Search-As-You-Type: Opportunities and Challenges

Chen Li[†] and Guoliang Li[‡]
[†]Department of Computer Science, University of California, Irvine, USA
chenli@ics.uci.edu
[‡]Department of Computer Science and Technology, Tsinghua University, Beijing, China
liguoliang@tsinghua.edu.cn

Abstract

Traditional information systems return answers after a user submits a complete query. Users often feel "left in the dark" when they have limited knowledge about the underlying data, and have to use a try-and-see approach for finding information. A recent trend of supporting autocompletion in these systems is a first step towards solving this problem. In this paper, we study a new information-access paradigm, called "search-as-you-type" or "type-ahead search," in which the system searches the underlying data on the fly as the user types in query keywords. It extends traditional prefix-based autocompletion interfaces by supporting full-text search on the data using tokenzied query keywords. We give an overview of this information-access paradigm, discuss research challenges and opportunities, and report recently developed techniques.

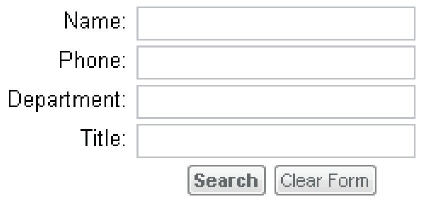
1 Introduction

Traditional information systems allow users to compose and submit a query to retrieve relevant answers. This information-access paradigm requires the user to have certain knowledge about the structure and content of the underlying data repository. With limited knowledge about the data, a user often feels "left in the dark" when issuing queries, and the user and has to use a try-and-see approach for finding information. For instance, Figure 1 shows a traditional interface to search on the people directory of an organization. To find a person, a user needs to fill in the form by providing information for multiple attributes, such as name, phone, department, and title. If the user has limited information about the person she is looking for, such as the exact spelling of the person's name, the user needs to try a few possible keywords, go through the returned results, modify the keywords, and reissue a new query. She needs to repeat this step multiple times to find the person, if lucky enough. This search interface is neither efficient nor user friendly.

To solve this problem, many systems provide instant feedback as users formulate search queries. Most search engines and many online search forms support autocompletion, which shows suggested queries or even answers "on the fly" as a user types in a keyword query letter by letter. For instance, consider the Web search interface at Netflix (http://www.netflix.com/BrowseSelection), which allows a user to search for movies by their titles, actors, directors, and genres. If a user types in a partial query "`mad`", the system shows movies matching this keyword as a prefix, such as "`Madagascar`" and "`Mad Men: Season 1`".

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

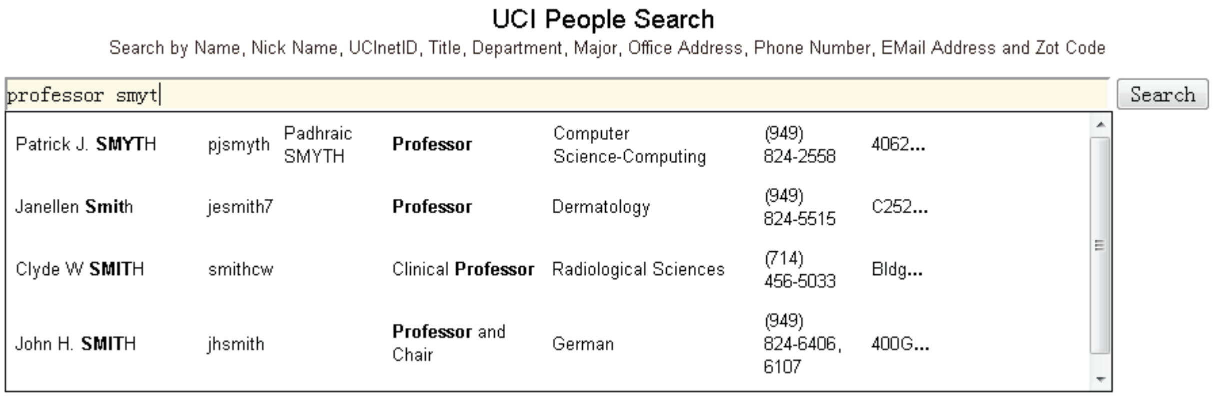
The quick feedback helps the user not only in formulating the query, but also in understanding the underlying data. Most autocompletion systems make suggestions by simply treating a query as a *prefix* condition. As a result, a Netflix user cannot type in "`Spielberg sci-fi`" to find sci-fi movies directed by Steven Spielberg, since the system treats the query as a prefix string, which does not exist in any movie record.



Name:	
Phone:	
Department:	
Title:	

Figure 1: A traditional directory-search form.

Recently a new type-ahead-search paradigm has arisen that generalizes prefix-based autocompletion. In this paradigm, a system supports *full-text* search on the underlying data to find answers as a user types in query keywords. We have developed several prototypes using this paradigm. The first one, called PSearch, supports search on the UC Irvine people directory. A screenshot is shown in Figure 2. In the figure, a user has typed in a query string "`professor smyt`." Even though the user has not typed in the second keyword completely, the system can already find person records that might be of interest to the user. Notice that the two keywords in the query string (including a partial keyword "`smyt`") can appear in different attributes of the records. In particular, in the first record, the keyword "`professor`" appears in the "title" attribute, and the partial keyword "`smyt`" appears in the "name" attribute. The matched prefixes are highlighted. The system also utilizes a-priori knowledge such as synonyms. For instance, given the fact that "`william`" and "`bill`" are synonyms, the system can find a person called "`William Kropp`" when the user has typed in "`bill crop`." This search prototype has been used regularly by many people at UCI, and received positive feedback due to the friendly user interface and high efficiency.



UCI People Search

Search by Name, Nick Name, UCInetID, Title, Department, Major, Office Address, Phone Number, EMail Address and Zot Code

```
professor smyt
```

Patrick J. **SMYT**H	pjsmyth	Padhraic SMYTH	**Professor**	Computer Science-Computing	(949) 824-2558	4062...
Janellen **Smit**h	jesmith7		**Professor**	Dermatology	(949) 824-5515	C252...
Clyde W **SMIT**H	smithcw		Clinical **Professor**	Radiological Sciences	(714) 456-5033	Bldg...
John H. **SMIT**H	jhsmith		**Professor** and Chair	German	(949) 824-6406, 6107	400G...

Figure 2: Fuzzy type-ahead search on the UC Irvine people directory (http://psearch.ics.uci.edu).

There are several other systems developed using this search paradigm: (1) The "Search" box on the page http://www.ics.uci.edu that searches on the people of the school of ICS at UCI and its important internal pages; (2) A system called "iPubMed" (http://ipubmed.ics.uci.edu) that supports interactive, fuzzy search on more than 18 million MEDLINE publications; (3) A prototype (http://tastier.cs.tsinghua.edu.cn/urlsearch/) that supports searches on 10 million popular URLs [11]; and (4) A search interface (http://fr.ics.uci.edu/haiti) for family reunification for the recent Haiti earthquake.

In this paper, we give an overview of this information-access paradigm, discuss research challenges and opportunities, and report recently developed techniques.

2 Overview of Type-ahead Search

Figure 3 illustrates a client-server architecture of a system supporting search-as-you-type. The underlying data residing on the server can be a relational database [10], a collection of documents, or XML data [9]. For simplicity, in this paper we focus on the case where the server has a set of relational records. The client has a browser, using which a user can send requests to the server to retrieve results. 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. For each query, the server treats the last keyword as a *partial keyword* the user is completing, and other earlier keywords as *complete keywords*. For a keyword query, we want to find the records that contain every complete keyword in the query and a keyword with the partial keyword as a prefix. For example, in Figure 2, for a keyword query "`professor smyt`", the keyword "`smyt`" is a partial keyword, while "`professor`" is a complete keyword. The first record in the figure is an answer since it contains keyword "`professor`" and a keyword "`smyth`" with the prefix "`smyt`".

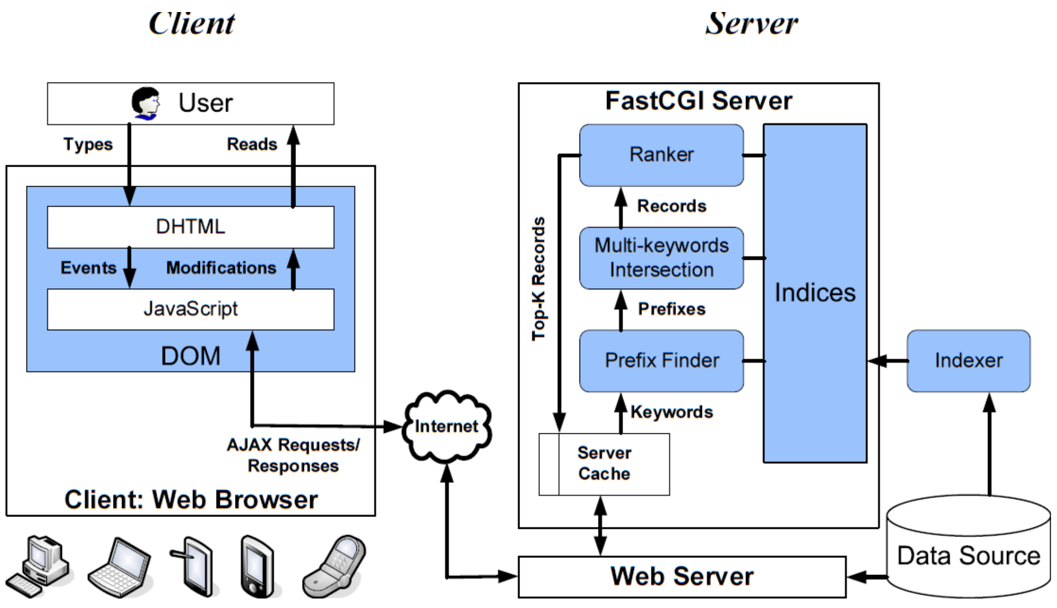


Figure 3: Search-as-you-type system architecture.

Supporting *fuzzy search* is very important especially when users do not remember the exact spelling of the right keywords. To support this feature, for each complete keyword in a query, 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 compute the relevant records that contain a similar keyword for every query keyword. We can use edit distance to quantify the similarity between keywords. 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, the edit distance of "`feloutose`" and "`faloutsos`" is 3. We say two keywords are *similar* if their edit distance is within a given threshold τ. This threshold could be proportional to the length of a query keyword to allow more errors for longer keywords. For example, suppose the edit-distance threshold $\tau = 1$. In Figure 2, for keyword query "`professor smyt`", the second record is an answer, since it contains keyword "`professor`" and a keyword "`smith`" with a prefix "`smit`" similar to input keyword "`smyt`".

2.1 Client

The client side contains HTML contents with JavaScript code 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. In this way, we can avoid overloading the server if the user types very fast.

2.2 Server

As shown in Figure 3, there are several components on the server side. We use a FastCGI module to store the data and indices. (Other programming languages such as Java Servlets are also possible.) The FastCGI server module is loaded once when the Web server starts, and continually handles queries without spawning more instances. The server loads the data and indices from disks, and searches on the data. The FastCgi Server waits for queries from the 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 Prefix Finder incrementally computes keywords of partial keywords. The Multi-keyword Intersection module computes the relevant answers. The Ranker module ranks the answers to identify the best answers. The Indexer component indexes the underlying data. Now we explain the modules in more detail.

Prefix Finder: For exact search, it finds the keywords with a prefix of the 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. In the query "`professor smyt`", for exact search, this module finds complete keywords with the prefix "`smyt`", such as "`smyth`". For fuzzy search, it finds complete keywords with a prefix similar to "`smyt`," such as "`smyth`" and "`smith`".

Multi-keyword Intersection: This module takes the sets of keywords produced by the prefix finder as input (for multiple keywords), and computes the relevant answers, which contain a matching keyword from each set. For the partial keyword, there could be multiple keywords, and each similar partial prefix has multiple similar keywords. The union of the inverted lists of those keywords similar to one keyword is called the "union list" for this keyword. A straightforward method to identify the relevant answers is to first construct the union list for every query keyword, and compute the intersection of the union lists. In our running example, for exact search, based on keywords "`professor`" and "`smyth`", this module computes the intersection of inverted lists of the two keywords. For fuzzy search, we first compute the union lists of "`smyt`", which is the union of inverted lists of similar keywords of "`smyt`" (e.g., "`smyth`" and "`smith`"). Then we compute the intersection of the union lists of "`professor`" and "`smyt`". More efficient algorithms have been developed for solving this problem [7].

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.

Server Cache: After finding the answers to a query, we can cache some information, and incrementally answer the subsequent keyword queries using the cached information. For instance, suppose the results of the keyword query "`professor smyt`" have been cached. If the user types in one more letter and submits a new query "`professor smyth`", we can use the cached information to answer the new query.

Indexer: To improve performance, we can create index structures for efficiently answering type-ahead search, which will be discussed in the next section.

3 Research Challenges

There are several unique challenges in type-ahead search, mainly due to the requirement on a high interactive speed and the capability of relaxing keyword conditions. Each keystroke from the user can invoke a query on the backend server. The total round-trip time between the client browser and the backend server includes the network delay and data-transfer time, query-execution time on the server, and the javascript-execution time on the client browser. In order to achieve an interactive speed, this total time should be short (typically within 100ms). The query-execution time on the server should be even shorter. This high speed is more challenging to achieve when we need to relax the keywords on-the-fly. At a high level, a main challenge is: *in type-ahead search we need to support the features available in traditional search systems as the user types in keywords character by character.* Notice that in traditional autocompletion, we can easily recommend good queries by traversing a trie structure. However, when we allow keywords to appear at different places in the answers, the problem becomes "search the data on the fly," rather than just recommending a few queries. As a consequence, the join nature of the online-search problem can be computationally costly.

In this section, we present several recently developed techniques to address these challenges and discuss open problems that need more research investigation.

3.1 Efficient Indexing and Ranking

To facilite prefix search, we can construct a trie structure with inverted lists on the leaf nodes [3, 7]. Each word in the data set 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 the word. For each leaf node, we store an inverted list of IDs of records that contain the corresponding word. Figure 4 gives an example trie structure, in which node 12 has an inverted list of records 3 and 4, when records 3 and 4 contain the corresponding keywords "`lin`".

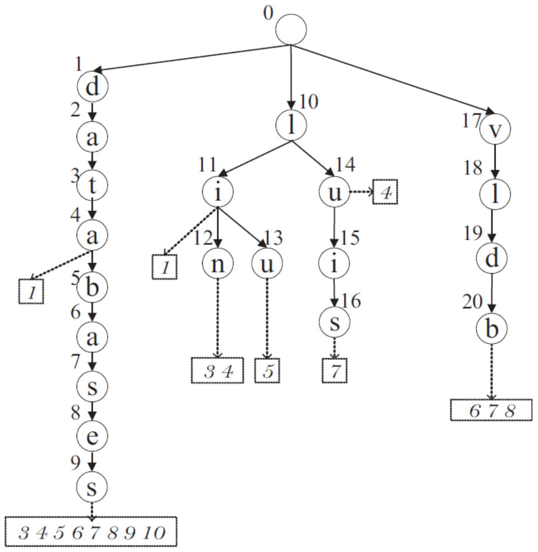


Figure 4: Trie structure with inverted lists on leaf nodes.

Exact Prefix Search: Given a partial keyword, we want to find the keywords with a prefix of the partial keyword. We can use the trie structure to answer prefix queries as follows. We first find the corresponding trie node of the prefix, and then traverse the subtrie to get the keywords with a prefix of the input keyword. Finally, we compute the records on the inverted lists of the leaf nodes. For example, consider the trie structure in Figure 4. When a user types in a keyword "lu", we find the trie node 14, and retrieve the corresponding leaf nodes "luis," and get record 7. When the user types in a new keyword "lui", we find trie node 15. In this way, we can use the trie structure to efficiently find the complete keywords of the partial keyword.

Fuzzy Prefix Search: For the case of exact prefix search using a trie index, there can be at most one trie node corresponding to a query prefix. The solution to the problem of fuzzy search is more challenging since a keyword prefix can have multiple similar prefixes, and we need to compute them efficiently. Traditional gram-based methods [8] are inefficient in this case due to their poor pruning power for short strings.

There are recent studies on supporting fuzzy prefix search efficiently [7, 5]. The idea is the following. When the user types in one more letter after a query prefix p, the similar prefixes of p can be used to compute the similar prefixes of the new query. Specifically, when the user types in one more letter after the partial keyword p, only the similar prefixes of p and their descendants could be similar prefixes of the new query keyword. We can use this property to incrementally compute the similar prefixes of a new query. For a new query, we first find similar prefixes of previous queries from the server cache, compute similar prefixes for the current query incrementally, and store the results in the cache for future computation. For example, consider the trie structure in Figure 4. Assume a user types in a query "nlis" letter by letter. Suppose two strings are considered to be similar if their edit distance is within 2. First, the similar prefixes of the empty string are nodes 0, 10, 11, and 14. When the user types in the first character "n", we compute its similar prefixes based on that of the empty string as follows. For node 0, since we can delete the letter "n", it is a similar prefix of "n". For node 10, which is a child of node 0 with a letter "l", as we can substitute "l" for "n", it is a similar prefix of "n". In this way, we can compute the similar prefixes of "n".

Multi-Keyword Intersection: The goal of multi-keyword intersection is to efficiently and incrementally compute the relevant records based on the keywords generated from the prefix finder. In [7] we studied the following problems. (1) *Intersection of multiple lists of keywords*: Each query keyword (treated as a prefix) has multiple predicted complete keywords, and the union of the lists of these complete keywords includes potential answers. The union lists of multiple query keywords need to be intersected in order to compute the answers to the query. These operations can be computationally costly, especially when each query keyword can have multiple similar prefixes. (2) *Cache-based incremental intersection*: Users tend to type in a query letter by letter. Thus we can use the cached results of earlier queries to answer a query incrementally. We use an example to illustrate how to cache query results and use them to answer subsequent queries. Suppose a user types in a keyword query Q_1 = "cs co". All the records in the answers to Q_1 are computed and cached. For a new query Q_2 = "cs conf" that appends two letters to the end of Q_1, we can use the cached results of Q_1 to answer Q_2, because the second keyword "conf" in Q_2 is more restrictive than the corresponding keyword "co" in Q_1. Each record in the cached results of Q_1 is verified to check if "conf" can appear in the record as a prefix. In this way, Q_2 does not need to be answered from scratch.

3.2 Ranking

A good ranking function needs to consider various factors to compute an overall relevance score of a record to a query. The following are important factors. (1) *Matching prefixes*: We consider the similarity between a query keyword and its best matching prefix. The more similar a record's matching keywords are to the query keywords, the higher this record should be ranked. The similarity is also related to keyword length. For example, when a user types in a keyword "circ", the word "circle" is possibly more similar to the query keyword than "circumstance". Therefore records containing the word "circle" could be ranked higher than those with the word "circumstance". Exact matches on the query should have a higher weight than fuzzy matches. (2) *Predicted keywords*: Different predicted keywords for the same prefix can have different weights. One way to assign a score to a keyword is based on its inverse document frequency (IDF). (3) *Record weights*: Different records could have different weights. For example, a publication record with many citations could be ranked higher than a less cited publication.

As an example, the following is a scoring function that combines the above factors. Suppose the query Q has keywords $p_1, p_2, \ldots, p_\ell$, while p'_i is the best matching prefix for p_i, and k_i is the best predicted keyword for p'_i. Let $sim(p_i, p'_i)$ be an edit similarity between p'_i and p_i. The score of a record r for Q can be defined as:

$$Score(r, Q) = \sum_i [sim(p_i, p'_i) + \alpha \cdot (|p'_i| - |k_i|) + \beta \cdot score(r, k_i)],$$

where α and β are coefficients ($0 < \beta < \alpha < 1$), $sim(p_i, p'_i) = \frac{\mathsf{ed}(p_i, p'_i)}{|p'|}$, and $score(r, k_i)$ is a score of record r for keyword k_i.

One technical challenge is how to utilize a ranking function in the search process to compute the top answers efficiently. While traditional top-k algorithms (e.g., [6]) could be utilized, new techniques need to be developed specifically for the unique index structure in type-ahead search to support efficient list access and pruning.

3.3 Additional Features

Keyword Highlighting: When displaying records to users, we want to highlight the most similar prefixes for an input prefix. This highlighting feature is straightforward for the exact-match case. For fuzzy search, a query prefix could be similar to several prefixes of the same predicted keyword. Thus, there could be multiple ways to highlight the predicted keyword. For example, suppose a user types in "`lus`", and there is a predicted keyword "`luis`". Both prefixes "`lui`" and "`luis`" are similar to "`lus`". There are several ways to highlight them, such as "**lui**s" or "**luis**", where underlined characters are highlighted. To address this issue, we use the concept of *normalized edit distance*. Formally, given two prefixes p_i and p_j, their normalized edit distance is:

$$\mathsf{ned}(p_i, p_j) = \frac{\mathsf{ed}(p_i, p_j)}{max(|p_i|, |p_j|)}, \tag{1}$$

where $|p_i|$ denotes the length of p_i. Given an input prefix and one of its predicted keywords, the prefix of the predicted keyword with the minimum ned to the query is highlighted. We call such a prefix a *best matched prefix*, and call the corresponding normalized edit distance the "minimal normalized edit distance," denoted as "mned." This prefix is considered to be most similar to the input keyword. For example, for the keyword "`lus`" and its predicted word "`luis`," we have ned("`lus`", "`l`") = $\frac{2}{3}$, ned("`lus`", "`lu`") = $\frac{1}{3}$, ned("`lus`", "`lui`") = $\frac{1}{3}$, and ned("`lus`", "`luis`") = $\frac{1}{4}$. Since mned("`lus`", "`luis`") = ned("`lus`", "`luis`"), "**luis**" will be highlighted.

Supporting Synonyms: We can also utilize a-priori knowledge about synonyms to find relevant records. For example, "`William` $=$ `Bill`" is a common synonym in the domain of person names. Suppose in the underlying data, there is a person called "`Bill Gates`". If a user types in "`William Gates`", we want to find this person. One way to support this feature is the following. On the trie, the node corresponding to "`Bill`" has a link to the node corresponding to "`William`", and vice versa. When a user types in "`Bill`", in addition to retrieving the records for "`Bill`", we also identify those of "`William`" following the link.

3.4 Supporting Type-Ahead Search on Relational Databases

In [10] we studied how to support type-ahead search on a relational database with multiple tables. We model the underlying data as a graph, and propose efficient index structures and algorithms for finding relevant answers on-the-fly by joining tuples in the database. This join operation is more challenging than the single-table case due to the complexity of the data model. We devised a partition-based method to improve query performance by grouping relevant tuples and pruning irrelevant tuples efficiently. We also developed a technique to answer a query efficiently by predicting highly relevant complete queries for the user. The idea is to first predict the query the user may want to type, then use it to compute answers.

3.5 Open Problems

Reducing Memory Requirements: There have been studies on disk-based index structures and algorithms for type-ahead search [4, 1, 3, 2]. To achieve a very high interactive speed, especially for Web servers with a lot of queries, ideally we want to store in memory the data, index structures, and cached query results. This memory requirement will increase as the data set increases. Furthermore, it is expensive to answer queries with short prefixes, since these prefixes can have many complete keywords. To make such queries more efficient, we can also cache the results for short-keyword queries, which also require a lot of memory. Thus new techniques are needed to solve this problem of high memory requirement.

One way to solve the problem is utilize solid-state drives (SSDs or flash drives). SSDs serve as a storage layer between traditional hard disks and memory. They are much faster but more expensive than traditional hard disks, yet cheaper and slower than memory. Compared to hard disks, an SSD has several advantages such as smaller startup time, extremely low read latency times, and relatively deterministic read performance. Due to these advantages, there have been many recent studies on how to use this new storage medium to do efficient data management. It is worth studying how to use SSDs to support efficient type-ahead search.

Type-ahead Search in non-English Languages: Considering the global reach of the Internet and its internationalization trend, it is important to study how to extend type-ahead-search techniques mainly designed for English to other languages, and solve unique, common challenges specific to these languages. For instance, there are many languages using the Latin alphabet such as French, Greek, German, and Turkish. They are similar to English in the way words are separated by delimiters such as space. Therefore, existing techniques for search on queries with multiple keywords can be naturally applied to these languages. At the same time, we need to consider their language-specific characteristics. For example, these languages often have special characters that do not exist in English, and these letters often have corresponding English letters. Many special letters are obtained by adding a diacritical mark to an English letter, such as the letter `ä` in German corresponding to the English letter `a`. In most cases these special characters are used to change the accent of their corresponding base letters. On an English keyboard, users often substitute these special characters with their base letters.

We use Turkish as an example to show language-specific characteristics and the corresponding search-related issues. Turkish has six special characters: `ç`, `ğ`, `ı`, `ö`, `ş`, and `ü`, which correspond to characters `c`, `g`, `i`, `o`, `s`, and `u`, respectively. When typing in a special letter, users often type in the base letter since it is easier. This common input behavior affects the way a search should be done. In a search engine such as Google, if a user types in a keyword `koruk`, the search engine will return both `koruk` pages and `körük` pages, since the engine may not know if the user typed the word exactly or substituted special characters with base letters. On the other hand, if the user types in the word `körük` with special characters, the system may still find documents containing the word `koruk`, but possibly give them a lower rank compared to those documents with the word `körük`. The reason is that the user may specifically want to use the special letters to find documents with this word. As a result, the similarity between these two words becomes asymmetric. Similar issues need to considered in type-ahead search in these languages.

Go Beyond String Data: Consider the case where we have a publication database, and a user types in a keyword “`2001`” in type-ahead search. Based on edit distance, we may find a record with a keyword “`2009`”. On the other hand, if we knew this keyword is about the year of a publication, then we may relax the condition in a different way, e.g., by finding publications in a year between 1999 and 2003. This kind of relaxation and matching depends on the type and semantics of the corresponding attribute, and requires new techniques to do indexing and searching.

Supporting Large Scale Data: For large amounts of data that cannot be handled by a single machine, we need to use multiple machines to support type-ahead search. For structured data, such as relational records, we can partition them to different machines, and let each machine process its local data. A keyword query is sent to these machines, which answer the query locally and send the results to a master machine to do the aggregation. New techniques are needed to support type-ahead search, especially related to how to partition complex data models such as graphs.

4 Conclusion

In this paper, we gave an overview of the new information-access paradigm, called search-as-you-type or type-ahead search, which finds answers to queries as a user types in keywords character by character. We discussed technical challenges related to achieving a high interactive search speed, reported recently developed results, and presented open problems that need more research investigation.

5 Acknowledgment

The work was partially supported by the National Science Foundation of the US under the awards IIS-0742960 and IIS-0844574, gift funds from Google and Microsoft, the National Science Foundation of China under the grant numbers 60828004 and 60873065, the National High Technology Development 863 Program of China under grant numbers 2007AA01Z152 and 2009AA011906, and the National Grand Fundamental Research 973 Program of China under the grant number 2006CB303103.

References

[1] Holger Bast, Alexandru Chitea, Fabian M. Suchanek and Ingmar Weber. ESTER: efficient search on text, entities, and relations. SIGIR, pages 671-678, 2007.

[2] Holger Bast Christian Worm Mortensen and Ingmar Weber. Output-Sensitive Autocompletion Search. SPIRE, 2006.

[3] Holger Bast and Ingmar Weber. Type less, find more: fast autocompletion search with a succinct index. SIGIR, pages 364-371, 2006.

[4] Holger Bast and Ingmar Weber. The CompleteSearch Engine: Interactive, Efficient, and Towards IR& DB Integration. CIDR, pages 88-95, 2007.

[5] Surajit Chaudhuri and Raghav Kaushik. Extending Autocompletion to Tolerate Errors. SIGMOD, pages 707-718, 2009.

[6] Ronald Fagin. Combining Fuzzy Information from Multiple Systems. PODS, pages 216-226, 1996.

[7] Shengyue Ji, Guoliang Li, Chen Li and Jianhua Feng. Interative Fuzzy Keyword Search. WWW, pages 142-153, 2009.

[8] Chen Li, Jiaheng Lu and Yiming Lu. Efficient Merging and Filtering Algorithms for Approximate String Searches. ICDE, pages 257-266, 2008.

[9] Guoliang Li, Jianhua Feng and Lizhu Zhou. Interactive Search in XML Data. WWW, pages 1063-1064, 2009.

[10] Guoliang Li, Shengyue Ji, Chen Li and Jianhua Feng. Efficient Type-Ahead Search on Relational Data: a TASTIER Approach. SIGMOD, pages 695-706, 2009.

[11] Jiannan Wang, Guoliang Li and Jianhua Feng. Automatic URL completion and prediction using fuzzy type-ahead search. SIGIR, pages 634-635, 2009.