Supporting Efficient Top-k Queries in Type-Ahead Search

Guoliang Li[†] Jiannan Wang[†] Chen Li[‡] Jianhua Feng[†]
[†] Department of Computer Science, Tsinghua National Laboratory for Information Science and Technology (TNList), Tsinghua University, Beijing 100084, China.
[‡] Department of Computer Science, UC Irvine, CA 92697-3435, USA
liguoliang@tsinghua.edu.cn, wjn08@mails.tsinghua.edu.cn, chenli@ics.uci.edu, fengjh@tsinghua.edu.cn

ABSTRACT

Type-ahead search can on-the-fly find answers as a user types in a keyword query. A main challenge in this search paradigm is the high-efficiency requirement that queries must be answered within milliseconds. In this paper we study how to answer top-k queries in this paradigm, i.e., as a user types in a query letter by letter, we want to efficiently find the k best answers. Instead of inventing completely new algorithms from scratch, we study challenges when adopting existing top-k algorithms in the literature that heavily rely on two basic list-access methods: random access and sorted access. We present two algorithms to support random access efficiently. We develop novel techniques to support efficient sorted access using list pruning and materialization. We extend our techniques to support fuzzy type-ahead search which allows minor errors between query keywords and answers. We report our experimental results on several real large data sets to show that the proposed techniques can answer top-k queries efficiently in type-ahead search.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: Retrieval models

General Terms

Algorithms, Experimentation, Performance

Keywords

Type-ahead search, top-k search, fuzzy search

1. INTRODUCTION

To give instant feedback when users formulate search queries, many information systems support *autocomplete* search, which shows results immediately after a user types in a partial keyword query. As an example, almost all the major search engines nowadays automatically suggest possible keyword queries as a user types in partial keywords. Most autocomplete systems treat a query with multiple keywords as *a single string*, and find answers with text that matches the string exactly. To overcome this limitation, a new type-ahead search paradigm has emerged recently [2, 13]. Using this paradigm, a system treats a query as a set of keywords, and does a *full-text* search on the underlying data to find answers including the keywords. We treat the last keyword in the query as a partial keyword the user is completing. For instance, a query "`graph sig`" on a publication table can find publication records with the keyword "`graph`" and a keyword that has "`sig`" as a prefix, such as "`sigir`", "`sigmod`", and "`signature`". In this way, a user can get instant feedback after typing keywords, thus can obtain more knowledge about the underlying data to formulate a query more easily.

Ji et al. [13] extended type-ahead search by allowing minor errors between queries and answers. As a user types in query keywords, the system can find relevant records with keywords *similar* to the query keywords. This feature is especially important when the user has limited knowledge about the exact representation of entities she is looking for. For instance, if a user types in a partial query "`chritos falut`", the system can find records approximately matching the two keywords despite the typo in the query, such as a record with keywords "Christos Faloutsos". Clearly these features can further improve user search experiences.

In this paper we study how to answer *ranking queries* in type-ahead search on large amounts of data. That is, as a user types in a keyword query letter by letter, we want to on-the-fly find the *most relevant* (or "top-k") records. One approach first finds records matching those query keywords, and then computes their ranking scores to find the most relevant ones. This approach is not efficient when there are a large number of candidate answers to compute and store. Existing type-ahead search approaches assume an index structure with a trie for the keywords in the underlying data, and each leaf node has an inverted list of records with this keyword, with the weight of this keyword in the record [13, 19]. As an example, Table 1 shows a sample collection of publication records. For simplicity, we only list some of the keywords for each record. Figure 1 shows the corresponding index structure. (More details about the index are in Section 3.)

Suppose a user types in a query "`graph icdm li`". For exact search, we find records containing the first two keywords and a word with prefix of "`li`", e.g., record r_5. For fuzzy search, we compute records with keywords *similar to* query keywords, and rank them to find the best answers. For each complete keyword, we find keywords *similar to* the query keyword. For instance, both keywords "`icdm`" and "`icdl`" are similar to the second query keyword. The last keyword

SIGIR'12, August 12–16, 2012, Portland, Oregon, USA.

Table 1: Publication records with sample keywords.

Record ID	Record
r_0	`graph icdm ...`
r_1	`graph group lui ...`
r_2	`gray icdl liu ...`
r_3	`graph icdl lin lui ...`
r_4	`graph group icdm lin liu ...`
r_5	`graph gray gross icdm lin liu ...`
r_6	`gray group icdm lin liu ...`
r_7	`gray gross group icdl lin ...`
r_8	`gross icdl liu ...`
r_9	`icdm liu ...`

"`li`" is treated as a prefix condition, since the user is still typing at the end of this keyword. We find keywords that have a prefix similar to "`li`", such as "`lin`", "`liu`", and "`lui`". We access the inverted lists of these similar keywords to find records and rank them to find the best answers for the user.

A key question is: "how to access inverted lists on trie leaf nodes efficiently to answer top-k queries?" Instead of inventing completely new algorithms from scratch, we study how to adopt a plethora of algorithms in the literature for answering top-k queries by accessing lists (e.g., [21, 12]). These algorithms share the same framework proposed by Fagin [6], in which we have lists of records sorted based on various conditions. An aggregation function takes the scores of a record from these lists and computes the final score of the record. There are two methods to access these lists: (1) *Random Access*: Given a record id, we can retrieve the score of the record on each list; (2) *Sorted Access*: We retrieve the record ids on each list following the list order.

In this paper we study technical challenges when adopting these algorithms, and focus on new optimization opportunities that arise in our problem. In particular, we study how to support the two types of access operations efficiently by utilizing characteristics specific to our index structures and access methods. We make the following contributions: 1) In Section 3, we present a forward-list-based method for supporting random access on the inverted lists, and develop a heap-based method and list-materialization techniques to support sorted access efficiently. 2) In Section 4 we study fuzzy type-ahead search. We propose a list-pruning technique to improve the performance of sorted access, and study how to improve the techniques based on forward lists and list materialization for fuzzy search. Due to the challenging nature of the problem, our extensions are technically nontrivial. 3) In Section 5 we present our experimental results on real large data sets to show the efficiency of our techniques. We have deployed several systems using this paradigm, which have been used regularly and well accepted by users due to its friendly interface and high efficiency[1].

2. FORMULATION AND PRELIMINARIES

Type-Ahead Search: Let R be a collection of records such as the tuples in a relational table. Let D be the set of words in R. Let Q be a query the user has typed in, which is a sequence of keywords $\langle w_1, w_2, \ldots, w_m \rangle$. We treat the last keyword w_m as a partial keyword the user is completing, and other keywords as complete keywords the user has completed[2]. As a user types in a keyword query letter by letter, type-ahead search on-the-fly finds records that contain the first $m-1$ keywords and a word with the last keyword as a prefix.

[1] http://tastier.ics.uci.edu/

[2] Our method can be easily extended to the case that every keyword is taken as a partial keyword.

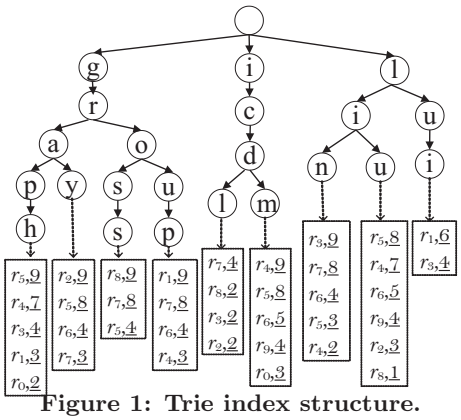


Figure 1: Trie index structure.

Without loss of generality, each string in the data set and a query is assumed to use lower-case letters. For example, in Table 1, $R = \{r_0, r_1, \ldots, r_9\}$, $D = \{\texttt{graph}, \texttt{icdm}, \texttt{group}, \texttt{lui}, \ldots\}$. Suppose a user types in a query "`icdm gra`". We treat "`icdm`" as a complete keyword and "`gra`" as a partial keyword. Records r_0, r_4, r_5, and r_6 are potentially relevant answers. For example, r_0 contains complete keyword "`icdm`" and word "`graph`" with a prefix of "`gra`". When the user types in more letters and submits query "`icdm graph li`", we treat "`icdm`" and "`graph`" as complete keywords and "`li`"as a partial keyword. Records r_4 and r_5 are potentially relevant answers.

Top-k Answers: We rank each record r in R based on its relevance to the query. Given a positive integer k, our goal is to compute the best k records in R ranked by their relevance to Q. Notice that our problem setting allows an important record to be in the answer, even if not all query keywords appear in the record (the "OR" semantics). Thus the algorithms in [13] cannot be used directly in our problem.

Ranking: In the literature there are many algorithms for answering top-k queries by accessing lists (e.g., [21, 12]). These algorithms share the same framework proposed by Fagin [6], in which we have lists of records sorted based on various conditions, such as term frequency and inverse document frequency ("tf*idf"). Each record has a score on a list, and we use an aggregation function to combine the scores of the record on different lists to compute its overall relevance to the query. The aggregation function needs to be *monotonic*, i.e., decreasing the score of a record on a list cannot increase the record's overall score. This approach has the advantage of allowing a general class of ranking functions. In this paper, we focus on an important class of ranking functions with the following property: the score $F(r, Q)$ of a record r to a query Q is a monotonic combination of scores of the query keywords with respect to the record r. Formally, we compute the score $F(r, Q)$ in two steps. In the first step, for each keyword w, we compute a score of the keyword with respect to the record r, denoted by $F(r, w)$. In the second step, we compute the score $F(r, Q)$ by applying a monotonic function on the $F(r, w)$'s for all the keywords w. The intuition of this property is that the more relevant an individual query keyword is to a record, the more likely this record is a good answer to this query. For example, we compute the score of a record to query "`icdm graph li`" by aggregating the scores of each of keywords with respect to the record.

Each complete keyword w has a weight associated with a record r, denoted by $W(r, w)$. This weight could depend

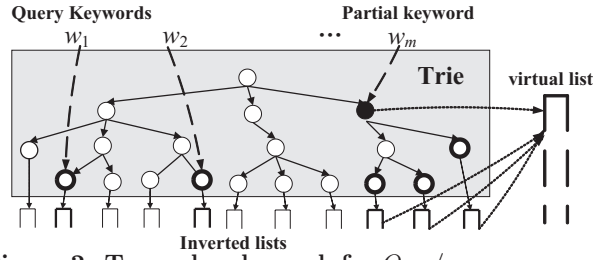


Figure 2: Type-ahead search for $Q = \langle w_1, w_2, \ldots, w_m \rangle$.

on the keyword, such as the tf*idf value of the keyword in the record. As a specific case, it can also be independent from the keyword. For instance, if a record is a URL with tokenized keywords, its weight could be a rank score of the corresponding Web page. If a record is an author, we can use the number of publications of the author as a weight of this record. For the last partial keyword w_m, there could be multiple complete words. We compute the relevance score of w_m in the record, i.e., $F(r, w_m)$, based on the following property: $F(r, w_m)$ is the maximal value of the $W(r, d)$ weights for all the keywords d with respect to w_m in r, where d is a keyword in record r and has a prefix of w_m. This property states that we only look at the most relevant keyword in a record to the partial keyword when computing the relevance of the keyword to the record. It means that the ranking function is "greedy" to find the most relevant keyword in the record as an indicator of how important this record is to the partial keyword. As we can see in Section 3, this property allows us to do effective pruning when accessing the multiple lists of a query keyword. The following is an example function.

$$F(r, Q) = \sum_{i=1}^{m} F(r, w_i), \tag{1}$$

where

$$F(r, w_i) = \begin{cases} W(r, w_i) & if\ 1 \le i < m, \\ \max_{complete\ word\ d\ of\ w_m}\{W(r, d)\} & if\ i = m. \end{cases} \tag{2}$$

In Figure 1, consider query "`icdm graph li`" and record r_5. $F(r_5,$ "`icdm`"$) = W(r_5,$ "`icdm`"$) = 8$ and $F(r_5,$ "`graph`"$) = W(r_5,$ "`graph`"$) = 9$. The partial keyword "`li`" has two complete words "`lin`" and "`liu`". $F(r_5,$ "`li`"$) = max\{W(r_5,$ "`lin`"$), W(r_5,$ "`liu`"$)\}$=8. $F(r_5,$ "`icdm graph li`"$) = 25$.

3. EXACT TYPE-AHEAD SEARCH

In this section, we study efficient list-access methods to support *exact* type-ahead search, i.e., no mismatches between query keywords and answers.

Indexing: We construct a trie for the data keywords in the data D. A trie node has a character label. Each keyword in D corresponds to a unique path from the root to a leaf node[3] on the trie. For simplicity, a trie node is mentioned interchangeably with the keyword corresponding to the path from the root to the node. A leaf node has an inverted list of IDs of pairs $\langle rid, weight \rangle$, where rid is the ID of a record containing the leaf-node string, and $weight$ is the weight of the keyword in the record. Figure 1 shows the index structure in our running example. For instance, for the leaf node of keyword "`graph`", its inverted list has five elements.

[3]A common "trick" to make each leaf node corresponds to a complete word and vice versa is to add a special mark to the end of each word. For simplicity we did not use this trick.

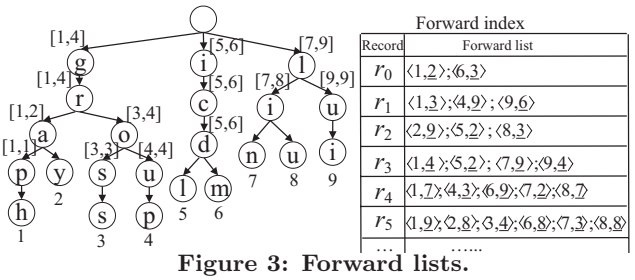


Figure 3: Forward lists.

The first element "$\langle r_5, 9 \rangle$" indicates that the record r_5 has this keyword, and the weight of this keyword in this record is "9", i.e., $W(r_5,$ "`graph`"$) = 9$.

Searching: We compute the top-k answers to a query Q in two steps. As illustrated in Figure 2, in the first step, for each complete keyword $w_i (1 \le i \le m-1)$, we get its inverted list. For the last partial keyword, we locate the trie node of w_m and retrieve the inverted lists of the trie node's leaf descendants. For example, in Figure 1, consider a query "`icdm li`". The partial keyword "`li`" has two leaf-node keywords: "`lin`" and "`liu`". In the second step, we access the inverted lists to compute the k best answers.

Many algorithms have been proposed for answering top-k queries by accessing sorted lists [12, 6]. When adopting these algorithms to solve our problem, we need to efficiently support two basic types of access used in these algorithms: random access and sorted access on the lists.

3.1 Efficient Random Access

To support random access, we construct a forward index in which each record has a forward list of IDs of its keywords. We assume each keyword has a unique ID with respect to its leaf node on the trie, and the IDs of the keywords follow their alphabetical order. Figure 3 shows the forward lists. The element "$\langle 1, 9 \rangle$" on the forward list of record r_5 shows that this record has a keyword with ID 1 and weight 9, which is keyword "`graph`" as shown on the trie.

Given a record and a complete keyword, we can get the corresponding weight by doing a binary-search on the forward list. For example, to get the weight of keyword "`icdm`" with ID 6 in r_5, we can do a binary search on r_5's forward list and get the corresponding weight 8. For the partial keyword, as it has multiple complete words, we need first locate its trie node and then enumerate its leaf-descendants to get the corresponding weights. This method could be expensive if the trie node has many leaf-descendants. To improve the performance, we can use an alternative method. For each trie node n, we can maintain a keyword range $[l_n, u_n]$, where l_n and u_n are the minimal and maximal keyword IDs of its leaf nodes, respectively [13]. An interesting observation is that a complete word with n as a prefix must have an ID in this keyword range, and each complete word in the data set with an ID in this range must have a prefix of n. In Figure 3, the keyword range of node "`g`" is $[1, 4]$, since 1 is the smallest ID of its leaf nodes and 4 is the largest one.

Based on this observation, this method verifies whether record r contains a keyword with a prefix of w_m as follows. We first locate the trie node w_m and then check if there is a keyword ID on the forward list of r in the keyword range $[l_{w_m}, u_{w_m}]$. Since we can keep the forward list of r sorted, this checking can be done efficiently. For instance, consider query "`graph icdm l`". For the first element on the inverted list of "`graph`", $\langle r_5, 9 \rangle$, we can check whether

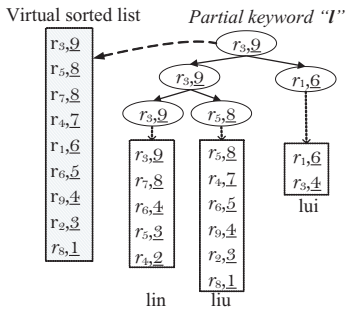


Figure 4: A heap-based method to compute the virtual sorted list of partial keyword "l".

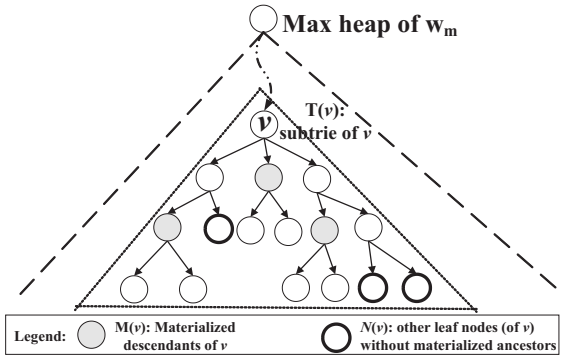


Figure 5: Benefits of materializing the union list $U(v)$ for node v with respect to partial keyword w_m.

r_5 contains other two keywords as follows. For complete keyword "icdm" with ID 6, we do a binary search on r_5's forward list and get weight 8. For partial keyword "l" with keyword range $[7, 9]$, using a binary search on r_5's forward list $\big(\langle 1, 9\rangle; \langle 2, 8\rangle; \langle 3, 4\rangle; \langle 6, 8\rangle; \langle 7, 3\rangle; \langle 8, 8\rangle\big)$, we find keyword IDs 7 and 8 in this range. Thus we know that the record indeed contains keywords with prefix "l", and compute the corresponding score $F(r_5, \text{"l"}) = max\big\{F(r_5, \text{"lin"}), F(r_5, \text{"liu"}), F(r_5, \text{"lui"})\big\} = 8$. Thus $F(r_5, \text{"graph icdm l"}) = 25$.

3.2 Efficient Sorted Access

To support sorted access, we can keep the elements on the inverted lists sorted based on their weights in a descending order. Thus, for the complete keyword, we can get an ordered list. For the partial keyword w_m, it has multiple leaf descendants and corresponding inverted lists. We use $U(w_m)$ to denote the union of those inverted lists, called *union list* of w_m. We need to support sorted access on $U(w_m)$ to retrieve the next most relevant record ID for w_m. Fully computing $U(w_m)$ using the keyword lists could be expensive in terms of time and space. In this section, we propose two techniques to support sorted access efficiently.

3.2.1 Heap-Based Method

We can support sorted access on $U(w_m)$ by building a max heap on the inverted lists of its leaf nodes. In particular, we maintain a cursor on each inverted list. The max heap initially consists of the record IDs pointed by the cursors so far, sorted on the weights of the keywords in these records. Notice that each inverted list is already sorted based on the weights of its keyword in the records. To retrieve the next best record, we pop the top element from the heap, increment the cursor of the list of the popped element by 1, and push the new element of this list to the heap. When popping all elements from the heap, we can get a sorted list for the partial keyword. For example, consider the partial keyword "l". It has three complete keywords "lin", "liu", and "lui". We can compute its union list as shown in Figure 4. Note that since our method does not need to compute the entire list of $U(w_m)$, $U(w_m)$ is a *virtual sorted list* of partial keyword w_m. On top of the inverted lists of complete keywords and the max heap of the partial keyword, we can adopt an existing top-k algorithm to find the k best records.

As an example, suppose we want to compute the top-1 best answer for query "graph icdm l" using sorted access only. We get the first elements of "graph" and "icdm", $\langle r_5, 9\rangle$ and $\langle r_4, 9\rangle$, pop the top element of the max heap in Figure 4, $\langle r_3, 9\rangle$, and compute an upper bound on the overall score of an answer, i.e., 27. We get the next elements of "graph" and "icdm", $\langle r_4, 7\rangle$ and $\langle r_5, 8\rangle$. We increment the cursor of the list that produces the top element, push it into the heap, and retrieve the next top element: $\langle r_5, 8\rangle$. Based on the accessed elements, we have 1) The score of record r_5 is $9+8+8 = 25$; 2) The maximal score of record r_3 is $7 + 8 + 9 = 24$, and that of r_4 is $7 + 9 + 8 = 24$, while those of other records are at most $7 + 8 + 8 = 23$. Thus, record r_5 is the best answer.

3.2.2 List Materialization

We can further improve the performance of sorted access for the partial keyword w_m by precomputing and storing the unions of some of the inverted lists on the trie. Let v be a trie node, and $U(v)$ be the union of the inverted lists of v's leaf nodes, sorted by their record weights. If a record appears more than once on these lists, we choose its maximal weight as its weight on list $U(v)$. For example, $U(\text{"li"}) = \{\langle r_3, 9\rangle, \langle r_5, 8\rangle, \langle r_7, 8\rangle; \langle r_4, 7\rangle, \langle r_6, 5\rangle, \langle r_9, 4\rangle, \langle r_2, 3\rangle, \langle r_8, 1\rangle\}$. When using a max heap to retrieve records sorted by their scores for the partial keyword, this materialized list could help us build a max heap with fewer lists and reduce the cost of push/pop operations on the heap. Therefore, this method allows us to utilize additional memory space to answer top-k queries more efficiently. For instance, consider the index in Figure 1 and a query "icdm gr". For the partial keyword "gr", we access its data keywords "graph", "gray", "gross", and "group", and build a max heap on their inverted lists based on record scores with respect to this query keyword. If we materialize the union lists of "gra" and "gro", we can use their materialized lists, saving the time to traverse the four leaf nodes and some push/pop operations on the heap.

We next give a detailed cost-based analysis to quantify the benefit of materializing a node on the performance of operations on the max heap of w_m, for exact type-ahead search. Let B be a budget of storage space we are given to materialize union lists. Given a trie node v, let $U(v)$ be the union of inverted lists of leaf nodes in the subtrie of v. Our goal is to select trie nodes to materialize their union lists for maximizing the performance of queries. The following are naive algorithms for choosing trie nodes:

- Random: We randomly select trie nodes.
- TopDown: We select nodes top down from the trie root.
- BottomUp: We select nodes bottom up from leaf nodes.

Each naive approach keeps choosing trie nodes to materialize their union lists until the sum of their list sizes reaches the space limit B. One main limitation of these approaches is that they do not quantitatively consider the benefits of

materializing a union list. To overcome this limitation, we propose a cost-based method called CostBased to do list materialization. Its main idea is the following.

For simplicity we say a node has been "*materialized*" if its union list has been materialized. For a query Q with a prefix keyword w_m, suppose some of the trie nodes have their union lists materialized. Let v be such a materialized node. If we can use $U(v)$ to construct the heap of w_m, we need not visit v's descendants and access the inverted lists of v's leaf descendants, and thus achieve the benefit of reducing the time of traversing the subtrie rooted at v and push/pop operations on the max heap of w_m. We say the materialized node v is *usable* for partial keyword w_m.

Next we discuss how to check whether a node v is usable for partial keyword w_m. If v is not a descendant of w_m, materializing v is unusable to w_m; otherwise, if no node on the path from v to w_m (including w_m) has been materialized, materializing v is usable to w_m. Notice that if v has a materialized ancestor v' on the path from v to w_m, then we can use the materialized list $U(v')$ instead of $U(v)$, and the list $U(v)$ will no longer be usable to w_m. To summarize, a materialized node v is *usable* for partial keyword w_m if,

1. v is a descendant of w_m; and
2. v has no materialized ancestor between v and w_m.

For example, consider a query "`icdm g`", materializing node "`l`" is unusable for partial keyword "`g`" as "`l`" is not a descendant of "`g`". Materializing "`gr`" is usable for "`g`" if "`g`" is not materialized. If "`gr`" is materialized, then materializing "`gra`" is unusable for "`g`" as we will use the materialized list of "`gr`" to build the max heap of "`g`", instead of using "`gra`".

If v is usable for w_m, materializing $U(v)$ has the following benefits for the heap of w_m. (1) We do not need to traverse the trie to access these leaf nodes and use them to construct the max heap; (2) Each push/pop operation on the heap is more efficient since it has fewer lists. Here we present an analysis of the benefits of materializing the usable node v. In general, for a trie node v, let $T(v)$ denote its subtrie and $|T(v)|$ denote the number of nodes in $T(v)$. The total time of traversing this subtrie is $\mathcal{O}\big(|T(v)|\big)$.

Now we analyze the benefit of materializing node v. As illustrated in Figure 5, suppose v has materialized descendants. Let $M(v)$ be the set of highest materialized descendants of v. These materialized nodes can help reduce the time of accessing the inverted lists of v's leaf nodes in two ways. First, we do not need to traverse the descendants of a materialized node $d \in M(v)$. We can just traverse $|T(v)|-\sum_{d\in M(v)}|T(d)|$ trie nodes. Second, when inserting lists to the max heap of w_m, we insert the union list of v into the heap and need not insert the union list of each $d \in M(v)$ and the inverted lists of $d' \in N(v)$ into the heap, where $N(v)$ denotes the set of v's leaf descendants having no ancestors in $M(v)$. Let $S(v) = M(v) \cup N(v)$. We quantify benefits of materializing node v:

1. *Reducing traversal time*: Since we do not traverse v's descendants, the time reduction is $B_1 = \mathcal{O}\big(|T(v)| - \sum_{d\in M(v)}|T(d)|\big)$.
2. *Reducing heap-construction time*: When constructing the max heap for keyword w_m, we insert the union list $U(v)$ into the heap, instead of the inverted lists of those nodes in $S(v)$. The time reduction is $B_2 = |S(v)| - 1$.
3. *Reducing sorted-access time*: If we insert the union list $U(v)$ to the max heap of w_m, the number of leaf nodes in the heap is $|S(w_m)|$. Otherwise, it is $|S(w_m)| + |S(v)| - 1$. The time reduction of a sorted access is $B_3{=}\mathcal{O}\big(log(|S(w_m)|{+}|S(v)|{-}1)\big) - \mathcal{O}\big(log(|S(w_m)|)\big)$.

The following is the overall benefit of materializing v for the partial keyword w_m:

$$B_v = B_1 + B_2 + A_v * B_3, \tag{3}$$

where A_v is the number of sorted accesses on $U(v)$. A_v can be computed using the number of records in the union list $U(v)$, and the number of keywords in the query.

The analysis above is on a query workload. If there is no query workload, we can use the trie structure to count the probability of each node to be queried and use such information to compute the benefit of materializing a node. In this paper, we employ a no query workload setting.

4. FUZZY TYPE-AHEAD SEARCH

In this section, we first define the problem of top-k queries in fuzzy type-ahead search [13]. We then develop new techniques to support efficient list access to answer such queries by extending techniques developed in exact search.

4.1 Ranking

As a user types in a query letter by letter, fuzzy type-ahead search on-the-fly finds records with words *similar to* the query keywords. For example, consider the data in Table 1. Suppose a user types in a query "`graph grose`". We return r_5 as a relevant answer since it has a keyword "`gross`" similar to query keyword "`grose`". We use edit distance to measure the similarity between strings. Formally, the edit distance between two strings s_1 and s_2, denoted by $\mathsf{ed}(s_1, s_2)$, is the minimum number of single-character edit operations (i.e., insertion, deletion, and substitution) needed to transform s_1 to s_2. For example, $\mathsf{ed}(\texttt{gross}, \texttt{grose}) = 1$.

Similarity Function: Let π be a function that computes the similarity between a data string s and a query keyword w in $Q = \langle w_1, w_2, \ldots, w_m\rangle$. An example is:

$$\pi(s, w) = 1 - \frac{\mathsf{ed}(s, w)}{|w|},$$

where $|w|$ is the length of the query keyword w. We normalize the edit distance based on the query-keyword length in order to allow more errors for longer query keywords. Our results in the paper focus on this function, and they can be generalized to other functions using edit distance.

Let d be a keyword in the data set D. For each complete keyword w_i $(i = 1, 2, \ldots, m-1)$ in the query, we define the similarity of d to w_i as:

$$\mathsf{Sim}(d, w_i) = \pi(d, w_i).$$

Since the last keyword w_m is treated as a prefix condition, we define the similarity of d to w_m as the maximal similarity of d's prefixes using function π, i.e.:

$$\mathsf{Sim}(d, w_m) = \max_{\text{prefix } p \text{ of } d}\{\pi(p, w_m)\}.$$

Let τ be a similarity threshold. We say a keyword d in D is *similar to* a query keyword w if $\mathsf{Sim}(d, w) \geq \tau$. We say a prefix p of a keyword in D is *similar to* the query keyword w_m if $\pi(p, w_m) \geq \tau$. We want to find the keywords in the data set that are similar to query keywords, since records with such a keyword could be of interest to the user.

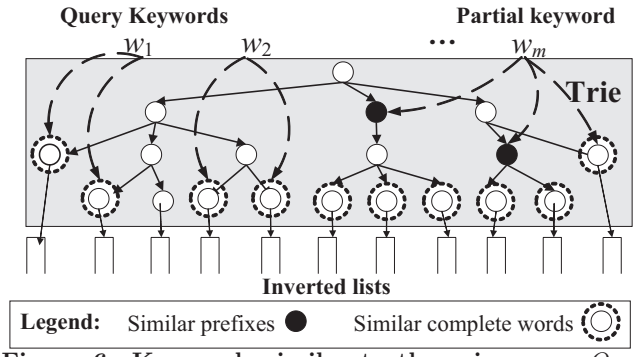


Figure 6: Keywords similar to those in query $Q = \langle w_1, w_2, \ldots, w_m \rangle$. Each query keyword w_i has similar keywords on leaf nodes. The last prefix keyword w_m has similar prefixes.

Let $\Phi(w_i)$ $(i = 1, \ldots, m)$ denote the set of keywords in D similar to w_i, and $P(w_m)$ denote the set of prefixes (of keywords in D) similar to w_m. We compute the top-k answers to the query Q in two steps. In the first step, for each keyword w_i in the query, we first compute an edit-distance upper bound based on the similarity function, i.e., $(1 - \tau) * |w_i|$, and then compute the similar keywords $\Phi(w_i)$ and similar prefixes $P(w_m)$ on the trie (shown in Figure 6). Ji et al. [13] developed an efficient algorithm for incrementally computing these similar strings as the user modifies the current query. A similar algorithm is developed in [5]. In the second step, we access the inverted lists of these similar data keywords to compute the k best answers.

For example, assume a user types in a query "`grose li`" letter by letter on the data shown in Table 1. Suppose the similarity threshold τ is 0.45. The set of prefixes similar to the partial keyword "`li`" is $P(\text{"li"}) = \{\texttt{l}, \texttt{li}, \texttt{lin}, \texttt{liu}, \texttt{lu}, \texttt{lui}, \texttt{i}\}$, and the set of data keywords similar to the partial keyword "`li`" is $\Phi(\text{"li"}) = \{\texttt{lin}, \texttt{liu}, \texttt{lui}, \texttt{icdl}, \texttt{icdm}\}$. In particular, "`lui`" is similar to "`li`" since $\mathsf{Sim}(\texttt{lui}, \texttt{li}) = 1 - \frac{\texttt{ed(lui,li)}}{|\texttt{li}|} = 0.5 \geq \tau$. The set of similar words for the complete keyword "`grose`" is $\Phi(\text{"grose"}) = \{\texttt{gross}\}$. Then we compute top-$k$ answers using the inverted lists of those words in $\Phi(\text{"grose"})$ and $\Phi(\text{"li"})$.

Ranking: We still assume the ranking function has the first property described in Section 2, which computes the score $F(r, Q)$ by applying a monotonic function on the $F(r, w_i)$'s for all the keywords w_i in the query. Given a complete keyword w_i and a record r, for exact search, we can use the weight of w_i in r, i.e., $W(r, w_i)$, to denote their relevancy $F(r, w_i)$. But for fuzzy search, the keyword w_i can be similar to multiple keywords in the record r, and different similar words have different similarities to w_i and different weights in r. A question is how to compute the relevance value of keyword w_i in record r, $F(r, w_i)$.

Let d be a keyword in record r such that d is similar to the query keyword w_i, i.e., $d \in \Phi(w_i)$. We use $F(r, w_i, d)$ to denote the relevance of this query keyword w_i in the record with respect to keyword d. The value should depend on both the weight of d in r, i.e., $W(r, d)$, as well as the similarity between w_i and d, i.e., $\mathsf{Sim}(d, w_i)$. Intuitively, the more similar they are, the more relevant w_i is to r in terms of d. For instance, $F(r, w_i, d) = \mathsf{Sim}(d, w_i) * W(r, d)$ is an example ranking function to evaluate the relevancy of w_i in the record with respect to keyword d. We use the following function with the second property in Section 2 to compute $F(r, w_i)$:

$$F(r, w_i) = \max_{\text{keyword } d \text{ (in } r\text{) similar to } w_i} \{F(r, w_i, d)\}. \quad (4)$$

4.2 Efficient Random Access

We first study how to support efficient random access for fuzzy type-ahead search. For simplicity, in the discussion we focus on how to verify whether the record has a keyword with a prefix similar to the partial keyword w_m. With minor modifications the discussion extends to the case where we want to verify whether r has a keyword similar to a complete keyword $w_i (1 \leq i \leq m - 1)$.

In each random access, given an ID of a record r, we want to retrieve information related to a query keyword w_i, which allows us to retrieve $W(r, d)$ for each of w_i's similar word d so as to compute the score $F(r, w_i)$. In particular, for a keyword w_i in the query, does the record r have a keyword similar to w_i? One naive way to get the information is to retrieve the original record r and go through its keywords. This approach has two limitations. First, if the data is too large to fit into memory and has to reside on hard disks, accessing the original data from the disks may slow down the process significantly. This costly operation will prevent us from achieving an interactive-search speed. The second limitation is that it may require a lot of computation of string similarities based on edit distance, which could be time consuming. In this section, we present two efficient approaches for solving this problem.

Method 1: Probing on Forward Lists: This method verifies whether record r contains a keyword with a prefix similar to w_m as follows. For each prefix p on the trie similar to w_m (computed in the first step of the algorithm as discussed above), we check if there is a keyword ID on the forward list of r in the keyword range $[l_p, u_p]$ of the trie node of p as discussed in Section 3.

Method 2: Probing on Trie Leaf Nodes: Using this method, for each prefix p similar to w_m, we traverse the subtrie of p and identify its leaf nodes. For each leaf node d, we store the fact that for the query Q, this keyword d has a prefix similar to w_m in the query. Specifically, we store

$$\langle \text{Query ID, partial keyword } w_m, \mathsf{Sim}(p, w_m) \rangle.$$

We store the query ID in order to differentiate it from other queries in case multiple queries are answered concurrently. We store the similarity between w_m and p to compute the score of this keyword in a candidate record. In case the leaf node has several prefixes similar to w_m, we only keep their maximal similarity to w_m. For each complete keyword w_i, we also store the same information for those trie nodes similar to w_i. Therefore, a leaf node might have multiple entries corresponding to different keywords in the same query. We call these entries for the leaf node as its collection of *relevant query keywords*. Notice that this structure needs very little storage space, since the entries of old queries can be quickly reused by new queries, and the number of keywords in a query tends to be small. We use this additional information to efficiently check if a record r contains a complete word with a prefix similar to the partial keyword w_m. We scan the forward list of r. For each of its keyword IDs, we locate the corresponding leaf node, and test whether its collection of relevant query keywords includes this query and

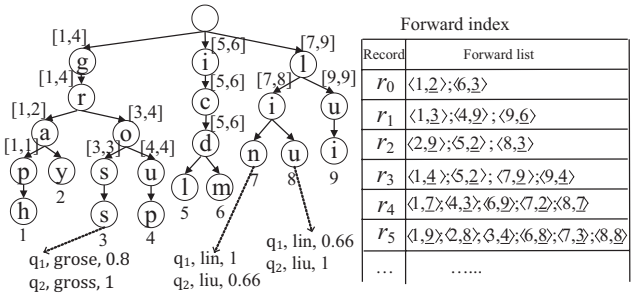


Figure 7: Probing on trie leaf nodes.

the keyword w_m. If so, we use the stored string similarity to compute the score of this keyword in the query.

Figure 7 shows how we use this method in our running example, where the user types in a keyword query $q_1 = \langle \texttt{lin}, \texttt{grose} \rangle$. When computing the similar words of "`grose`", i.e., "`gross`", we insert the query ID (shown as "q_1"), the partial keyword "`grose`", and the corresponding prefix similarity to its collection of relevant query keywords. To verify whether record r_5 has a word with a prefix similar to "`grose`", we scan its forward list. Its third keyword is "`gross`". We access its corresponding leaf node, and see that the node's collection of relevant query keywords includes "`grose`". Thus we know that r_5 indeed contains a keyword similar to "`grose`", and can retrieve the corresponding prefix similarity.

Comparison: The time complexity of the forward-list based method (Method 1) is $\mathcal{O}\big(G * log(|r|)\big)$, where G is the total number of similar prefixes of w_m and similar complete words of w_i's for $1 \leq i \leq m-1$, and $|r|$ is the number of distinct keywords in record r. Since the similar prefixes of w_m could have ancestor-descendant relationships, we can optimize the step of accessing them by considering the "highest" ones. The time complexity of the second method is

$$\mathcal{O}\Big(\sum_{\text{similar prefix } p \text{ of } w_m} |T(p)| + |r| * |Q|\Big).$$

The first term corresponds to the time of traversing the subtries of similar prefixes, where $T(p)$ is the subtrie rooted at a similar prefix p. The second term corresponds to the time of probing the leaf nodes, where $|Q|$ is the number of query keywords. Notice that to identify the answers, we need access the inverted lists of complete words, thus the first term can be removed from the complexity. Method 1 is preferred for data sets where records have a lot of keywords such as long documents, while Method 2 is preferred for data sets where records have a small number of keywords such as relational tables with relatively short attribute values.

4.3 Efficient Sorted Access

Heap-Based Method: For a query keyword w, we want to support sorted access that can access record IDs based on the relevance of w to these records. As w has multiple similar words, we can support sorted access efficiently by building a max heap on the inverted lists of such similar words, as described in Section 3. Notice that, in exact search, each leaf node has the same similarity to w; but for fuzzy search, different leaf nodes could have different similarities. Thus, when pushing a record r from an inverted list of a similar word d to the heap, we maintain $\langle r, F(r,d) \rangle$ in the heap. We push/pop the record on the heap with the maximal $F(r,d)$.

Consider the query "`icdm li`". Figure 8 shows the two heaps for the two keywords. For illustration purposes, for

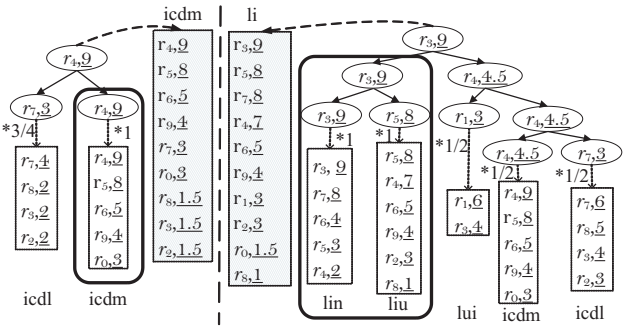


Figure 8: Max heaps for the query keywords "`icdm`" and "`li`". Each shaded list is merged from the underlying lists. It is "virtual" since we do not need to compute the entire list.

each keyword we also show the virtual merged list of records with their scores, and this list is only partially computed during the traversal of the underlying lists. Each record on a heap has an associated score of this keyword with respect to the query keyword, computed using Equation 4.

List Pruning: As there may be a large number of similar words for a query keyword, especially for the partial keyword, it could be expensive to construct a heap on the fly. We further improve the performance of sorted access on the virtual sorted list $U(w)$ by using the idea of "on-demand heap construction," i.e., we want to avoid constructing a heap for all the inverted lists of keywords similar to a query keyword. Suppose w has t similar words. Each push/pop operation on the heap of these lists takes $\mathcal{O}(log(t))$ time. If we can reduce the number of lists on the heap, we can reduce the cost of its push/pop operations. We have two observations about this pruning method. (1) As a special case, if those keywords matching query keywords *exactly* have the highest relevance scores, this method allows us to consider these records prior to considering other records with mismatching keywords. (2) The pruning can be more powerful if w is the last partial keyword w_m, since many of its similar keywords share the same prefix p on the trie.

Consider query "`icdm li`", Figure 8 illustrates how we can prune low-score lists and do on-demand heap constructions. The prefix "`li`" has several similar keywords. Among them, the two words "`lin`" and "`liu`" have the highest similarity value to the query keyword, mainly because they have a prefix matching the keyword exactly. We build a heap using these two lists. To compute the top-1 best answer, the lists of "`lui`", "`icdm`", and "`icdl`" are never included in the heap since their upper bounds are always smaller than the scores of popped records before the traversal terminates.

We next introduce how to do list pruning for the max-heap based methods in fuzzy type-ahead search. Given a keyword w, let $d_1, \ldots, d_t$ be its similar words and $L_1, \ldots, L_t$ be the corresponding inverted lists, respectively. We need not use all the inverted lists to build the max heap of w. Instead, we use those with higher similarities to w to "*on-demand build the max heap*". We first sort these inverted lists based on the similarities of their keywords to w, without loss of generality, suppose $\mathsf{Sim}(d_1, w) > \ldots > \mathsf{Sim}(d_t, w)$. We first construct the max heap using the lists with the highest similarity values and then include other lists on-demand.

Suppose L_i is a list not included in the heap so far. We can derive an upper bound u_i on the score of a record from L_i (with respect to the query keyword w) using the largest

weight on the list and the string similarity $\mathsf{Sim}(d_i, w)$. Let r be the top record on the heap, with a score $F(r, w)$. If $F(r, w) \geq u_i$, then this list does not need to be included in the heap, since it cannot have a record with a higher score. Otherwise, this list needs to be included in the heap. Based on this analysis, each time we pop a record from the heap and push a new record r, we compare the score of the new record with the upper bounds of those lists not included in the heap so far. For those lists with an upper bound greater than this score, they need to be included in the heap from now on. Notice that this checking can be done very efficiently by storing the maximal value of these upper bounds, and ordering these lists based on their upper bounds. The pruning power can be even more significant if the keyword w is the partial keyword w_m, since many of its similar keywords share the same prefix p on the trie similar to w_m. We can compute an upper bound of the record scores from these lists and store the bound on the trie node p. In this way, we can prune the lists more effectively by comparing the value $F(r, w)$ with this upper bound stored on the trie, without needing to on-the-fly compute the bound.

List Materialization: For fuzzy search, the partial keyword w_m has multiple similar prefixes and each similar prefix has multiple similar words. The max heap of w_m is built on top of inverted lists of such similar words. Let d be such a similar word. Recall that the value $F(r, w_m, d)$ of a record r on the list of a similar word d with respect to w_m is based on both $W(d, r)$ and $\mathsf{Sim}(d, w_m)$. Let v be a materialized node. To use $U(v)$ to replace the lists of v's leaf nodes in the max heap, the following two conditions need to be satisfied:

- All the leaf nodes of v have the same similarity to w_m.
- All the leaf nodes of v are similar to w_m, i.e., their similarity to w_m is no less than the threshold τ.

When the conditions are satisfied, the sorting order of the union list $U(v)$ is also the order of the scores of the records on the leaf-node lists with respect to w_m. A materialized node v that satisfies the two conditions must be a descendant of a similar prefix of partial keyword w_m. We can prove this by contradiction. Suppose node v is not a descendant of any similar prefix of partial keyword w_m. Then node v and its ancestors are not similar prefixes of w_m, that is the leaf nodes of v are not similar keywords of w_m. This is contradicted with the second condition. Thus a materialized node v that satisfies the two conditions must be a descendant of a similar prefix of partial keyword w_m.

Suppose $p_1, p_2, \ldots, p_n$ are similar prefixes of w_m. We check whether their materialized descendants satisfy the two conditions as follows. Consider a materialized node v which has ancestors among $p_1, p_2, \ldots, p_n$. If node v has no descendants that are similar prefixes of w_m, v must satisfy the two conditions; otherwise suppose p_j is a descendant of v that is a similar prefix of w_m and has the largest similarity to v among all such descendants. Without loss of generality, let p_i be an ancestor of v and has the largest similarity with v among all similar prefixes. If $\mathsf{Sim}(v, p_j) \leq \mathsf{Sim}((v, p_i)$, v satisfies the two conditions; otherwise v will not. Thus we can find *usable* materialized nodes to construct the max heap of w_m and use our proposed techniques in Section 3.2.2 to do a cost-based analysis to select high-quality nodes for materialization.

5. EXPERIMENTS

We implemented our proposed techniques and compared with existing methods on three real data sets. (1) "DBLP": It included computer science publication records[4]. (2) "URL"[5]: It included 10 million URLs. (3) "Enron": It was an email collection[6]. Table 2 shows details of the data.

Table 2: Data sets and index costs.

Data Set	URL	DBLP	Enron
# of Records (millions)	10	1	0.5
Data size	1.1 GB	500 MB	1.4 GB
Avg. # of words/record	7.7	17.1	271.7
# of distinct keywords (millions)	1.79	0.392	1.26
Trie size	421 MB	31 MB	128 MB
Size of inverted lists	379 MB	83 MB	342 MB

For the DBLP data set, we selected 1000 real queries from the logs of our deployed systems and each query contained 1-6 keywords[7]. For the other two data sets, we generated 1000 queries with keywords randomly selected from the set of words used in the collection. We assumed the letters of a query were typed in one by one. For each keystroke, we measured the time of computing the top-k answers to this query. For exact search, we measured the total running time. For fuzzy search, we measured the time in two steps: in step 1 we computed keywords on the trie similar to the query keywords (using the algorithm described in [13]); in step 2 we found the top-k answers using the inverted lists of these similar keywords. Unless otherwise specified, $k = 10$.

We compared our method with state-of-the-art method [13]. We implemented the NRA algorithm described in [6] if we only do sorted access, and the Threshold Algorithm ("TA") if we can do both sorted access and random access.

All the indexes were built off-line and pre-loaded and full-resident in memory during all querying operations. All experiments were run on a Ubuntu Linux machine with an Intel Core processor (X5450 3.00GHz and 4 GB RAM).

5.1 Exact Search

Sorted Access Only: We implemented the following methods. (1) BinaryProbe [13]: We considered the inverted lists of the complete query keywords, and the union of the inverted lists for the complete keywords of the partial keyword. We chose the shortest list, and for each of its record IDs, we did binary probings on other lists. (2) NRA(Heap): We implemented the NRA algorithm using the heap-based technique. (3) NRA(Heap+Materialization[8]): We implemented the NRA algorithm using the heap-and-materialization-based techniques. Figure 9 shows the results on the Enron dataset, which showed that our method improved search efficiency. For instance, for queries with a partial keyword of length 2, NRA(Heap) reduced the query time of BinaryProbe from 128 ms to 10 ms. NRA(Heap+Materialization) further reduced the time to 2 ms. This is because 1) BinaryProbe first computed all results and then ranked them; 2) BinaryProbe on-the-fly computed the union list of the partial keyword. NRA(Heap) used the max heap to generate a sorted *partial* list and NRA(Heap+Materialization) used materialized lists to save push/pop operations on the heap.

Sorted Access + Random Access: We implemented the following methods. (1) BinaryProbe (Forward List)[13], we chose the shortest list, and for each of its record IDs, we verified whether the record ID contained other keywords

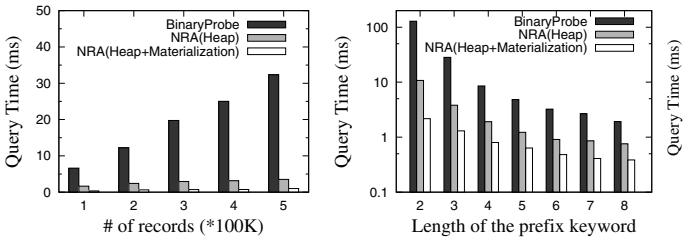
[4] http://dblp.uni-trier.de/xml/

[5] http://www.sogou.com/labs/dl/t-rank.html

[6] http://www-2.cs.cmu.edu/∼enron/

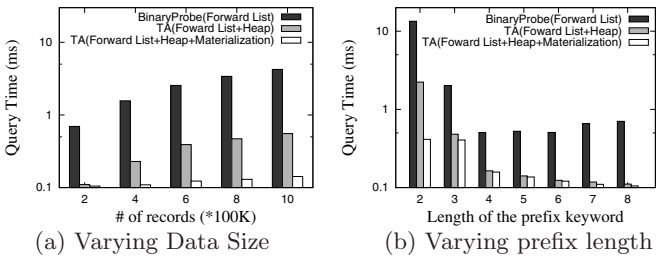
[7] Details are omitted due to double-blind review.

[8] We used additional 50% space with respect to inverted index for materialization in the experiments.



(a) Varying Data Size (b) Varying prefix length

Figure 9: Exact search using sorted access (Enron).



(a) Varying Data Size (b) Varying prefix length

Figure 10:Exact search using random access(DBLP).

using the forward list. (2) TA(Forward List+Heap): We implemented the TA algorithm using forward list for random access and max heap for sorted access. (3) TA(Forward List+Heap+Materialization): We implemented the TA algorithm using forward list, max heap, and list materialization. Figure 10 shows the results on the DBLP dataset. We can see that the random-access techniques indeed improved efficiency.

5.2 Fuzzy Search

Sorted Access Only: We first evaluated the effect of the list-pruning technique. Figure 11 shows the experimental results (including two steps). We can observe that list pruning indeed improved search efficiency. For the Enron dataset with 0.5M records, the method with pruning can reduce the time from 30 ms to 17 ms. The pruning technique was more effective on the Enron dataset than on the other two datasets mainly due to two reasons. First, the Enron dataset had more trie nodes due to its large number of distinct keywords in the emails. Thus a query keyword can have more similar prefixes on the trie. Second, the Enron dataset had fewer records, and the inverted lists were relatively shorter. During the list traversal, the NRA algorithm visited fewer records, and its higher score of the top record from the max heap helped us prune more lists.

List Materialization: We evaluated the improvement on sorted access using list materialization for fuzzy type-ahead search. We measured the amount of storage space for storing materialized lists as a percentage of the total size of the inverted lists on the trie. We varied this amount, and measured the average time of finding the top-10 answers using the NRA algorithm. Figure 12 shows the results. We can see that list materialization improved the search performance.

We implemented the different methods for list materialization, namely Random, TopDown, BottomUp, and CostBased as discussed in Section 3.2.2. Figure 13 shows the results. Among the three naive methods, Random gave the best results. The CostBased algorithm outperformed all the naive methods. This is because CostBased selected high-quality nodes for materialization using a cost-based analysis.

Sorted Access + Random Access: We implemented the TA algorithm using the two methods for random access and list pruning for sorted access (described in Section 4). Figure 14 shows the scalability results on the three datasets. The two random-access methods scaled well. Method 2 (probing on trie leaf nodes) outperformed Method 1 (probing on forward lists). This is because for the three data sets, there were many prefixes similar to the partial keyword, and Method 1 needed to consider all similar prefixes for each record on forward lists.

6. RELATED WORK

There are many studies on autocomplete and phrase prediction for user queries [22, 15, 9, 23, 7]. Google instant search was launched to support type-ahead search. It first suggested relevant queries based on user profiles and query logs and then answered the top queries. 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 [13, 20]. They did not study the case where a query has multiple keywords that need list-intersection operations. The search paradigm studied in this paper is different since we support *fuzzy, full-text search* as users type in queries.

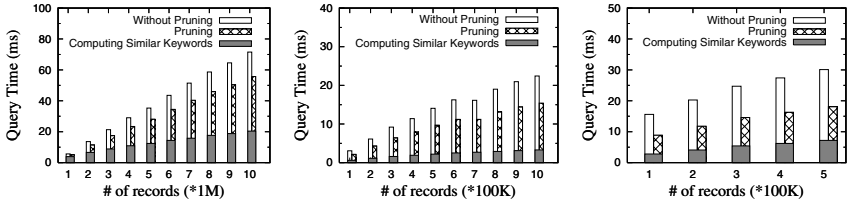
Bast et al. proposed techniques to support type-ahead search in their CompleteSearch systems [2, 3, 1]. Another study [19] is about type-ahead search on relational data graphs. Ji et al. [13] developed algorithms for fuzzy type-ahead search. Our work extends these studies by developing efficient algorithms to support top-k search.

Khoussainova et al. [14] proposed to suggest relevant SQL snippets as users type in SQL queries. Li et al. [18] studied how to use SQLs to support type-ahead search in databases. Feng et al. [8] studied fuzzy search on XML data. There have been many studies on supporting fuzzy search (e.g., [10, 17, 4, 11, 24, 16]). However these algorithms are inefficient for type-ahead search since they have low pruning power for short strings (partial keywords). The experiments in [13, 5] showed that these approaches are not as efficient as trie-based methods for fuzzy type-ahead search. Theobald et al. [25] proposed a heap-based method for query expansion. They used WordNet words and only utilized sorted access. We consider both sorted access and random access.

7. CONCLUSION

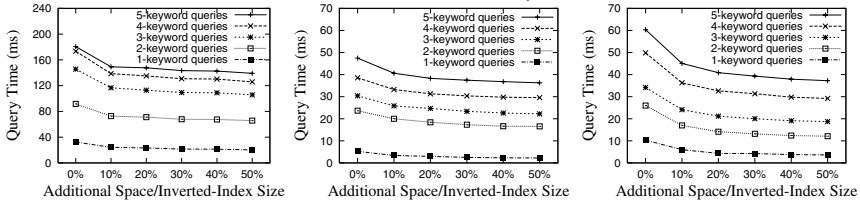
In this paper we studied how to efficiently answer top-k queries in type-ahead search. We focused on an index structure with a trie of keywords in a data set and inverted lists of records on the trie leaf nodes. We studied technical challenges when adopting existing top-k algorithms in the literature: how to efficiently support random access and sorted access on inverted lists? We presented two algorithms for supporting random access, and proposed optimization techniques using list pruning and materialization to support sorted access. Our techniques can be easily extended to support large datasets through data partition. For example, we have built a system to search on 20 million MEDLINE publication records using two machines.

Acknowledgement. The authors have financial interest in Bimaple Technology Inc., a company currently commercializing some of the techniques described in this publication. Chen Li is partially supported by the NIH grant 1R21LM010143-01A1 and the National Natural Science Foundation of China (No. 61129002). Guoliang Li, Jianan Wang, and Jianhua Feng were partly supported by the National Natural Science Foundation of China (No. 61003004), the National Grand Fundamental Research 973 Program of China (No. 2011CB302206), Tsinghua University (No. 20111081073), and the "NExT Research Center" funded by MDA, Singapore (No. WBS:R-252-300-001-490).



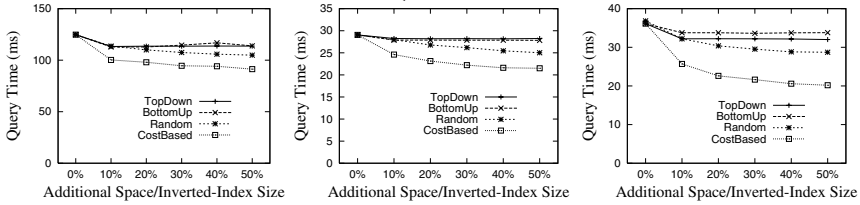
(a) URL (b) DBLP (c) Enron

Figure 11: Fuzzy search using list pruning (similarity threshold $\tau = 0.6$).



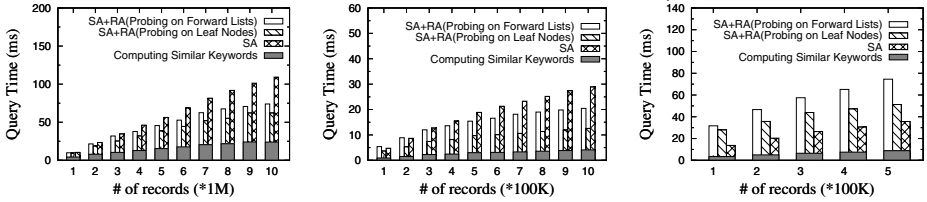
(a) URL (b) DBLP (c) Enron

Figure 12: Fuzzy search using list materialization (sorted access only, with list pruning, threshold $\tau = 0.6$).



(a) URL (b) DBLP (c) Enron

Figure 13: Comparison of different materialization methods (similarity threshold $\tau = 0.6$).



(a) URL (b) DBLP (c) Enron

Figure 14: Fuzzy search with sorted access ("SA") and random access ("RA") (similarity threshold $\tau = 0.6$).

8. REFERENCES

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

[2] H. Bast and I. Weber. Type less, find more: fast autocompletion search with a succinct index. In *SIGIR*, pages 364–371, 2006.

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

[4] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, pages 5–16, 2006.

[5] S. Chaudhuri and R. Kaushik. Extending autocompletion to tolerate errors. In *SIGMOD Conference*, pages 707–718, 2009.

[6] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, 2001.

[7] J. Fan, G. Li, and L. Zhou. Interactive SQL query suggestion: Making databases user-friendly. *ICDE*, pages 351–362, 2011.

[8] J. Feng, and G. Li. Efficient Fuzzy Type-Ahead Search in XML Data. *IEEE TKDE*, 24(5):882–895, 2012.

[9] K. Grabski and T. Scheffer. Sentence completion. In *SIGIR*, pages 433–439, 2004.

[10] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, 2001.

[11] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava. Fast indexes and algorithms for set similarity selection queries. In *ICDE*, pages 267–276, 2008.

[12] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.

[13] S. Ji, G. Li, C. Li, and J. Feng. Efficient interactive fuzzy keyword search. In *WWW*, pages 371–380, 2009.

[14] N. Khoussainova, Y. Kwon, M. Balazinska, and D. Suciu. Snipsuggest: Context-aware autocompletion for sql. *PVLDB*, 4(1):22–33, 2010.

[15] K. Kukich. Techniques for automatically correcting words in text. *ACM Comput. Surv.*, 24(4):377–439, 1992.

[16] H. Lee, R. T. Ng, and K. Shim. Extending q-grams to estimate selectivity of string matching with low edit distance. In *VLDB*, pages 195–206, 2007.

[17] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.

[18] G. Li, J. Feng, and C. Li. Supporting search-as-you-type using sql in databases. *IEEE TKDE*, 2012.

[19] G. Li, S. Ji, C. Li, and J. Feng. Efficient type-ahead search on relational data: a tastier approach. In *SIGMOD Conference*, pages 695–706, 2009.

[20] G. Li, S. Ji, C. Li, and J. Feng. Efficient fuzzy full-text type-ahead search. *VLDB J.*, 20(4):617-640, 2011.

[21] N. Mamoulis, K. H. Cheng, M. L. Yiu, and D. W. Cheung. Efficient aggregation of ranked inputs. In *ICDE*, page 72–83, 2006.

[22] H. Motoda and K. Yoshida. Machine learning techniques to make computers easier to use. *Artif. Intell.*, 103(1-2):295–321, 1998.

[23] A. Nandi and H. V. Jagadish. Effective phrase prediction. In *VLDB*, pages 219–230, 2007.

[24] J. Qin, W. Wang, Y. Lu, C. Xiao, and X. Lin. Efficient exact edit similarity query processing with the asymmetric signature scheme. In *SIGMOD Conference*, pages 1033–1044, 2011.

[25] M. Theobald, R. Schenkel, and G. Weikum. Efficient and self-tuning incremental query expansion for top-k query processing. In *SIGIR*, pages 242–249, 2005.