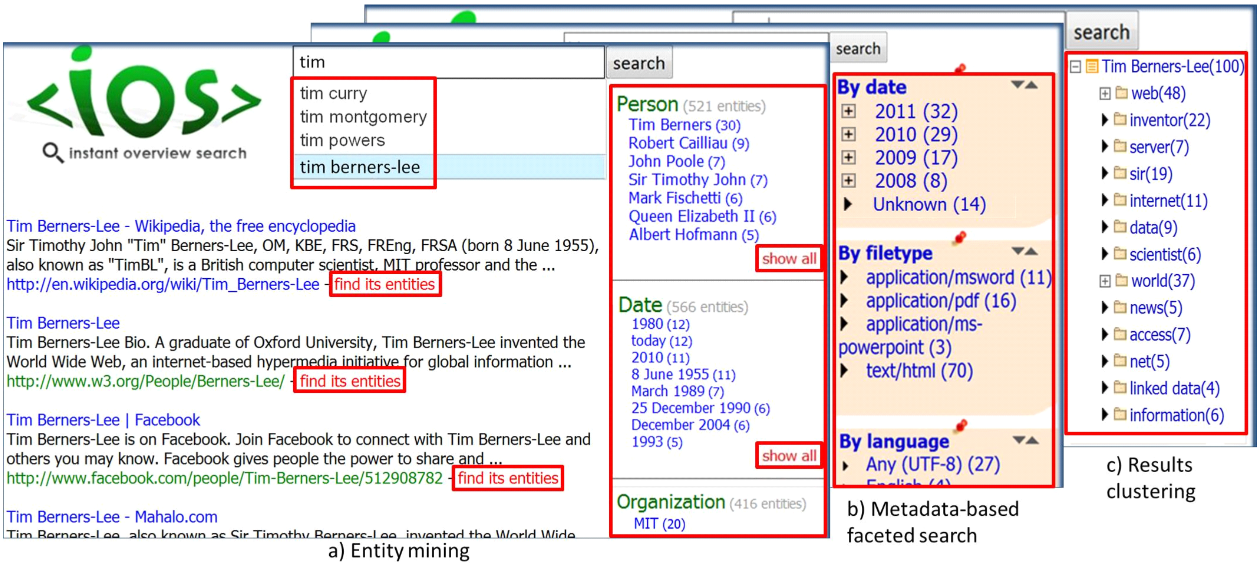
The applications that will be demonstrated (most are already web accessible[4]) include:
1) a meta-search engine (MSE) over Google offering *instant clustering*[5] *of results* (see Fig.1c). The current deployment contains precomputed information for 20,000 queries, at demo we will use a deployment with over one *million* queries.
2) a standalone web search engine (WSE) offering *instant metadata-based groupings [10] of the results* (see Fig.1b), and
3) a MSE offering *instant entity mining* over the top hits (see Fig.1a). This system retrieves the top-50 hits from Google, mines the content of *each* result (using the entity mining tool *Gate*[6]) and presents to the user a categorized list with the discovered entities. When the user clicks on an entity, the results of the specific entity are loaded instantly. Moreover, the system mines the query string and ranks higher the categories and the entities that are found in the query string. Note that, without `IOS` functionality, this computation costs a lot both in time (about one minute) and in main memory (500 MB in average) per query.

In all these applications, the user with a few keystrokes gets quite informative overviews of the search space.

2. HOW IT WORKS

Figure 2 sketches the architecture, while the key issues are described below.

[1] http://www.google.com/instant/
[2] http://www.facebook.com/
[3] http://www.imdb.com/
[4] http://www.ics.forth.gr/isl/ios
[5] We adopt the clustering algorithm NM-STC [9]
[6] http://gate.ac.uk/ie/



Tim Berners-Lee - Wikipedia, the free encyclopedia
Sir Timothy John "Tim" Berners-Lee, OM, KBE, FRS, FREng, FRSA (born 8 June 1955), also known as "TimBL", is a British computer scientist, MIT professor and the ...
http://en.wikipedia.org/wiki/Tim_Berners-Lee - find its entities

Tim Berners-Lee
Tim Berners-Lee Bio. A graduate of Oxford University, Tim Berners-Lee invented the World Wide Web, an internet-based hypermedia initiative for global information ...
http://www.w3.org/People/Berners-Lee/ - find its entities

Tim Berners-Lee | Facebook
Tim Berners-Lee is on Facebook. Join Facebook to connect with Tim Berners-Lee and others you may know. Facebook gives people the power to share and ...
http://www.facebook.com/people/Tim-Berners-Lee/512908782 - find its entities

Tim Berners-Lee - Mahalo.com

a) Entity mining

b) Metadata-based faceted search

c) Results clustering

Figure 1: IOS applications.

Partitioned Index

We index all distinct queries of the query log (or only the more frequent ones) using a trie. For being able to accommodate very large number of queries we adopt the trie partitioning method proposed in [8]. According to that method, the trie is partitioned to a number of subtries. For each query in the trie (or subtries) we apply the preprocessing steps (e.g. results clustering, metadata-based groupings, entity mining, etc), and we save the results in one or more separate *random access files*. Consequently, for each query entry the subtrie keeps only three numbers: one for the file, one for the bytes to skip and one for the bytes to read. Actually we adopt the "PTIE" (Partitioned Trie with Indexes to External files), which is the more scalable index from those proposed and described at [6] (this index is sketched at Fig.3). In our setting this means that for serving one request, at most one "subtrie" has to be loaded in main memory, and one file to be accessed. Another important point is that the indexes are kept fresh using an *incremental* algorithm (note that the trie-based indexes are very convenient for this task).

Caching

We have developed a *hybrid* caching mechanism, i.e. we keep always in memory the subtries of the most frequent queries based on past log analysis (static part), and anticipate an amount of memory for loading subtries which are not cached (dynamic part). We give 70% at the static and 30% at the dynamic part of the cache, since (as we will see in Section 3) this is the best policy for speeding up the system and increasing its throughput. If we have a request for a subtrie that is in the static or the dynamic part of the cache, we serve it instantly, otherwise we first load the requested subtrie at the dynamic part. In case we have a request that cannot be served neither from the static nor from the dynamic part, and the dynamic part is full, we remove the less frequent subtrie that is not in use from the dynamic part and then load the new one.

Typos and Word Order

Typo-tolerant search. The system is able to load suggestions whose string is "similar" to the query the user is typing. To this end, it computes the *Edit* (Levenshtein) distance between user's current input and each full query of the trie that starts with the first character of user's input. If this number is lower than a threshold, the system adds that query to the list of suggestions and the suggestion is ranked as if no edit distance were employed.

Note that [5] and [3] study the problem of online spelling correction for query completions and both propose a trie-based approach and capture typos either via Edit distance [3] or a n-gram transformation model [5]. However, instant behavior is more challenging in our *partitioned* trie-based index because the system may have to access many subtries. For instance, if we have created subtries each corresponding to the first *two* characters and the user has typed "merilyn", then the system would have to access all subtries that correspond to the character sequences "ma", "mb", "mc", etc. For this reason, we decided to check only the subtries that lie in the cache and only in case we have no other suggestions to prompt.

Word-order independent search. The system can also load suggestions whose word order is different than that of the user's current input. Specifically, it checks for suggestions that start from one of the user's input words and contain at least one word of the remaining ones. The more of the remaining words a query contains, the higher score it receives. However, this aggravates the response time since the system has to perform more trie traversals and may have to access other subtries (especially if the user has typed many words). For this reason this functionality is activated only when there are no suggestions.

Benefits for the Service Provider

Apart from being useful for the end users, since without doing anything else than typing they can get instantly various useful information that assist them in expressing their information need and inspecting the available resources, IOS is beneficial also for the server's side since it:

(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),

(b) reduces the computational cost because the *same pre-*

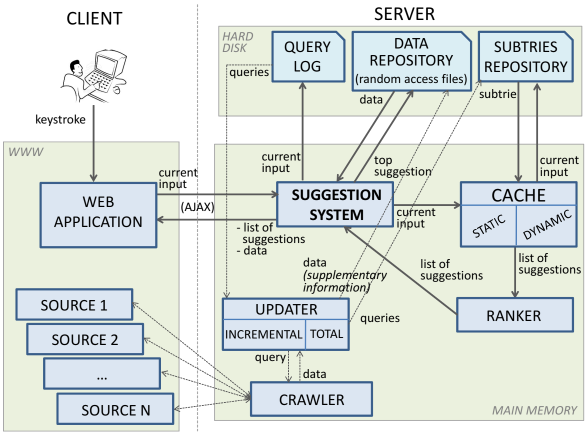


Figure 2: The architecture of IOS.

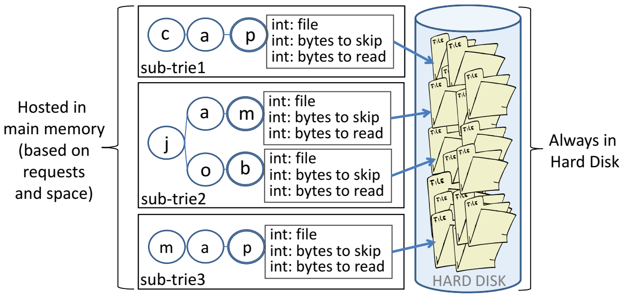


Figure 3: The PTIE partitioned trie-based index.

computed information is exploited in several requests and thus the engine has to evaluate less tasks at run time, (c) *reduces the monetary cost (at meta search level)*, i.e. if the construction of the "overview of the results" requires connecting to several external sources (e.g. web pages, LOD SPARQL endpoints, search engine APIs etc), this approach reduces the number of connections which are required. Furthermore, since in many cases the underlying services are not for free, i.e. billed according to the number of served queries, this approach reduces the queries sent to the underlying engines and thus can save money.

Related Work

There are several works describing letter-by-letter searching for various kinds of sources (e.g. textual or structured). To cope with the efficiency problem, most either need a high amount of main memory in order to load the indexing structures, or use disk-based compressed formats like [2]. The only work that does not load everything in main memory is the semantic search engine *CompleteSearch*[7] [1] which during user typing it finds on-the-fly and presents precomputed records and semantic classes that match these keywords. To tackle the efficiency challenge, the authors introduce a indexing data structure, called "HYB", which relies on compressed precomputed inverted lists for union of words (the union of all lists for an arbitrary word range). This index is not loaded in main memory but is stored in a single file with the individual lists concatenated and an array of list offsets at the end. The vocabulary is stored in a separate file. That work differs from ours in the following aspects: (1) CompleteSearch is not a generic search engine, but is based on specific known datasets (like Wikipedia and DBLP) with predefined semantic categories, (2) its suggestion system is word-based, not query-based, i.e. it suggests only words that match user's current input, not whole queries, (3) it focuses on compression of index structures, especially in disk-based settings, while IOS uses partial in-memory indexes, (4) in general CompleteSearch helps users to formulate a good semantic query while IOS helps users to locate directly and fast what they are looking for.

3. EXPERIMENTAL RESULTS

We have conducted experiments using a very large query log and large amounts of precomputed information. We used a synthetic query log of *one million distinct* queries and synthetic precomputed information for each query. Regarding trie partitioning, we created subtries each corresponding to the first two characters. This partitioning yields subtries whose sizes are very close [8]. Moreover, we chose to store at least 1,000 queries in each subtrie and 1,000 entries in each random access file (an entry represents the top hits and the supplementary information of a query). PTIE created 344 subtries of 615 MB in total and 992 random access files of about 1 terabyte in total. 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), using Apache Tomcat 6 (2GB of RAM) and Ajax JSP Tag Library.

Average Retrieval Time. We measure the average time for retrieving the suggestions, the results and the clusters for a random input string without using any cache (this does not contain the network's delay time and the javascript running time). We sequentially select 10,000 random queries from the query log, and for each one of them we take its first three characters, we find the subtrie that corresponds to this character sequence, we load it and traverse it to find all suggestions for that string. For the top suggestion, we access the corresponding random access file and retrieve its first page of results and its cluster label tree. The average retrieval time was about **135 ms**, proving that IOS is very efficient even for very large query logs and precomputed information.

Based on the experimental results of [6], we argue that the other indexing approaches (SET, PET, STIE) cannot be used in such a big number of queries and precomputed information. PTIE is scalable because the data that it has to load at request time has small size and its main memory cost is low.

Caching. To evaluate the benefits of caching we performed a comparative evaluation of the following policies: (1) full static cache, (2) full dynamic cache, (3) hybrid cache with 30% static and 70% dynamic, (4) hybrid cache with 50% static and 50% dynamic, (5) hybrid cache with 70% static and 30% dynamic, and (6) no cache. 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 to 60 subtries, i.e. 17.44% of all subtries can fit in main memory at the same time. The time threshold that a subtrie is considered in use (and thus it cannot be removed) was set to 10 seconds, i.e. this demands the ability to serve about $10 * 8 = 80$ queries simultaneously.

Served Queries. Fig. 4 reports (a) the percentage of

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

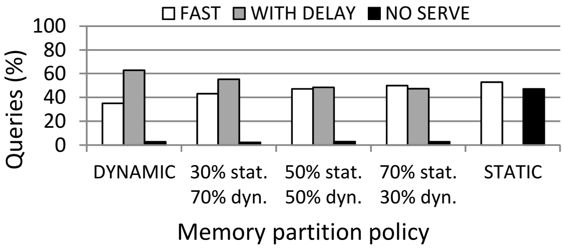


Figure 4: Percentage of queries that were served from the cache fast, with delay and that were not served, for various memory partition policies.

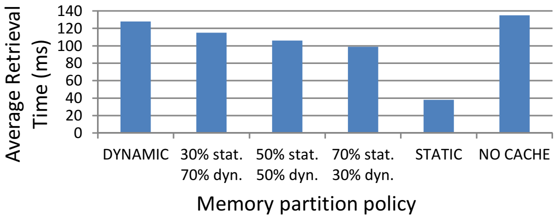


Figure 5: Average response time of various caching policies.

queries that were served from the cache without delay, (b) the percentage of queries that were served from the cache with delay (because the less frequent subtrie has to be removed and the requested one to be loaded), and (c) the percentage of queries that could not be served (because of memory overloading). We observe that as the cache becomes more static, more queries are served fast and less are served with delay. However, 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%).

Response Time (using Caching). Fig. 5 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. 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 simultaneously, 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 simultaneously. Thus, since we have 80 queries that have to be served together, 20 of them will not be served (25%).

From the above results it seems that the best caching policy is the hybrid with 70% static and 30% dynamic cache, as it combines low percentage of requests that cannot be served, high percentage of requests that are served fast and low average response time. Over the setting of the aforementioned experiment, the above 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.

Finally, we should clarify that the above results concern the case where each user requests only one subtrie, the subtrie that contains the randomly selected query. The results do not take into account the common scenario where 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.

4. CONCLUSION

We have described a generic method for enabling the instant provision of various enhanced search services in a cost-effective manner. This method is independent of the kind of precomputed information, hence it can be adopted for a plethora of applications (results clustering, metadata-based grouping, entity mining, query recommendations, semantic-based enhancements, etc). By exploiting this method systems like [4] and [12] could instantly present to the user relevant content for the most frequent queries, [11] could instantly suggest generalized links for each result of the frequent queries, while *Blognoon* [7] could offer instant search results and query recommendations for the most frequent queries.

We should clarify that the overall effectiveness of the provided functionality, apart from its efficiency, depends on the quality of the precomputed information. However we should stress that the proposed index structure and caching mechanism can be used with any autocompletion, ranking or clustering algorithm. Also note that its performance is independent of the size of the underlying corpus. It depends only on the number of distinct queries that we want to serve and the size of the precomputed information per query. A direction for further research is to measure the effect of `IOS` to user satisfaction and for this purpose we already gather user feedback by keeping a detailed log.

Acknowledgements

This work was partially supported by the EU project iMarine (FP7-283644).

5. REFERENCES

[1] H. Bast, A. Chitea, F. Suchanek, and I. Weber. ESTER: efficient search on text, entities, and relations. *SIGIR*, 2007.

[2] H. Bast and I. Weber. Type less, find more: fast autocompletion search with a succinct index. *SIGIR*, 2006.

[3] S. Chaudhuri and R. Kaushik. Extending autocompletion to tolerate errors. *SIGMOD*, 2009.

[4] L. Chilton and J. Teevan. Addressing people's information needs directly in a web search result page. *WWW*, 2011.

[5] H. Duan and B. Hsu. Online spelling correction for query completion. *WWW*, 2011.

[6] P. Fafalios and Y. Tzitzikas. Exploiting Available Memory and Disk for Scalable Instant Overview Search. *WISE*, 2011.

[7] M. Grineva, M. Grinev, D. Lizorkin, A. Boldakov, D. Turdakov, A. Sysoev, and A. Kiyko. Blognoon: exploring a topic in the blogosphere. *WWW*, 2011.

[8] D. Kastrinakis and Y. Tzitzikas. Advancing Search Query Autocompletion Services with More and Better Suggestions. *ICWE*, 2010.

[9] S. Kopidaki, P. Papadakos, and Y. Tzitzikas. STC+ and NM-STC: Two novel online results clustering methods for web searching. *WISE*, 2009.

[10] P. Papadakos, N. Armenatzoglou, S. Kopidaki, and Y. Tzitzikas. On exploiting static and dynamically mined metadata for exploratory web searching. *J.KAIS*, 30(3), 2012.

[11] J. Seo, F. Diaz, E. Gabrilovich, V. Josifovski, and B. Pang. Generalized link suggestions via web site clustering. *WWW*, 2011.

[12] X. Yin, W. Tan, and C. Liu. FACTO: a fact lookup engine based on web tables. *WWW*, 2011.