Efficient Fuzzy Type-Ahead Search in TASTIER

Guoliang Li [†], Shengyue Ji [‡], Chen Li [‡], Jiannan Wang [†], Jianhua Feng [†]

[†] *Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing 100084, China*

{liguoliang,fengjh}@tsinghua.edu.cn
wjn08@mails.tsinghua.edu.cn

[‡] *Department of Computer Science, UC Irvine, CA 92697-3435, USA*

{shengyuj,chenli}@ics.uci.edu

***Abstract*— TASTIER is a research project on the new information-access paradigm called *type-ahead search*, in which systems find answers to a keyword query on-the-fly as users type in the query. In this paper we study how to support *fuzzy* type-ahead search in TASTIER. Supporting fuzzy search is important when users have limited knowledge about the exact representation of the entities they are looking for, such as people records in an online directory. We have developed and deployed several such systems, some of which have been used by many people on a daily basis. The systems received overwhelmingly positive feedbacks from users due to their friendly interfaces with the fuzzy-search feature. We describe the design and implementation of the systems, and demonstrate several such systems. We show that our efficient techniques can indeed allow this search paradigm to scale on large amounts of data.**

I. INTRODUCTION

Keyword search is important in information systems. When using most Web search systems, a user types a *complete* query and waits for results from the server. In the case where users have limited knowledge about the data or do not know the exact keywords of the entities they are looking for, often they feel "left in the dark" when issuing queries, and have to use a try-and-see approach for finding information. Many systems are introducing various features to solve this problem. One of the commonly used methods is *autocomplete*, which predicts a word or phrase that the user may type in based on the partial string the user has typed in. As an example, almost all the major search engines nowadays automatically suggest possible keyword queries as a user types in partial keywords.

One limitation of traditional autocomplete is that the system treats a query with multiple keywords as a *single string*, thus it does not do a full-text search on the data. For instance, consider the search box on Apple.com, which allows auto-complete search on Apple products. Although a keyword query "itunes" can find a record "itunes wi-fi music store," a query "itunes music" cannot find this record (as of October 2009), simply because this query string *as a whole* does not appear in the record.

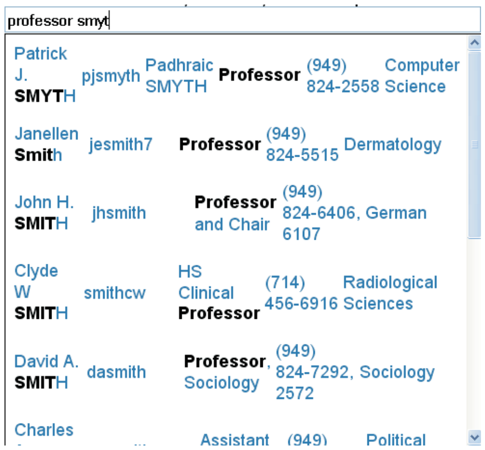
Beyond treating a query as a single prefix: To address this problem, recently a new type-ahead search paradigm has emerged. Such a system treats the query as a set of keywords, and finds answers with these keywords. It does a full-text search on the underlying data "on the fly" as the user types in query keywords letter by letter. In this way, the user can get instant feedback after typing a partial query, thus obtain more knowledge about the underlying data, which helps the user formulate queries. Bast et al. [1], [2], [3] described several techniques to do this type of search. An example is the CompleteSearch system on DBLP[1], which can find publications that match multiple keywords in a query interactively.

To study how to support efficient type-ahead search on large amounts of data, we started a project called "TASTIER", which stands for "**t**ype-**a**head **s**earch **t**echn**i**qu**e**s in la**r**ge data sets"[2]. In this paper we focus on how to support *fuzzy* type-ahead search in TASTIER [4]. With our techniques, a type-ahead system can find answers with keywords *similar* to the keywords in a query. It is based on the following motivation. Often users can make mistakes when they type in queries, especially when they have limited knowledge about the data. For instance, a user looking for the publications by Christos Faloutsos might not know the exact spelling of the author name (Figure 1). Our techniques are also useful when there are errors and inconsistencies even in the data itself.

Query performance is a key issue in designing such a fuzzy type-ahead search system, since there could be more queries submitted to the system than a traditional system, and each query should be answered within milliseconds. In this paper we describe several such systems developed using our techniques. We describe their design and implementation, and use performance numbers to show that our techniques can indeed make this search paradigm scale on large amounts of data. Compared to our earlier publications, here we mainly focus on the architecture and demonstrations of these systems.

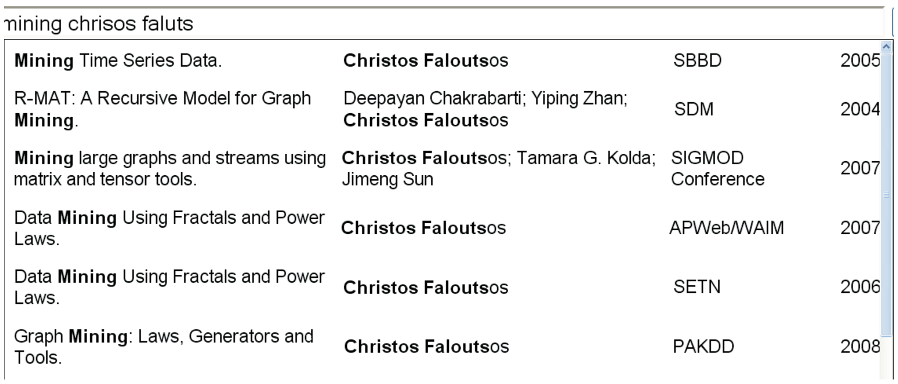
A possible concern about these systems is their "disruptive-ness," i.e., each keystroke from the user could invoke a query on the server. We address this concern using the following facts. (1) "Search-as-you-type" interfaces have been widely adopted in many search engines and Web services. (2) In the database community, the recently deployed CompleteSearch DBLP system with this feature has been well received. (3) We have deployed several systems with similar features, and

[1]http : //dblp.mpi-inf.mpg.de/dblp-mirror/index.php
[2]tastier.cs.tsinghua.edu.cn and tastier.ics.uci.edu


professor smyt

Patrick J. **SMYT**H	pjsmyth	Padhraic SMYTH	**Professor**	(949) 824-2558	Computer Science
Janellen **Smit**h	jesmith7	**Professor**	(949) 824-5515	Dermatology	
John H. **SMIT**H	jhsmith	**Professor** and Chair	(949) 824-6406, 6107	German	
Clyde W **SMIT**H	smithcw	HS Clinical **Professor**	(714) 456-6916	Radiological Sciences	
David A. **SMIT**H	dasmith	**Professor**, Sociology	(949) 824-7292, 2572	Sociology	
Charles		Assistant	(949)	Political	

(a) People Search on UCI Phonebook dataset


mining chrisos faluts

Mining Time Series Data.	**Christos Faloutsos**	SBBD	2005
R-MAT: A Recursive Model for Graph **Mining**.	Deepayan Chakrabarti; Yiping Zhan; **Christos Faloutsos**	SDM	2004
Mining large graphs and streams using matrix and tensor tools.	**Christos Faloutsos**; Tamara G. Kolda; Jimeng Sun	SIGMOD Conference	2007
Data **Mining** Using Fractals and Power Laws.	**Christos Faloutsos**	APWeb/WAIM	2007
Data **Mining** Using Fractals and Power Laws.	**Christos Faloutsos**	SETN	2006
Graph **Mining**: Laws, Generators and Tools.	**Christos Faloutsos**	PAKDD	2008

(b) Publication Search on DBLP dataset

Fig. 1. Two screenshots of two fuzzy type-ahead search systems

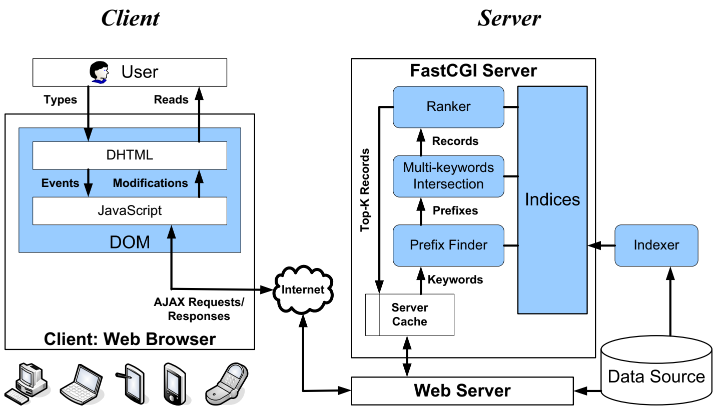


Fig. 2. Fuzzy type-ahead search architecture

received very positive feedbacks from users due to the friendly interfaces and high efficiency. (4) If needed, we do not need to submit a query to the server for each keystroke. In each of our systems, the client sends a single query possibly after multiple letters are typed in if these letters were typed in when the server was still processing the previous query. We can easily add a delay on the client side after a keystroke (using JavaScript) for users who type in a query very fast. This feature is especially useful when the user initially types in a query quickly, and pauses to digest the information from the server. A big advantage of this type-ahead interface is to allow users to *explore* the data when formulating a query. Chaudhuri et al. [5] studied how to find similar strings interactively as users type in a query string, using an approach similar to that in [4].

II. System Architecture and Implementation

Figure 2 illustrates the client-server architecture of a system using our TASTIER techniques. We assume the underlying data is a set of records residing on a server. Our method can be extended to support type-ahead search on documents, XML data [6], and relational databases [7]. The client is a Web browser. A user uses the Web browser to send requests to the server over the Internet and see the results from the server. Each keystroke of the user could invoke a query, which includes the current string the user has typed in. The browser sends the query to the server. The server tokenizes the query string, computes and returns to the user the best answers ranked by their relevancy to the query. Figure 1 gives two screenshots of fuzzy type-ahead search on a DBLP dataset and a MEDLINE dataset in the medical domain.

For each query sent to the server, we treat the last keyword as a *partial keyword* the user is completing, and other earlier keywords as *complete keywords*.[3] For each complete keyword, we identify the keywords in the data that are similar to the keyword. For the partial keyword, we identify its *similar keywords* as those in the data with a prefix similar to the partial keyword. We use edit distance to quantify the similarity between two words w_i and w_j, denoted as $\mathsf{ed}(w_i, w_j)$. The *edit distance* between two words is the minimum number of edit operations (i.e., insertion, deletion, and substitution) of single characters needed to transform the first one to the second. For example, $\mathsf{ed}(\texttt{feloutose}, \texttt{faloutsos}) = 3$. We say two keywords are similar if their edit distance is within a given threshold δ. This threshold could be proportional to the length of a query keyword to allow more errors for longer keywords. We compute the *relevant records* that contain a similar keyword for every keyword, and return the most relevant records ranked by their relevancy to the query.

There are several components on the server side. The Indexer component indexes the underlying data as a trie structure with inverted lists of keywords in the leaf nodes. We build a FastCGI module on the Web server to store the data and indices. Different from a CGI module, the FastCGI server module is loaded once when the Web server starts, and continually handles queries without spawning more instances. Therefore the server loads the data and indices from the disk once, and then searches on the data in memory without accessing the disk. The FastCgi Server waits for queries from the

[3]Our results generalize naturally to the case where each keyword is treated as a prefix.

client, and caches query results. The Server Cache component checks whether the query can be answered using the cached results. If not, the server incrementally answers the query by using the cached information. For each query keyword, the Fuzzy Prefix Finder incrementally computes its *similar keywords*. The Multi-keyword Intersection module computes the *relevant answers* that contain at least one similar keyword for every input keyword. The Ranker module ranks the answers to identify the top-k best answers for a constant k.

A. Server Design

We present the design of the server modules. We chose C++ to build the server module due to its high performance.

Indexer: It is an offline process that reads data from specified sources, tokenizes the data, and creates the following structures: (1) a radix trie structure with inverted lists on the corresponding leaf nodes; (2) a forward index that stores the sorted list of keyword IDs for each record; (3) the data itself. Each word w in the data corresponds to a unique path from the root of the trie to a leaf node. Each node on the path has a label of a character in w. The nodes with the same parent are sorted by the node label in their alphabetical order. For each leaf node, we store an inverted list of IDs of records that contain the corresponding word. To improve performance, optionally we can also maintain a forward index, which keeps the sorted keyword IDs for each record.

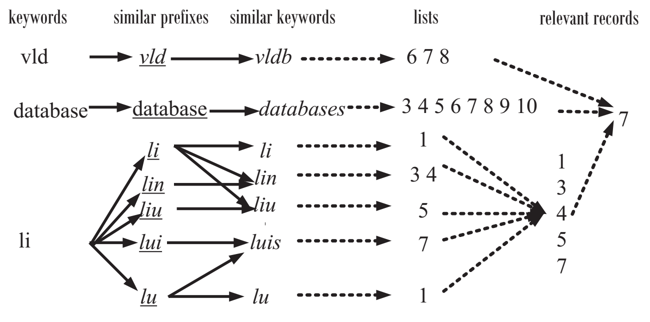
Incremental Fuzzy Prefix Finder: It is part of the FastCGI module. In the case of exact search, there exists only one trie node corresponding to a partial keyword. However, to support fuzzy search, we need to compute multiple prefixes that are *similar* to the partial keyword, and retrieve their corresponding complete keywords as the *similar keywords*. The Incremental fuzzy prefix finder incrementally identifies the prefixes in the dataset that are similar to the query keywords. The idea of our method is to use prefix filtering. That is, when the user types in one more letter after the partial keyword, only the descendants of the trie nodes of similar prefixes of the partial keyword could be potentially similar prefixes of the new query keyword. We use this property to incrementally compute the similar prefixes of a new query. For a new query, the Incremental fuzzy prefix finder first looks up similar prefixes of previous queries from the server cache, computes similar prefixes for the current query incrementally, and stores the results in the cache for future computation.

Multi-keyword Intersection: This module takes the sets of similar keywords produced by the fuzzy prefix finder as input (for multiple keywords), and computes the *relevant answers*, which contain a matching similar keyword from each set. For the partial keyword, there could be multiple similar prefixes, and each similar partial prefix has multiple similar keywords. We call the union of each keyword's similar keywords' inverted lists the *union list* for this keyword. A straightforward method to identify the relevant answers is to first construct the union list for every keyword, and then compute the intersection of the union lists. However, it is rather expensive to construct these union lists on-the-fly. Figure 3(a) illustrates an example in which we want to answer query "`li database vld`".

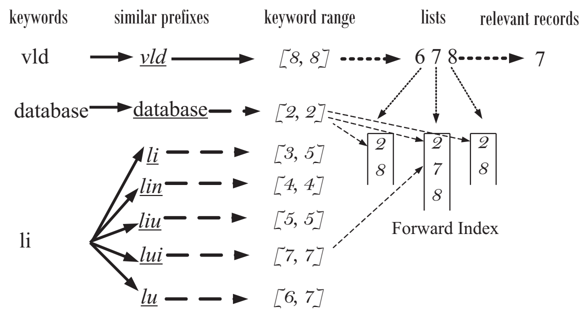
We can use forward lists to improve the performance of computing the intersection. We choose the keyword with the shortest union list based on estimation. We use the forward index to check whether each candidate record on the shortest union list contains similar keywords of every other query keyword. If so, this record is an answer. To do this checking efficiently, in the trie structure, each leaf node has a unique keyword ID for the corresponding word. The keyword ID is assigned in their pre-order on the trie. Each trie node maintains the range of the keyword IDs in its subtrie. For the keyword range of each similar prefix of other keywords, for example, $[s, \ell]$, we check whether the candidate record on the shortest union list contains keywords in the range. We first use a binary-search method to find the keyword ID on the corresponding forward list. We get the smallest keyword ID on the list that is larger than or equal to s. Then we check whether the keyword ID is smaller than ℓ. If so, this candidate contains a keyword in the range. Figure 3(b) illustrates this method using the running example.

Ranker: In order to compute high-quality results, we need to use a good ranking function for the candidates. The function should consider various factors such as the similarity between a query keyword and its similar prefixes, the weight of each keyword, term frequencies, inverse document frequencies, importance of each record, etc. If the edit distance between an input keyword and its similar prefixes dominates the other parameters, we want to compute the answer with the smallest edit distance first. If there are not enough top answers with edit distance τ, we then compute answers with an edit distance $\tau + 1$, and so on.

Server Cache: After finding the answers to a query, we cache the similar prefixes of each input keyword. Accordingly, we can incrementally answer the subsequent keyword queries using the cached similar prefixes. For the query with multiple keywords, we also cache the relevant answers. If the user types another keyword, we use the cached results to answer the query by checking whether the cached results contain the new keyword using the forward index. If there are too many relevant records, we can just cache the highly relevant ones. For each subsequent keyword, we first use the cached records to compute the answer. If there are not enough top answers, we continue to compute more answers for the previous query and store the results in the cache. This "on-demand" caching method can make sure each query can be answered very efficiently, and we can cache results of a query only if they are necessary. We can also postpone some unnecessary computation when the user has more keystrokes. In our design, not only the results but also the search context at the termination point were saved for future computation. Therefore, for a subsequent query, the system can use the cached results of previous queries to answer it. If needed, the system will also resume the search from the saved context until top-k results are retrieved.



(a) By intersecting union lists



(b) By probing lists

Fig. 3. Two methods for answering a keyword query "`li database vld`"

B. Client and Communication Design

The client side contains HTML contents with JavaScript code interpreted or executed in the browser. When the user types in a query, if there is no pending request being processed by the server the JavaScript code issues an AJAX query to the server. Otherwise, it waits until the request has been answered. This is to avoid the case where the user types so fast that the system is overloaded. The query results are returned in a JSON format, and the matched prefixes are returned along with the records. We highlight matching prefixes.

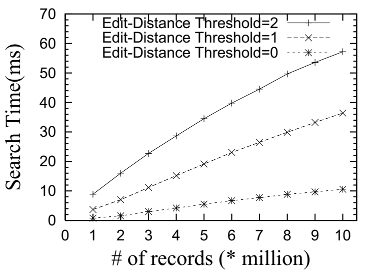
III. Demonstrations

We developed several systems based on our techniques of fuzzy type-ahead search. We will demonstrate the following systems. (1) People Search (http://psearch.ics.uci.edu/): It searches on the UCI people directory. (2) Search on DBLP authors (http://dblp.ics.uci.edu/authors/): It searches in authors with DBLP publications. (3) DBLP Search (http://dblp.ics.uci.edu/): It searches on more than one million DBLP publications. (4) Search on URL (http://tastier.cs.tsinghua.edu.cn/urlsearch/): It searches on 10M widely used URLs. In the experiments all queries can be processed within 80 milliseconds per query. Our method has a good scalability as illustrated in Figure 4 (See [4], [6], [7] for more experimental results).

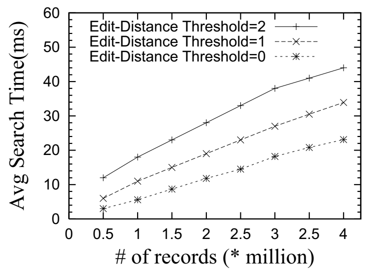
In addition to the feature of fuzzy-type-ahead search, we will also demonstrate the following features.

Highlighting Similar Prefix: We will show how to highlight a prefix in the results that best matches a keyword. Highlighting is straightforward for the case of exact matching, since each keyword must be a prefix of the matching keyword. For the case of fuzzy matching, a query keyword may not be an exact prefix of a similar keyword. Instead, the query keyword is just similar to some prefixes of the similar keyword. Thus, there can be multiple similar keywords to highlight. For example, suppose a user types in "`lus`", and there is a similar keyword "`luis`". Both prefixes "`lui`" and "`luis`" are similar to "`lus`". There are several ways to highlight "`luis`", such as "**lui**`s`" or "**luis**". We highlight the longest matched one ("**luis**").

Using Synonyms: We can utilize a-priori knowledge about synonyms to find relevant records. For example, in the domain of person names, "`William` = `Bill`" is a synonym. Suppose in the underlying data, there is a person called "`William Kropp`". If a user types in "`Bill Cropp`", we can also find this person. To this end, on the trie, the node corresponding to "`Bill`" has a link to the node corresponding to "`William`", and vise versa. When a user types in "`Bill`", in addition to retrieving the relevant records for "`Bill`", we also identify those of "`William`" following the link. In this way, our method can be easily extended to utilize synonyms.



(a) URL dataset



(b) PubMed dataset

Fig. 4. Scalability

References

[1] H. Bast and I. Weber, "Type less, find more: fast autocompletion search with a succinct index," in *SIGIR*, 2006, pp. 364–371.

[2] H. Bast, A. Chitea, F. M. Suchanek, and I. Weber, "Ester: efficient search on text, entities, and relations," in *SIGIR*, 2007, pp. 671–678.

[3] H. Bast and I. Weber, "The completesearch engine: Interactive, efficient, and towards ir& db integration," in *CIDR*, 2007, pp. 88–95.

[4] S. Ji, G. Li, C. Li, and J. Feng, "Interative fuzzy keyword search," in *WWW 2009*, 2009, pp. 371–380.

[5] S. Chaudhuri and R. Kaushik, "Extending autocompletion to tolerate errors," in *SIGMOD*, 2009, pp. 707–718.

[6] G. Li, J. Feng, and L. Zhou, "Interactive search in xml data," in *WWW*, 2009, pp. 1063–1064.

[7] G. Li, S. Ji, C. Li, and J. Feng, "Efficient type-ahead search on relational data: a tastier approach," in *SIGMOD*, 2009, pp. 695–706.