Efficient Interactive Fuzzy Keyword Search

Shengyue Ji
University of California, Irvine
Irvine, California 92697, USA
shengyuj@ics.uci.edu

Guoliang Li
Tsinghua University
Beijing 100084, China
liguoliang@tsinghua.edu.cn

Chen Li
University of California, Irvine
Irvine, California 92697, USA
chenli@ics.uci.edu

Jianhua Feng
Tsinghua University
Beijing 100084, China
fengjh@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 autocomplete in these systems is a first step towards solving this problem. In this paper, we study a new information-access paradigm, called "interactive, fuzzy search," in which the system searches the underlying data "on the fly" as the user types in query keywords. It extends autocomplete interfaces by (1) allowing keywords to appear in multiple attributes (in an arbitrary order) of the underlying data; and (2) finding relevant records that have keywords matching query keywords *approximately*. This framework allows users to explore data as they type, even in the presence of minor errors. We study research challenges in this framework for large amounts of data. Since each keystroke of the user could invoke a query on the backend, we need efficient algorithms to process each query within milliseconds. We develop various incremental-search algorithms using previously computed and cached results in order to achieve an interactive speed. We have deployed several real prototypes using these techniques. One of them has been deployed to support interactive search on the UC Irvine people directory, which has been used regularly and well received by users due to its friendly interface and high efficiency.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*query formulation, search process*

General Terms

Algorithms, Experimentation, Performance

Keywords

Interactive Search, Fuzzy Search, Autocomplete

1. INTRODUCTION

In a traditional information system, a user composes a query, submits it to the system, which retrieves relevant answers. This information-access paradigm requires the user to have certain knowledge about the structure and content of the underlying data repository. In the case where the user has limited knowledge about the data, often the user feels "left in the dark" when issuing queries, and has to use a try-and-see approach for finding information, as illustrated by the following example.

At a conference venue, an attendee named John met a person from a university. After the conference he wanted to get more information about this person, such as his research projects. All John knows about the person is that he is a professor from that university, and he only remembers the name roughly. In order to search for this person, John goes to the directory page of the university. Figure 1 shows such an interface. John needs to fill in the form by providing information for multiple attributes, such as name, phone, department, and title. Given his limited information about the person, especially since he does not know the exact spelling of the person's name, John needs to try a few possible keywords, go through the returned results, modify the keywords, and reissue a new query. He needs to repeat this step multiple times to find the person, if lucky enough. This search interface is neither efficient nor user friendly.

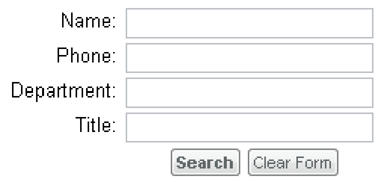
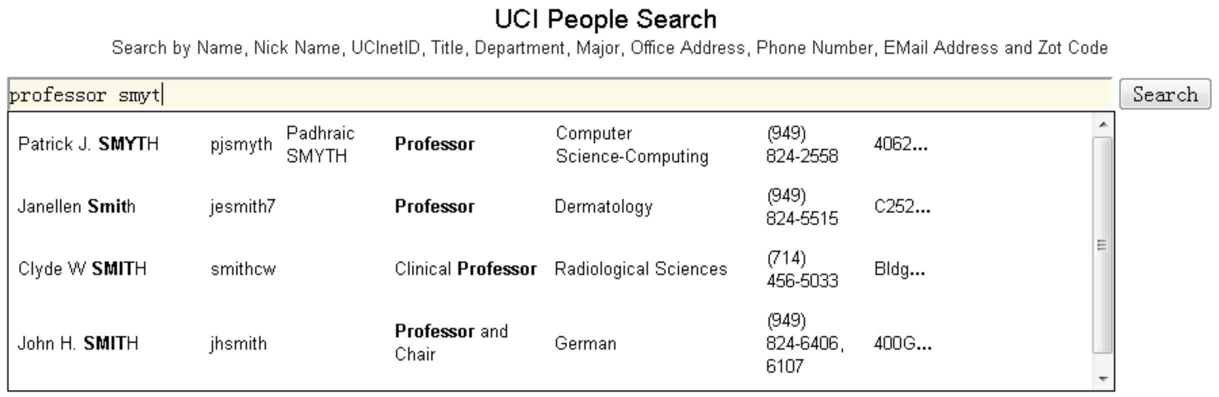


Figure 1: A typical directory-search form.

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 based on the partial query the user has entered. As an example, almost all the major search engines nowadays automatically suggest possible keyword queries as a user types in partial keywords. Both Google Finance and Yahoo! Finance support searching for stock information *interactively* as users type in keywords. More and more Web sites are having this feature, due to recent advances in high-speed networks and browser-based programming languages and tools such as JavaScript and AJAX.

In this paper, we study a new computing paradigm, called "interactive, fuzzy search." It has two unique features: (1)



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: Interactive fuzzy search on the UC Irvine people directory (http://psearch.ics.uci.edu).

Interactive: The system searches for the best answers "on the fly" as the user types in a keyword query; (2) *Fuzzy*: When searching for relevant records, the system also tries to find those records that include words *similar* to the keywords in the query, even if they do not match exactly.

We have developed several prototypes using this paradigm. The first one 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 can also find records with words that are similar to the query keywords, such as a person name "`smith`". The feature of supporting fuzzy search is especially important when the user has limited knowledge about the underlying data or the entities he or she is looking for. As the user types in more letters, the system interactively searches on the data, and updates the list of relevant records. 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. Another prototype, available at http://dblp.ics.uci.edu, supports search on a DBLP dataset (http//www.informatik.uni-trier.de/∼ley/db/) with about 1 million publication records. A third prototype, available at http://pubmed.ics.uci.edu, supports search on 3.95 million MEDLINE records (http://www.ncbi.nlm.nih.gov/pubmed).

In this paper we study research challenges that arise naturally in this computing paradigm. The main challenge is the requirement of a high efficiency. To make search really interactive, for each keystroke on the client browser, from the time the user presses the key to the time the results computed from the server are displayed on the browser, the delay should be as small as possible. An interactive speed requires this delay be within milliseconds. Notice that this time includes the network transfer delay, execution time on the server, and the time for the browser to execute its javascript (which tends to be slow). Providing a high efficiency on a large amount of data is especially challenging because of two reasons. First, we allow the query keywords to appear in different attributes with an arbitrary order, and the "on-the-fly join" nature of the problem can be computationally expensive. Second, we want to support fuzzy search by finding records with keywords that match query keywords approximately.

We develop novel solutions to these problems. We present several incremental-search algorithms for answering a query by using cached results of earlier queries. In this way, the computation of the answers to a query can spread across multiple keystrokes of the user, thus we can achieve a high speed. Specifically, we make the following contributions. (1) We first study the case of queries with a single keyword, and present an incremental algorithm for computing keyword prefixes similar to a prefix keyword (Section 3). (2) For queries with multiple keywords, we study various techniques for computing the intersection of the inverted lists of query keywords, and develop a novel algorithm for computing the results efficiently (Section 4.1). Its main idea is to use forward lists of keyword IDs for checking whether a record matches query keyword conditions (even approximately). (3) We develop a novel on-demand caching technique for incremental search. Its 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. In this way, we can efficiently compute and cache a small amount of results (Section 4.2). (4) We study various features in this paradigm, such as how to rank results properly, how to highlight keywords in the results, and how to utilize domain-specific information such as synonyms to improve search (Section 5). (5) In addition to deploying several real prototypes, we conducted a thorough experimental evaluation of the developed techniques on real data sets, and show the practicality of this new computing paradigm. All experiments were done using a single desktop machine, which can still achieve response times of milliseconds on millions of records.

1.1 Related Work

Prediction and Autocomplete:[1] There have been many studies on predicting queries and user actions [17, 14, 9, 19, 18]. With these techniques, a system predicts a word or

[1]The word "autocomplete" could have different meanings. Here we use it to refer to the case where a query (possibly with multiple keywords) is treated as a *single* prefix.

a phrase the user may type in next based on the sequence of partial input the user has already typed. Many prediction and autocomplete systems treat a query with multiple keywords as a single string, thus they do not allow these keywords to appear at different places. For instance, consider the search box on the home page of Apple.com, which allows autocomplete search on Apple products. Although a keyword query "`itunes`" can find a record "`itunes wi-fi music store`," a query with keywords "`itunes music`" cannot find this record (as of November 2008), simply because these two keywords appear at different places in the record. The techniques presented in this paper focus on "search on the fly," and they allow query keywords to appear at different places. As a consequence, we cannot answer a query by simply traversing a trie index. Instead, the backend intersection (or "join") operation of multiple lists requires more efficient indexes and algorithms.

CompleteSearch: Bast et al. proposed techniques to support "CompleteSearch," in which a user types in keywords letter by letter, and the system finds records that include these keywords (possibly at different places) [4, 5, 2, 3]. Our work differs from theirs as follows. (1) CompleteSearch mainly focused on compression of index structures, especially in disk-based settings. Our work focuses on efficient query processing using in-memory indexes in order to achieve a high interactive speed. (2) Our work allows fuzzy search, making the computation more challenging. (3) For a query with multiple keywords, CompleteSearch mainly caches the results of the query excluding the last keyword, which may require computing and caching a large amount of intermediate results. Our incremental caching technique described in Section 4.2 achieves a higher efficiency.

Gram-Based Fuzzy Search: There have been recent studies to support efficient fuzzy string search using grams [7, 1, 8, 10, 15, 16, 13, 12, 11, 20, 6]. A gram of a string is a substring that can be used as a signature for efficient search. These algorithms answer a fuzzy query on a collection of strings using the following observation: if a string r in the collection is similar to the query string, then r should share a certain number of common grams with the query string. This "count filter" can be used to construct gram inverted lists for string ids to support efficient search. In Section 6 we evaluated some of the representative algorithms. The results showed that, not surprisingly, they are not as efficient as trie-based incremental-search algorithms, mainly because it is not easy to do incremental computation on gram lists, especially when a user types in a relatively short prefix, and count filtering does not give enough pruning power to eliminate false positives.

2. PRELIMINARIES

Problem Formulation: We formalize the problem of interactive, fuzzy search on a relational table, and our method can be adapted to textual documents, XML documents, and relational databases. Consider a relational table T with m attributes and n records. Let $A = \{a_1, a_2, \ldots, a_m\}$ denote the attribute set, $R = \{r_1, r_2, \ldots, r_n\}$ denote the record set, and $W = \{w_1, w_2, \ldots, w_p\}$ denote the distinct-word set in T. Given two words w_i and w_j, "$w_i \preceq w_j$" denotes that w_i is a prefix string of w_j.

A query consists of a set of prefixes $Q = \{p_1, p_2, \ldots, p_l\}$. For each prefix p_i, we want to find the set of prefixes from the data set that are similar to p_i.[2] In this work we use edit distance to measure the similarity between two strings. The *edit distance* between two strings s_1 and s_2, denoted by $\texttt{ed}(s_1, s_2)$, 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, $\texttt{ed}(\texttt{smith}, \texttt{smyth}) = 1$.

DEFINITION 1 (INTERACTIVE FUZZY SEARCH). *Given a set of records R, let W be the set of words in R. Consider a query $Q = \{p_1, p_2, \ldots, p_\ell\}$ and an edit-distance threshold δ. For each p_i, let P_i be $\{p'_i | \exists w \in W, p'_i \preceq w$ and $\texttt{ed}(p'_i, p_i) \leq \delta\}$. Let the set of candidate records R_Q be $\{r | r \in R, \forall 1 \leq i \leq \ell, \exists p'_i \in P_i$ and w_i appears in r, $p'_i \preceq w_i\}$. The problem is to compute the best records in R_Q ranked by their relevancy to Q. These records are computed incrementally as the user modifies the query, e.g., by typing in more letters.*

Indexing: We use a trie to index the words in the relational table. Each word w in the table 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. For simplicity, a node is mentioned interchangeably with its corresponding string in the remainder of the paper. Each leaf node 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. For instance, Figure 3 shows a partial index structure for publication records. The word "`vldb`" has a trie node ID of 15, and its inverted list includes record IDs 6, 7, and 8. For simplicity, the figure only shows the record ids, without showing the additional information about attributes and positions.

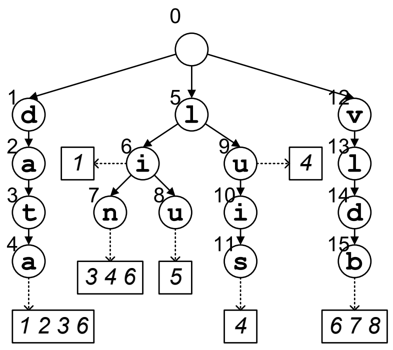


Figure 3: Trie with inverted lists at leaf nodes.

3. SINGLE KEYWORD

In this section we study interactive, fuzzy search on a single keyword prefix. For the case of exact prefix search using the trie index, the approach is straightforward: for each prefix, there exists only one corresponding trie node (if any). The inverted lists of its descendant leaf nodes are accessed to retrieve candidate records.

The solution to the problem of fuzzy search is trickier since one prefix can have multiple similar prefixes (called "active

[2]Clearly our techniques can be used to answer queries when only the last keyword is treated as a partial prefix, and the other are treated as completed keywords.

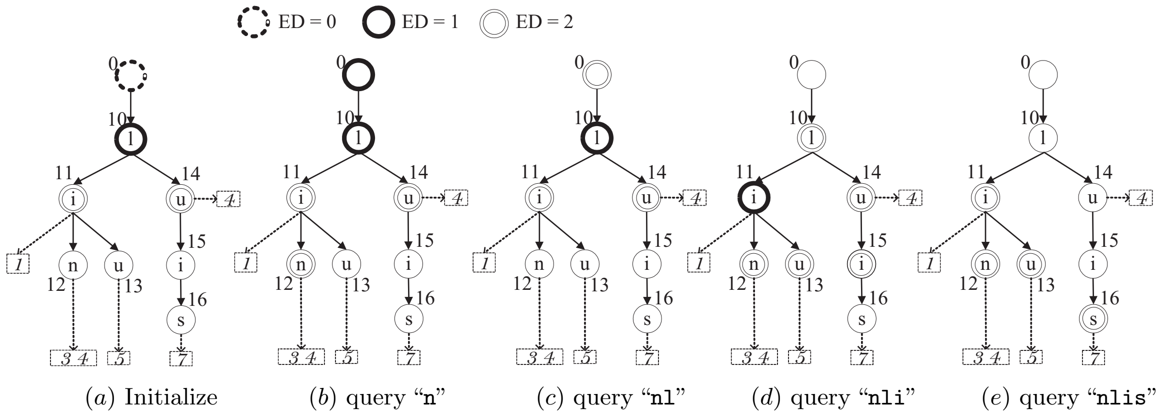


Figure 4: Fuzzy search of prefix queries of "`nlis`" (threshold $\delta = 2$).

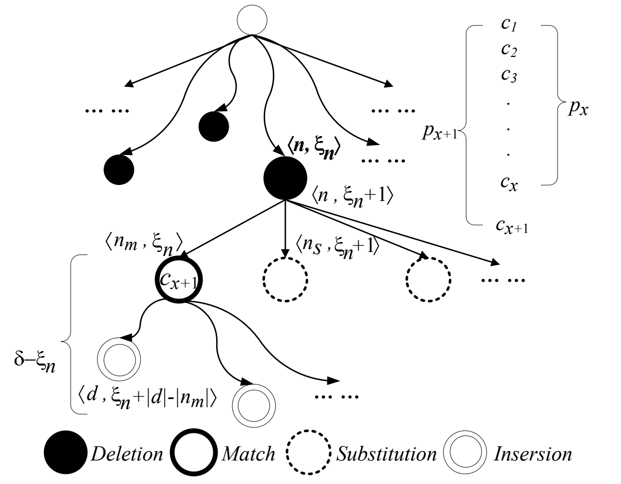
nodes"), and we need to compute them efficiently. The leaf descendants of the active nodes are called the *predicted keywords* of the prefix. For example, consider the trie in Figure 4. Suppose the edit-distance threshold $\delta = 2$, and a user types in a prefix p = "`nlis`". Prefixes "`li`", "`lin`", "`liu`", and "`luis`" are all similar to p, since their edit distances to "`nlis`" are within δ. Thus nodes 11, 12, 13, and 16 are active nodes for p (Figure 4 (e)). The predicted keywords for the prefix are "`li`", "`lin`", "`liu`", and "`luis`".

We develop a caching-based algorithm for incrementally computing active nodes for a keyword as the user types it in letter by letter. Given an input prefix p, we compute and store the set of active nodes $\Phi_p = \{\langle n, \xi_n\rangle\}$, in which n is an active node, and $\xi_n = \mathsf{ed}(p, n) \le \delta$. The idea behind our algorithm is to use prefix-filtering: when the user types in one more letter after p, the active nodes of p can be used to compute the active nodes of the new query.

The algorithm works as follows. First, an active-node set is initialized for the empty query ϵ, i.e., $\Phi_\epsilon = \{\langle n, \xi_n\rangle \mid \xi_n = |n| \le \delta\}$. That is, it includes all trie nodes n whose corresponding string has a length $|n|$ within the edit-distance threshold δ. These nodes are active nodes for ϵ since their edit distances to ϵ are within δ.

As the user types in a query string $p_x = c_1c_2 \ldots c_x$ letter by letter, the active-node set Φ_{p_x} is computed and cached for p_x. When the user types in a new character c_{x+1} and submits a new query p_{x+1}, the server computes the active-node set $\Phi_{p_{x+1}}$ for p_{x+1} by using Φ_{p_x}. For each $\langle n, \xi_n\rangle$ in Φ_{p_x}, the descendants of n are examined as active-node candidates for p_{x+1}, as illustrated in Figure 5. For the node n, if $\xi_n + 1 \le \delta$, then n is an active node for p_{x+1}, and $\langle n, \xi_n + 1\rangle$ is added into $\Phi_{p_{x+1}}$. This case corresponds to deleting the last character c_{x+1} from the new query string p_{x+1}. Notice even if $\xi_n + 1 \le \delta$ is not true, node n could still potentially become an active node of the new string, due to operations described below on other active nodes in Φ_{p_x}.

For each child n_c of node n, there are two possible cases. **Case 1**: the child node n_c has a character different from c_{x+1}. Figure 5 shows a node n_s for such a child node, where "s" stands for "substitution," the meaning of which will become clear shortly. We have $\mathsf{ed}(n_s, p_{x+1}) \le \mathsf{ed}(n, p_x) + 1 = \xi_n + 1$. If $\xi_n + 1 \le \delta$, then n_s is an active node for the new string, and $\langle n_s, \xi_n + 1\rangle$ is added into $\Phi_{p_{x+1}}$. This case corresponds to substituting the label of n_s for the letter c_{x+1}. **Case 2**: the child node n_c has a label c_{x+1}. Figure 5 shows the node n_m for such a child node, where "m" stands for "matching." In this case, $\mathsf{ed}(n_m, p_{x+1}) \le \mathsf{ed}(n, p_x) = \xi_n \le \delta$. Thus, n_m is an active node of the new string, so we add $\langle n_m, \xi_n\rangle$ into $\Phi_{p_{x+1}}$. This case corresponds to the match between the character c_{x+1} and the label of n_m. One subtlety here is that, if the distance for the node n_m is smaller than δ, i.e., $\xi_n < \delta$, the following operation is also required: for each n_m's descendant d that is at most $\delta - \xi_n$ letters away from n_m, we need to add $\langle d, \xi_d\rangle$ to the active-node set for the new string p_{x+1}, where $\xi_d = \xi_n + |d| - |n_m|$. This operation corresponds to inserting several letters after node n_m.[3]



c_1
c_2
c_3
c_x
c_{x+1}
p_x
p_{x+1}
$\langle n, \xi_n\rangle$
$\langle n, \xi_n{+}1\rangle$
$\langle n_m, \xi_n\rangle$
$\langle n_s, \xi_n{+}1\rangle$
c_{x+1}
$\delta{-}\xi_n$
$\langle d, \xi_n{+}|d|{-}|n_m|\rangle$
Deletion
Match
Substitution
Insersion

Figure 5: Incrementally computing the active-node set $\Phi_{p_{x+1}}$ from the active-node set Φ_{p_x}. We consider an active node $\langle n, \xi_n\rangle$ in Φ_{p_x}.

During the computation of set $\Phi_{p_{x+1}}$, it is possible to add multiple pairs $\langle v, \xi_1\rangle$, $\langle v, \xi_2\rangle$, ... for the same trie node v. In this case, only the one with the smallest edit operation is kept in $\Phi_{p_{x+1}}$. The reason is that, by definition, for the same trie node v, only the pair with the edit distance between the node v and the query string p_{x+1} should be kept in $\Phi_{p_{x+1}}$, which means the *minimum* number of edit operations to transform the string of v to the string of p_{x+1}.

[3]Descendants of node n_s do not need to be considered for insertions, because if these descendants are active nodes of $\Phi_{p_{x+1}}$, their parents must appear in Φ_{p_x}, and they can be processed by substitutions on their parents.

The following lemma shows the correctness of this algorithm.

LEMMA 1. *For a query string $p_x = c_1c_2 \ldots c_x$, let Φ_{p_x} be its active-node set. Consider a new query string $p_{x+1} = c_1c_2 \ldots c_xc_{x+1}$. (1) Soundness: Each node computed by the algorithm described above is an active node of the new query string p_{x+1}. (2) Completeness: Every active node of the new query string p_{x+1} will be computed by the algorithm above.*

For example, assume a user types in a query "`nlis`" letter by letter, and the threshold δ is 2. Figure 4 illustrates how the algorithm processes the prefix queries invoked by keystrokes. Table 1 shows the details of how to compute the active-node sets incrementally. The first step is to initialize $\Phi_\epsilon = \{\langle 0, 0, \langle 10, 1\rangle, \langle 11, 2\rangle, \langle 14, 2\rangle\}$ (Figure 4(a) and Table 1(a)). When the user types in the first character "`n`", for the string s = "`n`", its active-node set Φ_s is computed based on Φ_ϵ as follows. For $\langle 0, 0\rangle \in \Phi_\epsilon$, we add $\langle 0, 1\rangle$ into Φ_s, since letter "`n`" can be deleted. For node 10, a child of node 0 with a letter "`l`", $\langle 10, 1\rangle$ is added into Φ_s, as "`l`" can be substituted for "`n`". There are no match and insertion operations as node 1 does not have a child with label "`n`". Φ_s is computed in this way (Figure 4 (b) and Table 1 (b)). Similarly, prefix queries of "`nlis`" can be answered incrementally.

For each active node, the keywords corresponding to its leaf descendants are predicted keywords. Consider the active-node set for the prefix query "`nl`" as shown in Figure 4(c). For $\langle 11, 2\rangle$, "`li`", "`lin`", and "`liu`" are predicted keywords accordingly. We retrieve the records on the inverted lists of predicted words to compute answers to the query.

Table 1: Active-node sets for processing prefix queries of "`nlis`" (edit distance threshold $\delta = 2$)

(a) Query "`n`"

Φ_ϵ	⟨0,0⟩	⟨10,1⟩	⟨11,2⟩	⟨14,2⟩
Delete	⟨0,1⟩	⟨10,2⟩	–	–
Substitute	⟨10,1⟩	⟨11,2⟩;⟨14,2⟩	–	–
Match	–	–	⟨12,2⟩	–
Insert	–	–	–	–
Φ_n	⟨0,1⟩; ⟨10,1⟩; ⟨11,2⟩; ⟨12,2⟩; ⟨14,2⟩			

(b) Query "`nl`"

Φ_n	⟨0,1⟩	⟨10,1⟩	⟨11,2⟩	⟨12,2⟩	⟨14,2⟩
Delete	⟨0,2⟩	⟨10,2⟩	–	–	–
Substitute	–	⟨11,2⟩;⟨14,2⟩	–	–	–
Match	⟨10,1⟩	–	–	–	–
Insert	⟨11,2⟩;⟨14,2⟩	–	–	–	–
Φ_{nl}	⟨10,1⟩; ⟨0,2⟩; ⟨11,2⟩; ⟨14,2⟩				

(c) Query "`nli`"

Φ_{nl}	⟨10,1⟩	⟨0,2⟩	⟨11,2⟩	⟨14,2⟩
Delete	⟨10,2⟩	–	–	–
Substitute	⟨14,2⟩	–	–	–
Match	⟨11,1⟩	–	–	⟨15,2⟩
Insert	⟨12,2⟩;⟨13,2⟩	–	–	–
Φ_{nli}	⟨11,1⟩; ⟨10,2⟩; ⟨12,2⟩; ⟨13,2⟩; ⟨14,2⟩; ⟨15,2⟩			

(d) Query "`nlis`"

Φ_{nli}	⟨11,1⟩	⟨10,2⟩	⟨12,2⟩	⟨13,2⟩	⟨14,2⟩	⟨15,2⟩
Delete	⟨11,2⟩	–	–	–	–	–
Substitute	⟨12,2⟩;⟨13,2⟩	–	–	–	–	–
Match	–	–	–	–	–	⟨16,2⟩
Insert	–	–	–	–	–	–
Φ_{nlis}	⟨11,2⟩; ⟨12,2⟩; ⟨13,2⟩; ⟨16,2⟩					

4. MULTIPLE KEYWORDS

In this section we study answering interactive fuzzy search when a user types in multiple keywords. The goal is to efficiently and incrementally compute the records with keywords whose prefixes are similar to those query keywords. We focus on several challenges in this setting. (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 predicted 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. In Section 4.1 we study various algorithms for computing the answers efficiently. (2) *Cache-based incremental intersection*: In most cases, the user types the query letter by letter, and subsequent queries append additional letters to previous ones. Based on this observation, we study how to use the cached results of earlier queries to answer a query incrementally (Section 4.2).

4.1 Intersecting Union Lists of Prefixes

For simplicity, we first consider exact search, and then extend the results to fuzzy search. Given a query $Q = \{p_1, p_2, \ldots, p_\ell\}$, suppose $\{k_{i_1}, k_{i_2}, \ldots\}$ is the set of keywords that share the prefix p_i. Let L_{i_j} denote the inverted list of k_{i_j}, and $\mathcal{U}_i = \bigcup_j L_{i_j}$ be the union of the lists for p_i. We study how to compute the answer to the query, i.e., $\bigcap_i \mathcal{U}_i$.

Simple Methods: One method is the following. For each prefix p_i, we compute the corresponding union list $\mathcal{U}_i$ on-the-fly and intersect the union lists of different keywords. The time complexity for computing the unions could be $O(\sum_{i,j} |L_{i_j}|)$. The shorter the keyword prefix is, the slower the query could be, as inverted lists of more predicted keywords need to be traversed to generate the union list. This approach only requires the inverted lists of trie leaf nodes, and the space complexity of the inverted lists is $O(n \times L)$, where n is the number of records and L is the average number of distinct keywords of each record.

Alternatively, we can pre-compute and store the union list of each prefix, and intersect the union lists of query keywords when a query comes. The main issue of this approach is that the precomputed union lists require a large amount of space, especially since each record occurrence on an inverted list needs to be stored many times. The space complexity of all the union lists is $O(n \times L \times w)$, where w is the average keyword length. Compression techniques can be used to reduce the space requirement.

There have been other approaches for answering prefix intersection. For instance, Bast et al. [4] proposed a method that groups ranges of keywords and builds document lists separately for each range. Intersection is performed between an existing document list and several ranges called "HYB blocks." The limitation of this approach is that, for most queries, the ranges can include many irrelevant documents, which require a lot of time to process. We will show our experimental results of this method in Section 6.

Efficient Prefix Intersection Using Forward Lists: We develop a new solution based on the following ideas. Among the union lists $\mathcal{U}_1, \mathcal{U}_2, \ldots, \mathcal{U}_\ell$, we identify the shortest union list. Each record ID on the shortest list is verified by checking if it exists on all the other union lists (following the ascending order of their lengths). Notice that these union lists are not materialized in the computation. The shortest union list can be traversed by accessing the leaf nodes of the corresponding prefix. The length of each union list can be pre-computed and stored in the trie, or estimated on-the-fly. To verify record occurrences efficiently, a forward list can be maintained for each record r, which is a sorted list of IDs of keywords in r, denoted as F_r. A unique property of the keyword IDs is that they are encoded using their alphabetical order. Therefore, each prefix p_i has a range of keyword IDs $[MinId_i, MaxId_i]$, so that if p_i is a prefix of another string s, then the ID of s should be within this range.

An interesting observation is, for a record r on the shortest union list, the problem of verifying whether r appears on (non-materialized) union list $\mathcal{U}_k$ of a query prefix p_k, is equivalent to testing if p_k appears in the forward list F_r as a prefix. We can do a binary search for $MinId_k$ on the forward list F_r to get a lower bound Id_{lb}, and check if Id_{lb} is no larger than $MaxId_k$. The probing succeeds if the condition holds, and fails otherwise. The time complexity for processing each single record r is $O\big((\ell - 1) logL\big)$, where ℓ is the number of keywords in the query, and L is the average number of distinct keywords in each record. A good property of this approach is that the time complexity of each probing does not depend on the lengths of inverted lists, but on the number of unique keywords in a record (logarithmically).

Figure 6 shows an example when a user types in a query "`vldb li`". The predicted keywords for "`li`" are "`li`", "`lin`", "`line`", and "`linux`". The keyword-ID range of each query keyword is shown in brackets. For instance, the keyword-ID range for prefix "`li`" is [3, 5], which covers the ranges of "`lin`" and "`liu`". To intersect the union list of "`vldb`" with that of "`li`", we first identify "`vldb`" as the one with the shorter union list. The record IDs (6, 7, 8, ...) on the list are probed one by one. Take record 6 as an example. Its forward list contains keyword IDs 2, 4, 8, ... We use the range of "`li`" to probe the forward list. By doing a binary search for the keyword ID 3, we find keyword with ID 4 on the forward list, which is then verified to be no larger than MaxID = 6. Therefore, record 6 is an answer to the query, and the keyword with ID 4 (which appears in record 6) has "`li`" as a prefix.

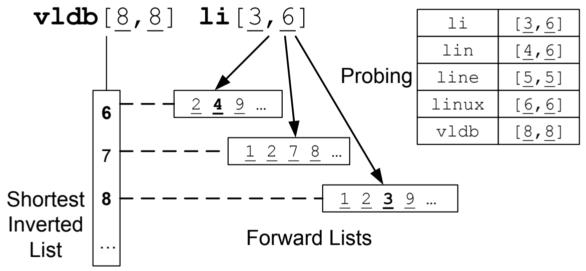


Figure 6: Prefix intersection using forward lists. (Numbers with underlines are keyword IDs, and numbers without underlines are record IDs.)

Extension to Fuzzy Search: The algorithm described above naturally extends to the case of fuzzy search. Since each query keyword has multiple active nodes of similar prefixes, instead of considering the union of the leaf nodes of one prefix node, now we need to consider the unions of the leaf nodes for all active nodes of a prefix keyword. The lengths of these union lists can be estimated in order to find a shortest one. For each record r on the shortest union list, for each of the other query keywords, for each of its active nodes, we test if the corresponding similar prefix can appear in the record r as a prefix using the forward list of r.

4.2 Cache-Based Intersection of Prefixes

In Section 3 we presented an algorithm for incrementally computing similar prefixes for a query keyword, as the user types the keyword letter by letter. Now we show that prefix intersection can also be performed incrementally using previously cached results.

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. As in this example, in the following discussion, we use "Q_1" to refer to a query whose results have been cached, and "Q_2" to refer to a new query whose results we want to compute using those of Q_1.

Cache Miss: Often the more keywords the user types in, the more typos and mismatches the query could have. Thus we may want to dynamically increase the edit-distance threshold δ as the query string is getting longer. Then it is possible that the threshold for the new query Q_2 is strictly larger than that of the original query Q_1. In this case, the active nodes of keywords in Q_1 might not include all those of keywords in Q_2. As a consequence, we cannot use the cached results of Q_1 (active nodes and answers) to compute those of Q_2. This case is a cache miss, and we need to compute the answers of Q_2 from scratch.

Reducing Cached Results: The cached results of query Q_1 could be large, which could require a large amount of time to compute and space to store. There are several cases where we can reduce the size. The first case is when we want to use pagination, i.e., we show the results in different pages. In this case, we can traverse the shortest list partially, until we have enough results for the first page. As the user browses the results by clicking "Previous" and "Next" links, we can continue traversing the shortest list to compute more results and cache them.

The second case is when the user is only interested in the best results, say, the top-k records according a ranking function, for a predefined constant k. Such a function could allow us to compute the answers to the query Q_1 without traversing the entire shortest list, assuming we are sure that all the remaining records on the list cannot be better than the results already computed. In other words, the ranking function allows us to do early termination during the traversal. When using the top-k results of Q_1 to compute the top-k results of Q_2, it is possible that the cached results are not enough, since Q_2 has a more restrictive keyword. In this case, we can continue the unfinished traversal on the shortest list, assuming we have remembered the place where the traversal stopped on the shortest list for query Q_1.

Figure 7 shows an example of incrementally computing top-k answers using cached results. A user types in a query "cs conf vanc" letter by letter, and the server receives queries "cs co", "cs conf", and "cs conf vanc" in order. (Notice that it is possible that some of the prefix queries were not received by the server due to the network overhead and the server delay.) The first query "cs co" is answered from scratch. Assuming the union list of keyword "cs" is the shorter one. The traversal stops at the first vertical bar. Each record accessed in the traversal is verified by probing the keyword range of "co" using the forward list of the record. Records that pass the verification are cached. When we want to answer the query "cs conf" incrementally, we first verify each record in the cached result of the previous query by probing the keyword range of "conf". Some of these results will become results of the new query. If the results from the cache is insufficient to compute the new top-k, we resume the traversal on the list of "cs", starting from the stopping point of the previous query, until we have enough top-k results for the new query. The next query "cs conf vanc" is answers similarly.

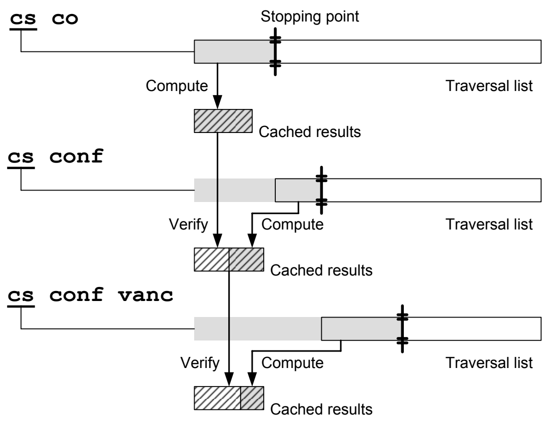


Figure 7: Computing top-k results using cached answers and resuming unfinished traversal on a list.

In the case of cache miss, i.e., earlier cached results cannot be used to compute the answers of a new query, we may need to answer the new query from scratch. We may choose a different list as the shortest one to traverse, and subsequent queries can be computed incrementally similarly.

5. ADDITIONAL FEATURES

5.1 Ranking

A ranking function considers various factors to compute an overall relevance score of a record to a query. The following are several 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 closer to the query keyword than "circumstance", therefore records containing the word "circle" could be ranked higher. In most cases exact matches on the query should have a higher priority 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 inverted 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 is $Q = \{p_1, p_2, \ldots\}$, 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 weights ($0 < \beta < \alpha < 1$), and $score(r, k_i)$ is a score of record r for keyword k_i.

5.2 Highlighting Best Prefixes

When displaying records to the user, the most similar prefixes for an input prefix should be highlighted. This highlighting 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", and there are several ways to highlight them, such as "**<u>lui</u>s**" or "**<u>luis</u>**". 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. 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.

5.3 Using Synonyms

We can 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 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 records for "Bill", we also identify those of "William" following the link. In this way, our techniques can be extended to utilize synonyms.

6. EXPERIMENTS

We deployed several prototypes in different domains to support interactive, fuzzy search. We conducted a thorough experimental evaluation of the developed techniques on real data sets, such as publications and people directories. Here we report the results on the following two data sets mainly because of their relative large sizes. (1) DBLP: It included about one million computer science publication records, with six attributes: authors, title, conference or journal name, year, page numbers, and URL. (2) MEDLINE: It had about 4 million latest publication records related to life sciences and biomedical information. We used five attributes: authors, their affiliations, article title, journal name, and journal issue. Table 2 shows the sizes of the data sets, index sizes, and index-construction times.

Table 2: Data sets and index costs

Data Set	DBLP	MEDLINE
Record Number	1 million	4 million
Original Data Size	190 MB	1.25 GB
# of Distinct Keywords	392K	1.79 million
Index-Construction Time	50 secs	588 secs
Trie Size	36 MB	165 MB
Inverted-List Size	52 MB	445 MB
Forward-List Size	54 MB	454 MB

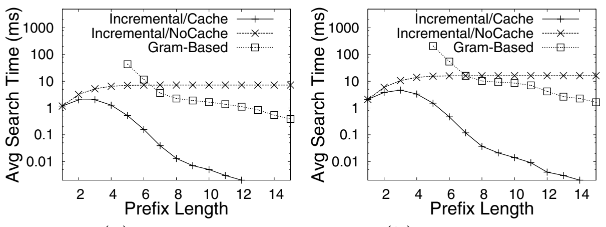
Two prototypes are available at http://dblp.ics.uci.edu/ and http://pubmed.ics.uci.edu/. For each of them, we set up a Web server using Apache2 on a Linux machine with an Intel Core 2 Quad processor Q6600 (2.40GHz, 8M, 1066MHz FSB) and 8G DDR2-800 memory. We implemented the backend as a FastCGI server process, which was written in C++, compiled with a GNU compiler. To make sure our experiments did not affect the deployed systems, we did the experiments on another Linux machine with an Intel Core 2 Duo processor E6600 (2.40GHz, 4M, 1066MHz FSB) and 3G DDR2-667 memory.

6.1 Efficiency of Computing Similar Prefixes

We evaluated the efficiency of computing the prefixes on the trie that are similar to a query keyword. For each data set, we generated 1,000 single-keyword queries by randomly selecting keywords in the data set, and applying a random number of edit changes (0 to 2) on the keyword. The average length (number of letters) of keywords was 9.9 for the DBLP data set, and 10.2 for the MEDLINE data set. For each prefix of each query, we measured the time to find similar prefixes within an edit distance of 2, not including the time to retrieve records. We computed the average time for the prefix queries with the same length.

We implemented three methods to compute similar prefixes. (1) Incremental/Cache: We computed the active nodes of a query using the cached active nodes of previous prefix queries, using the incremental algorithm presented in Section 3. This algorithm is applicable when the user types a query letter by letter. (2) Incremental/NoCache: We used the incremental algorithm, but assuming no earlier active nodes have been cached, and the computation started from scratch. This case happens when a user copies and pastes a long query, and none of the active nodes of any prefix queries has been computed. It also corresponds to the traditional non-interactive-search case, where a user submits a query and clicks the "Search" button. (3) Gram-Based: We built gram inverted lists on all prefixes with at least three letters using the method described in [15]. We used the implementation in the Flamingo release[4], using a gram length of 3 and a length filter. The total number of such prefixes was 1.1 millions for the DBLP data set and 5 millions for the MEDLINE data set. The index structure can be used to compute similar prefixes for keywords with at least four letters.

Figure 8 shows the performance results of these three methods. The method Incremental/Cache was most efficient. As the user types in letters, its response time first increased slightly (less than 5 ms), and then started decreasing quickly after the fourth letter. The main reason is that the number of active nodes decreased, and the cached results made the computation efficient. The method Incremental/NoCache required longer time since each query needed to be answered from scratch, without using any cached active nodes. The method Gram-Based performed efficiently when the query keyword had at least seven letters. But it had a very poor performance for shorter keywords, since the count filter had a weak power to prune false positives.



(a) DBLP (b) MEDLINE

Figure 8: Computing prefixes similar to a keyword.

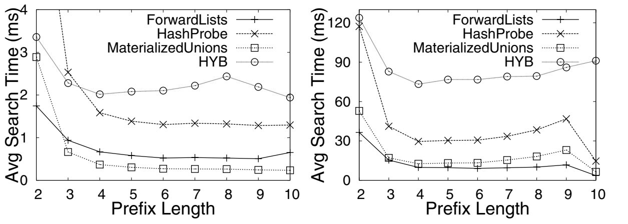
6.2 Efficiency of List Intersection of Multiple Keywords

We evaluated several methods for intersecting union lists of multiple keywords, as described in Section 4.1. For each data set, we generated 1,000 queries by randomly selecting records from the data set, and choosing keywords from each record. We implemented the following methods to intersect the union lists of the keywords. (1) ForwardLists: It is the algorithm presented in Section 4.1, which traverses the shortest union list and uses the other query keywords to probe the forward lists of records on the shortest list. The union list was traversed on the fly without being materialized. (2) HashProbe: The shortest union list was materialized as a hash table at query time. Each record ID on the other union lists was searched on the hash table. (3) MaterializedUnions: We materialized the union lists of all the query keywords and their prefixes, and computed an intersection by using the record IDs of the shortest list to probe the other union lists. We measured the intersection time only. (4) HYB: We implemented a structure called "HYB" as described in [4]. We used an in-memory implementation, and all IDs were stored without any encoding and compression, since decoding and decompression will introduce additional time overhead during query answering. The number of HYB blocks

[4]http://flamingo.ics.uci.edu/releases/2.0

was 162 for the DBLP dataset and 285 for the MEDLINE data set, using the parameters recommended in [4].

We evaluated these methods on queries with two keywords, assuming no previous query results were cached. Figure 9 shows the average time of each method as the length of the second keyword increased. The intersection operation was very time consuming when the second keyword had no more than two letters, since the union lists of the prefixes were long. The HashProbe method performed relatively poorly due to the cost of building the hash table for the shorter list and traversing the other list. The HYB method's performance was even worse for most cases. The main reason is that this method was designed to have a prefix overlapping with few HYB blocks (usually one or two) to avoid merging too many answer lists. However, it has a side effect that an HYB block could include too many keywords compared to the query prefix, forcing the method to process long lists. This drawback was not a main issue in a disk-based setting, as the algorithm can benefit from list compression and sequential disk IOs time. The MaterializedUnions method performed well, but with a high memory cost as discussed in Section 4.1. The ForwardLists algorithm achieved an excellent performance, at the cost of storing the forward lists. An interesting finding in the results is that ForwardLists even outperformed MaterializedUnions on the MEDLINE data set. This is because as the data set became larger, the average time of each binary search on the union lists increased, while the average time of each binary probe on the forward lists did not change much.



(a) DBLP (b) MEDLINE

Figure 9: List intersection of multiple keywords.

6.3 Performance of Cache-Based Intersection

We evaluated the performance of different methods of cache-based prefix intersection, as described in Section 4.2. We allowed at most one typo for each prefix with at most five letters, and two errors for prefixes with more than five letters. As a consequence, for a query with two keywords, when the sixth letter of the second keyword was typed in, a cache miss occurred. We implemented the following methods. (1) NoCache: No query results are cached. A query is computed without using any cached query results. Early termination is enabled. (2) CompleteTraversal: It traverses the shortest union list completely to compute the results of the current query. (3) PartialTraversal: It traverses the shortest union list partially until it finds the top 10 results for the current query (as discussed in Section 4.2).

Figure 10 shows the query time of the methods on the DBLP data set. CompleteTraversal outperformed the NoCache method for relatively long prefixes (with more than 6 letters) mainly due to the smaller set of cached results. The PartialTraversal method was the most efficient one in most cases, since it can stop early during the traversal of the list, and a new query can be incrementally computed using earlier results. All these methods required a relative long time when the prefix had 6 letters due to the cache miss.

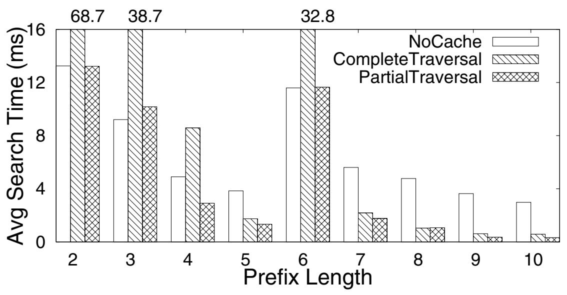
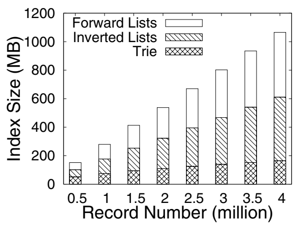


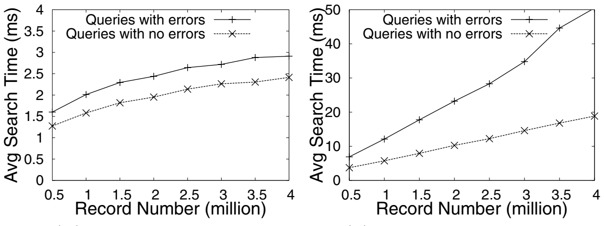
Figure 10: Performance of prefix intersection (DBLP).

6.4 Scalability

We evaluated the scalability of our algorithms. As an example, we used the MEDLINE dataset. Figure 11(a) shows how the index size increased as the number of records increased. It shows that all the sizes of trie, inverted lists, and forward lists increased linearly.



(a) Index Size



(b) Single Keyword (c) Multiple Keywords

Figure 11: Scalability (MEDLINE).

We measured the query performance as the data size increased. We first evaluated queries with a single keyword. We considered two types of queries: the first type was generated by randomly selecting keywords in the data set; the second type was obtained by modifying the first type by adding edit errors (0 to 2). Each query asked for the 10 best records. For each type, we measured the query response time for each keystroke. Figure 11(b) shows the results for the MEDLINE data set as we increased the number of records. It shows that the algorithms can answer a single-keyword query efficiently (within 3 ms), for both types of queries.

We next evaluated the algorithms for queries with multiple keywords, which asked for 10 best records. We generated queries with two keywords, and measured the average query time of each keystroke on the second keyword. Figure 11(c) shows that our algorithms can process such queries very efficiently. For instance, when the data set had 4 million records, a query without errors was processed within 20 ms, while a query with errors was answered within 55 ms.

6.5 Round-Trip Time

The round-trip time of interactive fuzzy search consists of three components: server processing time, network time, and JavaScript running time. Different locations in the world could have different network delays to our servers in southern California. We measured the time cost for a sample query "`divsh srivstava search`" on the DBLP prototype from five locations around the world: US west, US east, China, Israel, and Australia. Figure 12 shows the results. We can see that the server running time was less than 1/5 of the total round-trip time. JavaScript took around 40 to 60 ms to execute. The relative low speed at some locations was mainly due to the network delay. For example, it took about 4/5 of the total round-trip time when our system was tested from China. For all the users from different locations, the total round-trip time for a query was always below 300 ms, and all of them experienced an interactive interface. For large-scale systems processing queries from different countries, we can solve the possible network-delay problem by using distributed servers.

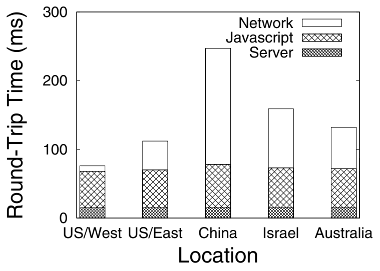


Figure 12: Round-trip time for different locations.

6.6 Saved Typing Effort

Interactive search can also save user typing efforts, since results can be found before the user types in complete keywords. To evaluate the saving of typing effort, we constructed six queries on the DBLP data set as shown in Table 3. The keywords in italic font are mistyped keywords. Each query was typed in letter by letter, until the system found the expected records. We measured how much letter-typing effort the system can save for the user. For each query Q_i, let $N(Q_i)$ be the number of letters the user typed before the relevant answers are found. We use $1 - \frac{N(Q_i)}{|Q_i|}$ to quantify the relative saved effort. For example, for query Q_6, the user could find relevant answers right after typing in "`divsh sri sea`". The saved effort of Q_6 is $1 - \frac{13}{22} = 41\%$, as the user only needed to type in 13 letters, instead of 22 letters in the full query. Table 3 shows that this paradigm can save the user 40% to 60% typing effort on average.

7. CONCLUSIONS

We studied a new information-access paradigm that supports interactive, fuzzy search. We proposed an efficient incremental algorithm to answer single-keyword fuzzy queries.

Table 3: Queries and saved typing effort.

ID	Query	Saved typing effort
Q_1	*sunta sarawgi*	42%
Q_2	surajit *chuardhuri*	50%
Q_3	*nick kodas approxmate*	41%
Q_4	*flostsos* icde similarity	38%
Q_5	*similarty* search icde	55%
Q_6	*divsh srivstava* search	41%

We studied various algorithms for computing the answers to a query with multiple keywords that are treated as fuzzy, prefix conditions. We developed efficient algorithms for incrementally computing answers to queries by using cached results of previous queries in order to achieve an interactive speed on large data sets. We studied several useful features such as ranking answers, highlighting results, and utilizing synonyms. We deployed several real systems to test the techniques, and conducted an thorough experimental study of the algorithms. The results proved the practicality of this new computing paradigm.

8. REFERENCES

[1] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.
[2] H. Bast, A. Chitea, F. M. Suchanek, and I. Weber. Ester: efficient search on text, entities, and relations. In *SIGIR*, pages 671–678, 2007.
[3] H. Bast, C. W. Mortensen, and I. Weber. Output-sensitive autocompletion search. In *SPIRE*, pages 150–162, 2006.
[4] H. Bast and I. Weber. Type less, find more: fast autocompletion search with a succinct index. In *SIGIR*, pages 364–371, 2006.
[5] H. Bast and I. Weber. The completesearch engine: Interactive, efficient, and towards ir& db integration. In *CIDR*, pages 88–95, 2007.
[6] A. Behm, S. Ji, C. Li, and J. Lu. Space-constrained gram-based indexing for efficient approximate string search. In *ICDE*, 2009.
[7] K. Chakrabarti, S. Chaudhuri, V. Ganti, and D. Xin. An efficient filter for approximate membership checking. In *SIGMOD Conference*, pages 805–818, 2008.
[8] S. Chaudhuri, V. Ganti, and R. Motwani. Robust identification of fuzzy duplicates. In *ICDE*, pages 865–876, 2005.
[9] K. Grabski and T. Scheffer. Sentence completion. In *SIGIR*, pages 433–439, 2004.
[10] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava. Fast indexes and algorithms for set similarity selection queries. In *ICDE*, pages 267–276, 2008.
[11] L. Jin, N. Koudas, C. Li, and A. K. H. Tung. Indexing mixed types for approximate retrieval. In *VLDB*, pages 793–804, 2005.
[12] L. Jin and C. Li. Selectivity estimation for fuzzy string predicates in large data sets. In *VLDB*, pages 397–408, 2005.
[13] N. Koudas, C. Li, A. K. H. Tung, and R. Vernica. Relaxing join and selection queries. In *VLDB*, pages 199–210, 2006.
[14] K. Kukich. Techniques for automatically correcting words in text. *ACM Comput. Surv.*, 24(4):377–439, 1992.
[15] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.
[16] C. Li, B. Wang, and X. Yang. VGRAM: Improving performance of approximate queries on string collections using variable-length grams. In *VLDB*, pages 303–314, 2007.
[17] H. Motoda and K. Yoshida. Machine learning techniques to make computers easier to use. *Artif. Intell.*, 103(1-2):295–321, 1998.
[18] A. Nandi and H. V. Jagadish. Effective phrase prediction. In *VLDB*, pages 219–230, 2007.
[19] H. E. Williams, J. Zobel, and D. Bahle. Fast phrase querying with combined indexes. *ACM Trans. Inf. Syst.*, 22(4):573–594, 2004.
[20] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW*, pages 131–140, 2008.