META: An Efficient Matching-Based Method for Error-Tolerant Autocompletion

Dong Deng† Guoliang Li† He Wen† H. V. Jagadish‡ Jianhua Feng†
†Department of Computer Science, Tsinghua National Laboratory for Information Science and Technology (TNList), Tsinghua University, Beijing, China.
‡Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, USA.
{dd11,wenhe13}@mails.tsinghua.edu.cn;jag@eecs.umich.edu;{liguoliang,fengjh}@tsinghua.edu.cn

ABSTRACT

Autocompletion has been widely adopted in many computing systems because it can instantly provide users with results as users type in queries. Since the typing task is tedious and prone to error, especially on mobile devices, a recent trend is to tolerate errors in autocompletion. Existing error-tolerant autocompletion methods build a trie to index the data, utilize the trie index to compute the trie nodes that are similar to the query, called active nodes, and identify the leaf descendants of active nodes as the results. However these methods have two limitations. First, they involve many redundant computations to identify the active nodes. Second, they do not support top-k queries. To address these problems, we propose a matching-based framework, which computes the answers based on matching characters between queries and data. We design a compact tree index to maintain active nodes in order to avoid the redundant computations. We devise an incremental method to efficiently answer top-k queries. Experimental results on real datasets show that our method outperforms state-of-the-art approaches by 1-2 orders of magnitude.

1. INTRODUCTION

Autocompletion has been widely used in many computing systems, e.g., Unix shells, Google search, email clients, software development tools, desktop search, input methods, and mobile applications (e.g., searching contact list), because it instantly provides users with results as users type in queries and saves their typing efforts. However in many applications, especially for mobile devices that only have virtual keyboards, the typing task is tedious and prone to error. A recent trend is to tolerate errors in autocompletion [7,14,16–19,30]. Edit distance is a widely used metrics to capture typographical errors [1,21] and is supported by many systems, such as `PostgreSQL`[1], `Lucene`[2], `OpenRefine`[3],

[1]`http://www.postgresql.org/docs/8.3/static/fuzzystrmatch.html`
[2]`http://lucene.apache.org/core/4_6_1/suggest/index.html`
[3]`https://github.com/OpenRefine/OpenRefine/wiki/Clustering`

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.
Proceedings of the VLDB Endowment, Vol. 9, No. 10
Copyright 2016 VLDB Endowment 2150-8097/16/06.

Table 1: Dataset $\mathcal{S}$.

id	s_1	s_2	s_3	s_4	s_5	s_6
string	*soho*	*solid*	*solo*	*solve*	*soon*	*throw*

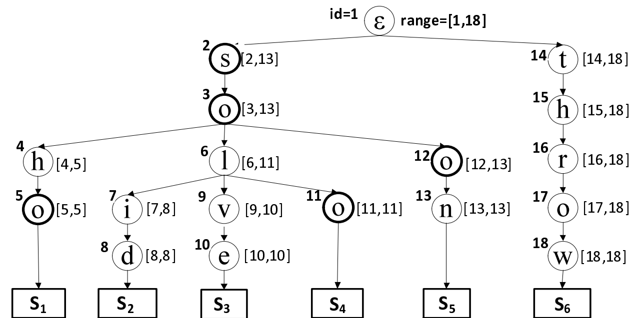


Figure 1: The trie index for strings in Table 1.

and the Unix file comparison tool `diff` [1]. In this paper, we study the error-tolerant autocompletion with edit-distance constraints problem, which, given a query (e.g., an email prefix) and a set of strings (e.g., email addresses), efficiently finds all strings with prefixes **similar** to the query (e.g., email addresses whose prefixes are similar to the query).

Existing methods [7,14,18,30] focus on the threshold-based error-tolerant autocompletion problem, which, given a threshold τ, finds all the strings that have a prefix whose edit distance to the query is within the threshold τ. Note every keystroke from the user will trigger a query and error-tolerant autocompletion needs to compute the answers for every query. Thus the performance is crucial and it is rather challenging to support error-tolerant autocompletion.

To efficiently support error-tolerant autocompletion, existing methods adopt a trie structure to index the strings. Given a query, they compute the trie nodes whose edit distances to the query are within the threshold, called *active nodes*, and the leaf descendants of actives nodes are answers. For example, Figure 1 shows the trie index for strings in Table 1. Suppose the threshold is 2 and the query is "`ssol`". Node n_6 (i.e., the prefix "*sol*") is an active node. Its leaf descendants (i.e., s_2, s_3 and s_4) are answers to the query. Actually, existing methods [7,14] need to access n_7, n_9, and n_{13} three times while our method only accesses them once.

Existing methods have three limitations. First, they cannot meet the high-performance requirement for large datasets. For example, they take more than 1 second per query on a dataset with 4 million strings (see Section 7). Second, they involve redundant computations to compute the active nodes. For example, both n_3 and n_6 are active nodes. They need to check descendants of n_3 and n_6 and obviously the

descendants of n_6 will be checked twice. In practice there are a large number of active nodes and they involve huge redundant computations. Third, it is rather hard to set an appropriate threshold, because a large threshold returns many results while a small threshold leads to few or even no results. For example, the query "parefurnailia" and its top match for human observer "paraphernalia" has an edit distance of 5 which is too large for short words and common errors. An alternative is to return top-k strings that are most similar to the query. However existing methods cannot directly and efficiently support top-k error-tolerant autocompletion queries. This is because the active-node set is dependent on the threshold, and once the threshold changes they need to calculate the active nodes from scratch.

To address these limitations, we propose a matching-based framework for error-tolerant autocompletion, called META, which computes the answers based on matching characters between queries and data. META can efficiently support the threshold-based and top-k queries. To avoid the redundant computations, we design a compact tree structure, which maintains the ancestor-descendant relationship between the active nodes and can guarantee that each trie node is accessed at most once by the active nodes. Moreover, we find that the maximum number of edit errors between a top-k query and its results increases at most 1 with each new keystroke and thus we can incrementally answer top-k queries. To summarize, we make the following contributions.

(1) We propose a matching-based framework to solve the threshold-based and the top-k error-tolerant autocompletion queries (see Sections 3 and 4). To the best of our knowledge, this is the first study on answering the top-k queries.

(2) We design a compact tree structure to maintain the ancestor-descendant relationship between active nodes which can avoid the redundant computations and guarantee each trie node is accessed at most once (see Section 5).

(3) We propose an efficient method to incrementally answer a top-k query by fully using the maximum number of edit errors between the query and its results (see Section 6).

(4) Experimental results on real datasets show that our methods outperform the state-of-the-art approaches by 1-2 orders of magnitude (see Section 7).

2. PRELIMINARY

2.1 Problem Definition

To tolerate errors between a query and a data string, we need to quantify the similarity between two strings. In this paper we utilize the widely-used edit distance to evaluate the string similarity. The edit distance $\mathsf{ED}(q, s)$ between two strings q and s is the minimum number of edit operations needed to transform q to s, where permitted edit operations include deletion, insertion and substitution. For example, $\mathsf{ED}(\mathsf{sso}, \mathsf{solve}) = 4$ as we can transform 'sso' to 'solve' by a deletion (s) and three insertions (l, v, e) .

Let $s[i]$ denote the i-th character of s and $s[i, j]$ denote the substring of s starting from $s[i]$ and ending at $s[j]$. A prefix of string s is a substring of s starting from the first character, i.e., $s[1, j]$ where $0 \le j \le |s|$. Specifically $s[1, 0]$ is an empty string and $s[0] = \phi$. For example, $s[1, 2]$='so' is a prefix of 'solve'. To support error-tolerant autocompletion, we define the prefix edit distance $\mathsf{PED}(q, s)$ from q to s as the minimum edit distance from q to any prefix of s.

Definition 1 (Prefix Edit Distance). *For any two strings q and s,* $\mathsf{PED}(q, s) = \min_{0 \le j \le |s|} \mathsf{ED}(q, s[1, j])$.

For example, $\mathsf{PED}(\mathsf{sso}, \mathsf{solve})$=min($\mathsf{ED}(\mathsf{sso}, \phi)$, $\mathsf{ED}(\mathsf{sso}, \mathsf{s})$, $\mathsf{ED}(\mathsf{sso}, \mathsf{so})$, $\mathsf{ED}(\mathsf{sso}, \mathsf{sol})$, $\mathsf{ED}(\mathsf{sso}, \mathsf{solv})$, $\mathsf{ED}(\mathsf{sso}, \mathsf{solve})$) = $\mathsf{ED}(\mathsf{sso}, \mathsf{so})$=1. We aim to solve the threshold-based and top-k error-tolerant autocompletions as formulated below.

Definition 2. *Given a set of strings $\mathcal{S}$, a query string q, and a threshold τ, the threshold-based error-tolerant autocompletion finds all $s \in \mathcal{S}$ such that $\mathsf{PED}(q, s) \le \tau$.*

Definition 3. *Given a set of strings $\mathcal{S}$, a query string q, and an integer k ($|\mathcal{S}| \ge k$), the top-k error-tolerant autocompletion finds a result set $\mathcal{R} \subseteq \mathcal{S}$ where $|\mathcal{R}| = k$ and $\forall s_1 \in \mathcal{R}, \forall s_2 \in \mathcal{S} - \mathcal{R}$, $\mathsf{PED}(q, s_1) \le \mathsf{PED}(q, s_2)$.*

In line with existing methods [7,14,18,30], we also assume the user types in queries letter by letter[4] and use q_i to denote the query $q[1, i]$. For example, consider the dataset $\mathcal{S}$ in Table 1 and suppose the threshold is $\tau = 2$. For the continuous queries q_1='s', q_2='ss', q_3='sso', and q_4='ssol', the results are respectively $\{s_1,s_2,s_3,s_4,s_5,s_6\}$, $\{s_1,s_2,s_3,s_4,s_5,s_6\}$, $\{s_1,s_2,s_3,s_4,s_5\}$, and $\{s_1,s_2,s_3,s_4,s_5\}$.

For top-k queries, suppose $k = 3$. For the top-k queries q_1='s', q_2='ss', q_3='sso', and q_4='ssol', the results are $\{s_1,s_2,s_3\}$, $\{s_1,s_2,s_3\}$, $\{s_1,s_2,s_3\}$, and $\{s_2,s_3,s_4\}$.

2.2 Related Works

Threshold-Based Error-Tolerant Autocompletion: Ji et al. [14] and Chaudhuri et al. [7] proposed two similar methods, which built a trie index for the dataset, computed an active node set, and utilized the active nodes to answer threshold-based queries. Li et al. [18] improved their works by maintaining the pivotal active node set, which is a subset of the active node set. Xiao et al. [30] proposed a neighborhood generation based method, which generated $\mathcal{O}(l^\tau)$ deletion neighborhoods for each data string with length l and threshold τ, and indexed them into a trie. Obviously this method had a huge index, which is $\mathcal{O}(l^\tau)$ times larger than ours. These methods keep an active node set $\mathcal{A}_i$ for each query q_i. When the user types in another letter and submits the query q_{i+1}, they calculate the active node set $\mathcal{A}_{i+1}$ based on $\mathcal{A}_i$. However, if the threshold changes, they need to calculate the active node set $\mathcal{A}'_{i+1}$ from scratch, i.e. calculate all $\mathcal{A}'_1$, $\mathcal{A}'_2$, ..., $\mathcal{A}'_{i+1}$. Thus they cannot efficiently answer the top-k queries. Our method has two significant differences from existing works. First, our method can support top-k queries. Second, our method can avoid redundant computations in computing active nodes and improve the performance. In addition, some recent work studied the location-based instant search on spatial databases [13,23,33], which is orthogonal to our problem.

Query Auto-Completion: There are a vast amount of studies [3,4,10,20,29] on Query Auto-Completion (QAC) which is different from our Error-Tolerant Autocompletion (ETA) problem. Typically QAC has two steps: (1) getting all the strings with the prefix **same** as the query; (2) ranking these strings to improve the accuracy. Existing work on QAC focus on the second step by using the historical data, e.g., search log and temporal information. Our paper studies the ETA problem with edit-distance constraints which is different from QAC. ETA is a general technique and can be used to improve the recall of QAC. Considering a query 'uun', the first step of QAC may return an empty list as there is no data string starting with 'uun' and the second step will be

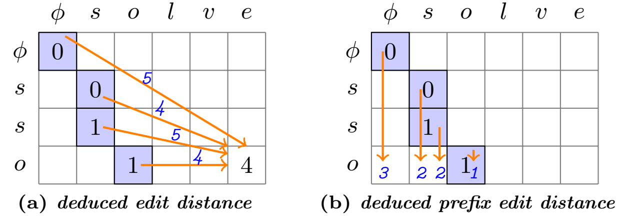
[4]For a copy-paste query, we answer it from scratch; for deleting a character, we use the result of its previous query to answer it.

skipped. With ETA, we can return those strings with prefixes similar to the query, e.g., 'unit' and 'universal', and then the second step can rank them. SRCH2[5] and Cetindil et al. [4] studied fuzzy top-k autocompletion queries: given a query, it finds the similar prefixes whose edit distance to the query is within a pre-defined threshold (using the method in [14]). Then it ranks the answer based on its relevance to the query, which is defined based on various pieces of information such as the frequencies of query keywords in the record, and co-occurrence of some query keywords as a phrase in the record. Duan et al. [10] proposed a Markov n-gram transformation model, where the edit distance model is a special case of the transformation model. However, it can only correct a misspelled query to previously observed queries which are not provided in our problem. There are also lots of studies on query recommendation which generate query reformulations to assist users [12,24,25]. However they mainly focus on improving the quality of recommendations while we aim to improve the efficiency.

String Similarity Search and Join: There have been many studies on string similarity search [2,5,8,9,22,32] and string similarity joins [6,15,27,28,31]. Given a query and a set of objects, the string similarity search (SSS) finds all similar objects to the query. Given two sets of objects, the string similarity joins (SSJ) compute the similar pairs from the two sets. We can extend the techniques of the SSS problem to address the ETA problem as follows. We first generate all the prefixes of each data string. Then we perform the SSS techniques to find the similar answers of the query from all the prefixes (called candidate prefixes). For the threshold-based query, the strings containing the candidate prefixes are the answers of the ETA query. For the top-k query, we incrementally increase the thresholds until finding top-k answers. However, the SSS techniques cannot efficiently support the ETA problem [7,14,18,30], because (i) they generate huge number of prefixes and (ii) cannot share the computations between the continuous queries typed letter by letter.

Discussion. (1) Our proposed techniques can support the ETA query with multiple words (e.g., the person name). We first split them to single words and add them to the trie index. Then for a multiple-word query, we return the intersection of the result sets of each query word as the results. Moreover, the techniques in [14,18] to support multiple-word query using the single-word error-tolerant autocompletion methods also apply to META. Our method can be integrated into them to improve the performance. (2) Edit distance can work with the other scoring functions (such as TF/IDF, frequency, keyboard edit distance, and Soundex). There are two possible ways to combine them. Firstly, we can aggregate edit distance with other functions using a linear combination, e.g., combining edit distance with TF/IDF. Then we can use the TA algorithm [11] to compute the answers. The TA algorithm takes as input several ranked score lists, e.g., the list of strings sorted by edit distance to the query string and the list of strings sorted by TF/IDF. Note that the second list can be gotten offline and we need to compute the first list online. Obviously our method can be used to get the first list, i.e., top-k strings. (2) We can use our method as the first step to generate k data strings with the smallest prefix edit distance to the query, and then re-rank these data strings by the other scoring functions.

[5] http://www.srch2.com



(a) *deduced edit distance* (b) *deduced prefix edit distance*

Figure 2: The matchings and deduced (prefix) edit distance between q ='sso' and s ='solve'.

3. PREFIX EDIT DISTANCE CALCULATION

By considering the last matching characters of two strings, we design a dynamic-programming algorithm to calculate their edit distance. More specifically, suppose $q[i] = s[j]$ is the last matching in a transformation from q to s, there are at least $\mathsf{ED}(q[1,i], s[1,j]) + max(|q|-i, |s|-j)$ edit operations in this transformation. Thus given two strings q and s, we can enumerate every matching $q[i] = s[j]$ for $1 \le i \le |q|, 1 \le j \le |s|$ and the minimum $\mathsf{ED}(q[1,i], s[1,j]) + max(|q| - i, |s| - j)$ is the edit distance between q and s. For example, as shown in Figure 2(a), there are four matchings (blue cells) between q and s. The minimum $\mathsf{ED}(q[1,i], s[1,j]) + max(|q|-i, |s|-j)$ is 4 when $q[i = 1] = s[j = 1] =$'s'. Thus $\mathsf{ED}(q, s) = 0 + 4 = 4$. We introduce how to utilize this idea to compute (prefix) edit distance in Section 3.1. We discuss how to compute the matching characters in Section 3.2.

3.1 Deducing Edit Distance by Matching Set

Matching-Based Edit Distance Calculation: For ease of presentation, we first give two concepts.

DEFINITION 4. *Given two strings q and s, a matching is a triple $\mathsf{m} = \langle i, j, ed\rangle$ where $q[i] = s[j]$ and $ed = \mathsf{ED}(q[1, i], s[1, j])$.*

For example, as shown in Figure 2(a), as $q[3] = s[2]$ and ED('sso','so') = 1, $\langle 3, 2, 1\rangle$ is a matching, so are all the cells filled in blue. All the matchings between two strings q and s compose their matching set $\mathcal{M}(q, s)$. For example, as shown in Figure 2(a), the matching set of q ='sso' and s ='solve' is $\mathcal{M}(q, s) = \{\langle 0, 0, 0\rangle, \langle 1, 1, 0\rangle, \langle 2, 1, 1\rangle, \langle 3, 2, 1\rangle\}$. Note for any q and s, $\mathcal{M}(q_0, s)$=$\{\langle 0, 0, 0\rangle\}$ as only j=0 satisfies $s[j]$=$q[0]$.

Given a matching $\mathsf{m} = \langle i, j, ed\rangle$ of two strings q and s, the edit distance between q and s is not larger than $ed + max(|q| - i, |s| - j)$, which is called *the deduced edit distance* from q to s based on the matching m and defined as below.

DEFINITION 5 (DEDUCED EDIT DISTANCE). *Given two strings q and s, the deduced edit distance from q to s based on a matching $\mathsf{m} = \langle i, j, ed\rangle$ is $\mathsf{m}_{(|q|,|s|)} = ed + max(|q|-i, |s|-j)$.*

For example, as shown in Figure 2(a), the deduced edit distance of q and s based on the matching $\mathsf{m} = \langle 3, 2, 1\rangle$ is $\mathsf{m}_{(3,5)} = 1 + max(4-3, 5-2) = 4$. The deduced edit distance based on the other matchings are also shown in the figure.

Based on the two concepts we develop a matching-based method to compute edit distance. Given two strings q and s, we enumerate every matching in their matching set and the minimum deduced edit distance from q to s based on these matchings is exactly $\mathsf{ED}(q, s)$ as stated in Lemma 1.

LEMMA 1. *For any q and s, $\mathsf{ED}(q, s) = \min_{\mathsf{m} \in \mathcal{M}(q,s)} \mathsf{m}_{(|q|,|s|)}$.*

We omit the formal proof of all lemmas and theorms due to the space limits. Based on Lemma 1 we can compute the

Table 2: A running example of the matching set calculation ($q =$'sso', $s =$'solve').

i and q_i	$i=1, q_1 = \mathsf{s}$	$i=2, q_2 = \mathsf{ss}$		$i=3, q_3 = \mathsf{sso}$		
$\mathsf{m}' \in \mathcal{M}(q_{i-1}, s)$	$\langle 0,0,0\rangle$	$\langle 0,0,0\rangle$	$\langle 1,1,0\rangle$	$\langle 0,0,0\rangle$	$\langle 1,1,0\rangle$	$\langle 2,1,1\rangle$
entry $\langle j, \mathsf{m}'_{(i-1,j-1)}\rangle$ in $\mathcal{H}$	$\langle 1,0\rangle$	$\langle 1,1\rangle$		~~$\langle 2,2\rangle$~~	$\langle 2,1\rangle$	~~$\langle 2,1\rangle$~~

edit distance based on the matching set. We show how to calculate the matching set $\mathcal{M}(q,s)$ later in Section 3.2.

Matching-Based Prefix Edit Distance Calculation: The basic idea of the matching-based prefix edit distance calculation is illustrated in Figure 2(b). Given a query q and a string s, to calculate their prefix edit distance $\mathsf{PED}(q,s)$ we need to find a transformation from q to a prefix of s with minimum number of edit operations. Suppose $q[i] = s[j]$ is the last matching in this transformation, we have $\mathsf{PED}(q,s) = \mathsf{ED}(q[1,i], s[1,j]) + (|q| - i)$, as the prefix edit distance is exactly the number of edit operations in this transformation. Thus we can enumerate every matching $q[i] = s[j]$ between a query q and a string s and the minimum of $\mathsf{ED}(q[1,i], s[1,j]) + (|q|-i)$ is the prefix edit distance between q and s. For example, as shown in Figure 2(b), the minimum of $\mathsf{ED}(q[1,i], s[1,j]) + (|q|-i)$ is 1 when $q[i=3] = s[j=2] =$'o' and thus $\mathsf{PED}(q,s) = 1 + 0 = 1$. Next we give a concept and formalize our idea.

Definition 6 (Deduced Prefix Edit Distance). *For any two strings q and s, the deduced prefix edit distance from q to s based a matching $\mathsf{m} = \langle i,j,ed\rangle$ is $\mathsf{m}_{|q|} = ed + (|q| - i)$.*

For example, as shown in Figure 2(b), the deduced prefix edit distance between q and s based on their matching $\mathsf{m} = \langle 3,2,1\rangle$ is $\mathsf{m}_3 = 1 + (3-3) = 1$. The deduced prefix edit distance based on the other matchings are also shown in the figure. Based on this definition we propose a matching-based method to compute prefix edit distance. Given two strings q and s we enumerate every matching in their matching set and the minimum deduced prefix edit distance based on these matchings is exactly $\mathsf{PED}(q,s)$ as stated in Lemma 2.

Lemma 2. *For any q and s, $\mathsf{PED}(q,s) = \min_{\mathsf{m} \in \mathcal{M}(q,s)} \mathsf{m}_{|q|}$.*

Based on Lemma 2, we can compute the prefix edit distance based on the matching set. Next we discuss how to calculate the matching set $\mathcal{M}(q,s)$.

3.2 Calculating the Matching Set

As $\mathcal{M}(q_{i-1}, s) \subseteq \mathcal{M}(q_i, s)$, we can calculate $\mathcal{M}(q_i,s)$ in an incremental way, i.e., calculate the matchings $\langle i,j,ed\rangle$ in $\mathcal{M}(q_i,s) - \mathcal{M}(q_{i-1},s)$ for each $1 \le i \le |q|$. More specifically, we first initialize $\mathcal{M}(q_0,s)$ as $\{\langle 0,0,0\rangle\}$. Then for each $1 \le i \le |q|$, we find all the $1 \le j \le |s|$ s.t. $s[j] = q[i]$ and calculate $ed = \mathsf{ED}(q[1,i], s[1,j])$ using $\mathcal{M}(q_{i-1},s)$. We have an observation that if $q[i] = s[j]$, $\mathsf{ED}(q[1,i], s[1,j])$ is exactly the minimum of $\mathsf{m}_{(i-1,j-1)}$ where $\mathsf{m} \in \mathcal{M}(q_{i-1}, s[1,j-1])$. This is because on the one hand, Ukkonen [26] proved when $q[i] = s[j]$, $\mathsf{ED}(q[1,i], s[1,j]) = \mathsf{ED}(q[1,i-1], s[1,j-1])$ and on the other hand, based on Lemma 1, $\mathsf{ED}(q[1,i-1], s[1,j-1])$ is the minimum of $\mathsf{m}_{(i-1,j-1)}$ where $\mathsf{m} \in \mathcal{M}(q_{i-1}, s[1,j-1])$. Thus $\mathsf{ED}(q[1,i], s[1,j]) = \min_{\mathsf{m} \in \mathcal{M}(q_{i-1}, s[1,j-1])} \mathsf{m}_{(i-1,j-1)}$ as stated in Lemma 3.

Lemma 3. *Given two strings q and s, for any $q[i] = s[j]$ we have $\mathsf{ED}(q[1,i], s[1,j]) = \min_{\mathsf{m} \in \mathcal{M}(q_{i-1}, s[1,j-1])} \mathsf{m}_{(i-1,j-1)}$.*

In addition, as $\mathcal{M}(q_{i-1}, s[1,j-1])$ is a subset of $\mathcal{M}(q_{i-1},s)$, we can enumerate every $\mathsf{m}' = \langle i',j',ed'\rangle$ in $\mathcal{M}(q_{i-1},s)$ s.t. $j' < j$ (which indicates $\mathsf{m}' \in \mathcal{M}(q_{i-1}, s[1,j-1])$) and the minimum of $\mathsf{m}'_{(i-1,j-1)}$ is exactly the minimum of $\mathsf{m}_{(i-1,j-1)}$ where $\mathsf{m} \in \mathcal{M}(q_{i-1}, s[1,j-1])$. In this way we can get ed and the new matching $\langle i,j,ed\rangle$ in $\mathcal{M}(q_i,s) - \mathcal{M}(q_{i-1},s)$. All these new matchings and all those matchings in $\mathcal{M}(q_{i-1},s)$ forms $\mathcal{M}(q_i,s)$. Finally we can get $\mathcal{M}(q,s)$.

Algorithm 1: MatchingSetCalculation

```
   Input: q: a query string;   s: a data string.
   Output: M(q, s): the matching set of q and s.
1  M(q_0, s) = {<0,0,0>};
2  foreach q_i where 1 <= i <= |q| do
3      H = φ; // minimum deduced edit distance
4      foreach m' = <i', j', ed'> ∈ M(q_{i-1}, s) do
5          foreach j > j' s.t. q[i] = s[j] do
6              if H[j] > m'_(i-1,j-1) then H[j] = m'_(i-1,j-1)
7      foreach entry <j, ed> in H do
8          add the matching <i, j, ed> to M(q_i, s);
9      add all the matchings in M(q_{i-1}, s) to M(q_i, s);
10 output M(q, s);
```

The pseudo code of the matching set calculation is illustrated in Algorithm 1. It takes two strings q and s as input and outputs their matching set. It first initializes $\mathcal{M}(q_0,s)$ as $\{\langle 0,0,0\rangle\}$ (Line 1). Then for each $1 \le i \le |q|$, it initializes a hash map $\mathcal{H}$ to keep the minimum deduced edit distance (Lines 2 to 3). For each matching $\mathsf{m}' = \langle i',j',ed'\rangle \in \mathcal{M}(q_{i-1},s)$, it finds all $j > j'$ s.t. $q[i] = s[j]$, calculates the deduced edit distance $\mathsf{m}'_{(i-1,j-1)}$, and updates $\mathcal{H}[j]$ if $\mathsf{m}'_{(i-1,j-1)}$ is smaller (Lines 4 to 6). Then for each entry $\langle j, ed\rangle$ in the hash map $\mathcal{H}$, it adds the matching $\langle i,j,ed\rangle$ to $\mathcal{M}(q_i,s)$. (Lines 7 to 8). In addition, it adds all the matchings in $\mathcal{M}(q_{i-1},s)$ to $\mathcal{M}(q_i,s)$ (Line 9). Finally it outputs the matching set $\mathcal{M}(q,s)$ (Line 10).

Example 1. *Table 2 shows a running example of the matching set calculation. Note the entries in $\mathcal{H}$ with deletions are replaced by others. We first set $\mathcal{M}(q_0,s) = \{\langle 0,0,0\rangle\}$. For $i=1$, for the matching $\mathsf{m}' = \langle 0,0,0\rangle$, we have $j=1$ s.t. $q[1] = s[1]$. Thus we set $\mathcal{H}[1] = \mathsf{m}'_{(0,0)} = 0$. We traverse $\mathcal{H}$ and add $\langle 1,1,0\rangle$ to $\mathcal{M}(q_1,s)$. We also add $\langle 0,0,0\rangle \in \mathcal{M}(q_0,s)$ to $\mathcal{M}(q_1,s)$. Thus $\mathcal{M}(q_1,s) = \{\langle 0,0,0\rangle, \langle 1,1,0\rangle\}$. For $i=2$, for $\mathsf{m}' = \langle 0,0,0\rangle$ we have $j=1$ s.t. $q[2] = s[1]$. Thus we set $\mathcal{H}[1] = \mathsf{m}'_{(1,0)} = 1$. For $\mathsf{m}' = \langle 1, j'=1, 0\rangle$, as $q[2] \neq s[j]$ for any $j > j' = 1$, we do nothing. We traverse $\mathcal{H}$ and add $\langle 2,1,1\rangle$ to $\mathcal{M}(q_2,s)$. We also add the matchings in $\mathcal{M}(q_1,s)$ to it and have $\mathcal{M}(q_2,s) = \{\langle 0,0,0\rangle, \langle 1,1,0\rangle, \langle 2,1,1\rangle\}$. For $i=3$, for $\mathsf{m}' = \langle 0,0,0\rangle$, we set $\mathcal{H}[2] = \mathsf{m}'_{(2,1)} = 2$. For $\mathsf{m}' = \langle 1,1,0\rangle$, as $\mathsf{m}'_{(2,1)} = 1 < \mathcal{H}[2] = 2$, we update $\mathcal{H}[2]$ as 1. For $\mathsf{m}' = \langle 2,1,1\rangle$, as $\mathsf{m}'_{(2,1)} = 1$ is not smaller than $\mathcal{H}[2] = 1$, we do not update $\mathcal{H}[2]$. We traverse $\mathcal{H}$ and add $\langle 3,2,1\rangle$ to $\mathcal{M}(q_3,s)$. We also add the matchings in $\mathcal{M}(q_2,s)$ to it and have $\mathcal{M}(q_3,s) = \{\langle 0,0,0\rangle, \langle 1,1,0\rangle, \langle 2,1,1\rangle, \langle 3,2,1\rangle\}$. Note based on Lemmas 1 and 2 we have $\mathsf{ED}(q_3,s) = \langle 1,1,0\rangle_{(3,5)} = 4$ and $\mathsf{PED}(q_3,s) = \langle 3,2,1\rangle_3 = 1$.*

4. THE MATCHING-BASED FRAMEWORK

In this section, we calculate the prefix edit distance between a query and a set of data strings. We first design a matching-based framework for threshold-based queries and then extend it to support top-k queries in Section 6.

Table 3: A running example of the matching-based framework ($\tau = 2, q=$‘sso’, $\mathcal{T}$ in Figure 1).

i and query q_i	$i = 1, q_1 = \mathsf{s}$	$i = 2, q_2 = \mathsf{ss}$		$i = 3, q_3 = \mathsf{sso}$		
$\mathsf{m}' \in \mathcal{A}(q_{i-1}, \mathcal{T})$	$\langle 0, n_1, 0\rangle$	$\langle 0, n_1, 0\rangle$	$\langle 1, n_2, 0\rangle$	$\langle 0, n_1, 0\rangle$	$\langle 2, n_2, 1\rangle$	$\langle 1, n_2, 0\rangle$
$\langle n, \mathsf{m}'_{(i-1,\lvert n\rvert-1)}\rangle \in \mathcal{H}$	$\langle n_2, 0\rangle$	$\langle n_2, 1\rangle$		~~$\langle n_3, 2\rangle$~~ ~~$\langle n_{12}, 2\rangle$~~	$\langle n_3, 1\rangle$ ~~$\langle n_{12}, 2\rangle$~~	~~$\langle n_3, 1\rangle$~~ $\langle n_{12}, 1\rangle$ $\langle n_5, 2\rangle$ $\langle n_{11}, 2\rangle$

Indexing. We index all the data strings into a trie. We traverse the trie in pre-order and assign each node an *id* starting from 1. For each leaf node, we also assign it with the corresponding string id. For example, Figure 1 shows the trie index for the dataset $\mathcal{S}$ in Table 1. Each node n in $\mathcal{T}$ contains a label $n.char$, an id $n.id$, a depth $n.depth = |n|$ and a range $n.range = [n.lo, n.up]$ where $n.lo$ and $n.up$ are respectively the smallest and largest node *id* in the subtree rooted at n. Each node n corresponds to a prefix which is composed of the characters from the *root* to n. All the strings in the subtree rooted at n share this common prefix. For simplicity, we interchangeably use node n with its corresponding prefix. We also use $n.parent$ to denote the parent node of n. For each keystroke x, to efficiently find the node n where $n.char = x$, we build a two-dimensional inverted indexes $\mathcal{I}$ for the trie nodes where the inverted list $\mathcal{I}[depth][x]$ contains all the nodes n in $\mathcal{T}$ s.t. $|n| = depth$ and $n.char = x$. The nodes in the inverted list are sorted by their id for ease of binary search.

Querying. Since each trie node corresponds to a prefix, the definitions of the matching and deduced (prefix) edit distance can be intuitively extended to trie nodes and we use them interchangeably. For example, consider the trie in Figure 1 and suppose $q =$‘ss’. $\mathsf{m} = \langle 2, n_2, 1\rangle$ is a matching as $n_2.char = q[2] =$‘s’ and $\mathsf{ED}(q[1, 2], n_2) = 1$. The deduced edit distance from q to n_3 based on m is $\mathsf{m}_{(2,|n_3|=2)} = 1 + max(2-2, 2-1) = 2$. The deduced prefix edit distance of q based on m is $\mathsf{m}_{|q|=2} = 1 + (2-2) = 1$. Next, we introduce a concept and give the basic idea of querying.

Definition 7 (Active Matching and Active Node). *Given a query q and a threshold τ, $\mathsf{m} = \langle i, n, ed\rangle$ is an active matching of q and n is an active node of q iff $\mathsf{m}_{|q|} \le \tau$.*

Consider the example above and suppose $\tau = 2$, we have $\mathsf{m} = \langle 2, n_2, 1\rangle$ is an active matching of q and n_2 is an active node of q as $\mathsf{m}_{|q|} = 1 \le \tau$. All the active matchings between a query q and a trie $\mathcal{T}$ compose their active matching set $\mathcal{A}(q, \mathcal{T})$. Following the example above we have $\mathcal{A}(q, \mathcal{T}) = \{\langle 0, n_1, 0\rangle\langle 1, n_2, 0\rangle\langle 2, n_2, 1\rangle\}$. Note $\mathcal{A}(q_0, \mathcal{T})=\{\langle 0, \mathcal{T}.root, 0\rangle\}$.

Next we give the basic idea of our matching-based framework. We have an observation that given a query q and a threshold τ, for any string $s \in \mathcal{S}$, if $\mathsf{PED}(q, s) \le \tau$ there must exist an active matching $\langle i, n, ed\rangle$ of q s.t. s is a leaf descendant of n. This is because if $\mathsf{PED}(q, s) \le \tau$, based on Lemma 2 there exists a matching $\mathsf{m} = \langle i, j, ed\rangle \in \mathcal{M}(q, s)$ s.t. $\mathsf{m}_{|q|} = ed + (|q|-i) \le \tau$. Suppose n is the corresponding trie node of the prefix $s[1, j]$, we have $q[i] = s[j] = n.char$ and $ed = \mathsf{ED}(q[1, i], s[1, j]) = \mathsf{ED}(q[1, i], n)$. Based on Definition 7, $\langle i, n, ed\rangle$ is an active matching of q as $\langle i, n, ed\rangle_{|q|} = ed + (|q| - i) \le \tau$. Thus to answer a query, we only need to find its active matching set. Next we show how to incrementally get $\mathcal{A}(q_i, \mathcal{T})$ based on $\mathcal{A}(q_{i-1}, \mathcal{T})$ for each $1 \le i \le |q|$.

For each query q_i, there are two kinds of active matchings $\mathsf{m}'' = \langle i'', n'', ed''\rangle$ in $\mathcal{A}(q_i, \mathcal{T})$. Those with $i'' < i$ and those with $i'' = i$. Note based on Definition 7, $\mathsf{m}''_i \le \tau$. For the first kind, m'' is also an active matching of q_{i-1} as $\mathsf{m}''_{i-1} = \mathsf{m}''_i - 1 \le \tau - 1 \le \tau$. Thus we can get all the first kind of active matchings from $\mathcal{A}(q_{i-1}, \mathcal{T})$. For the second kind, we have $\mathsf{m}''_i = ed'' + (i - i'') = ed'' \le \tau$. To get all this kind of active matchings, we need to find all the nodes n'' s.t.

Algorithm 2: MatchingBasedFramework

Input: $\mathcal{T}$: a trie; τ: a threshold; q: a continuous query;
Output: $\mathcal{R}_i=\{s \in \mathcal{S} \big| \mathsf{PED}(q_i, s) \le \tau\}$ for each $1 \le i \le |q|$;

1 $\mathcal{A}(q_0, \mathcal{T}) = \{\langle 0, \mathcal{T}.root, 0\rangle\}$;
2 **foreach** *query* q_i *where* $1 \le i \le |q|$ **do**
3 $\quad \mathcal{H} = \phi$; `// minimum deduced edit distance`
4 $\quad$ **foreach** $\mathsf{m}' = \langle i', n', ed'\rangle \in \mathcal{A}(q_{i-1}, \mathcal{T})$ **do**
5 $\quad\quad$ **foreach** *descendant node* n *of* n' *where* $n.char = q[i]$ *and* $\mathsf{m}'_{(i-1,|n|-1)} \le \tau$ **do**
6 $\quad\quad\quad$ **if** $\mathcal{H}[n] > \mathsf{m}'_{(i-1,|n|-1)}$ **then**
7 $\quad\quad\quad\quad \mathcal{H}[n] = \mathsf{m}'_{(i-1,|n|-1)}$;
8 $\quad$ **foreach** *entry* $\langle n, ed\rangle \in \mathcal{H}$ **do**
9 $\quad\quad$ add the active matching $\langle i, n, ed\rangle$ to $\mathcal{A}(q_i, \mathcal{T})$;
10 $\quad$ **foreach** $\mathsf{m}' = \langle i', n', ed'\rangle \in \mathcal{A}(q_{i-1}, \mathcal{T})$ **do**
11 $\quad\quad$ **if** $\mathsf{m}'_i \le \tau$ **then** add m' to $\mathcal{A}(q_i, \mathcal{T})$;
12 $\quad$ **foreach** $\langle i', n', ed'\rangle \in \mathcal{A}(q_i, \mathcal{T})$ **do**
13 $\quad\quad$ add all the strings on the leaves of n' to $\mathcal{R}_i$;
14 $\quad$ output $\mathcal{R}_i$;

$n''.char = q[i]$ and $\mathsf{ED}(q_i, n'') \le \tau$, and calculate the value $ed'' = \mathsf{ED}(q_i, n'')$. Based on Lemma 3, for any node n'' s.t. $n''.char = q[i]$, $\mathsf{ED}(q_i, n'')$ is the minimum of $\mathsf{m}_{(i-1,|n''|-1)}$ where $\mathsf{m} \in \mathcal{M}(q_{i-1}, n''.parent)$. To satisfy $\mathsf{ED}(q_i, n'') \le \tau$, we require $\mathsf{m}_{(i-1,|n''|-1)} \le \tau$. As $\mathsf{m}_{i-1} \le \mathsf{m}_{(i-1,|n''|-1)} \le \tau$, m is an active matching of q_{i-1}, i.e., $\mathsf{m} \in \mathcal{A}(q_{i-1}, \mathcal{T})$. Thus for each n'' s.t. $q[i] = n''.char$, we can enumerate every $\mathsf{m}' = \langle i', n', ed'\rangle \in \mathcal{A}(q_{i-1}, \mathcal{T})$ where n' is an ancestor of n'' (which indicates $\mathsf{m}' \in \mathcal{M}(q_{i-1}, n''.parent)$) and $\mathsf{m}'_{(i-1,|n''|-1)} \le \tau$, and the minimum of $\mathsf{m}'_{(i-1,|n''|-1)}$ is exactly $ed'' = \mathsf{ED}(q_i, n'')$ if $\mathsf{ED}(q_i, n'') \le \tau$. In this way we can get ed'' and all the second kind of active matchings.

The pseudo-code of the matching-based framework is shown in Algorithm 2. It takes a trie $\mathcal{T}$, a threshold τ and a query string q as input and outputs the result set $\mathcal{R}_i$ for each query q_i where $1 \le i \le |q|$. It first initializes $\mathcal{A}(q_0, \mathcal{T})$ as $\{\langle 0, \mathcal{T}.root, 0\rangle\}$ (Line 1). Then for each query q_i, it first initializes a hash map $\mathcal{H} = \phi$ to keep the minimum deduced edit distance (Line 3). Then, for each matching $\mathsf{m}' = \langle i', n', ed'\rangle \in \mathcal{A}(q_{i-1}, \mathcal{T})$, it finds all the descendant nodes n of n' s.t. $n.char = q[i]$ and $\mathsf{m}'_{(i-1,|n|-1)} \le \tau$ and uses the deduced edit distances $\mathsf{m}'_{(i-1,|n|-1)}$ to update $\mathcal{H}[n]$ (Line 4 to 7). Next for each entry $\langle n, ed\rangle$ in $\mathcal{H}$, it adds the active matching $\langle i, n, ed\rangle$, which is the second kind as described above, to $\mathcal{A}(q_i, \mathcal{T})$ (Lines 8 to 9). For each $\mathsf{m}' \in \mathcal{A}(q_{i-1}, \mathcal{T})$, if $\mathsf{m}'_i \le \tau$, it adds the active matching m', which is the first kind, to $\mathcal{A}(q_i, \mathcal{T})$ (Lines 10 to 11). Finally, it adds the leaves of the active nodes of the active matchings in $\mathcal{A}(q_i, \mathcal{T})$ to $\mathcal{R}_i$ and outputs $\mathcal{R}_i$ (Lines 12 to 14).

Example 2. *Table 3 shows a running example of the matching based framework. For $i = 3$ and $q_3 =$‘sso’, initially we have $\mathcal{A}(q_2, \mathcal{T}) = \{\langle 0, n_1, 0\rangle, \langle 2, n_2, 1\rangle, \langle 1, n_2, 0\rangle\}$. For $\mathsf{m}' = \langle 0, n_1, 0\rangle$, we have the descendants n_3 and n_{12} of n_1 s.t. $q[3] = n_3.char$ and $q[3] = n_{12}.char$ and $\mathsf{m}'_{(3-1,|n_3|-1)} = 2 \le \tau$ and $\mathsf{m}'_{(3-1,|n_{12}|-1)} = 2 \le \tau$. Thus we set $\mathcal{H}[n_3] = 2$ and $\mathcal{H}[n_{12}] = 2$. For $\mathsf{m}' = \langle 2, n_2, 1\rangle$, the descendants n_3 and n_{12} of n_2 have the same label as $q[3]$ and $\mathsf{m}'_{(2,1)} = 1 < \mathcal{H}[n_3]$*

and $\mathsf{m}'_{(2,2)} = 2 \geq \mathcal{H}[n_{12}]$. *Thus we only update* $\mathcal{H}[n_3] = 1$. *Note the descendants* n_5 *and* n_{11} *of* n_2 *also have the same label as* $q[3]$. *However we skip them as* $\mathsf{m}'_{(2,3)} = 3 > \tau$. *For* $\mathsf{m}' = \langle 1, n_2, 0 \rangle$, *the descendants* n_3, n_{12}, n_5 *and* n_{11} *of* n_2 *have the same label as* $q[3]$ *and* $\mathsf{m}'_{(2,1)} = 1$, $\mathsf{m}'_{(2,2)} = 1$, $\mathsf{m}'_{(2,3)} = 2$ *and* $\mathsf{m}'_{(2,3)} = 2$. *Thus we update* $\mathcal{H}[n_{12}] = 1$ *and set* $\mathcal{H}[n_5] = 2$ *and* $\mathcal{H}[n_{11}] = 2$. *Then we traverse* $\mathcal{H}$ *and add* $\langle 3, n_3, 1 \rangle$, $\langle 3, n_{12}, 1 \rangle$, $\langle 3, n_5, 2 \rangle$ *and* $\langle 3, n_{11}, 2 \rangle$ *to* $\mathcal{A}(q_3, \mathcal{T})$. *Next we traverse* $\mathcal{A}(q_2, \mathcal{T})$ *and add* $\langle 2, n_2, 1 \rangle$ *and* $\langle 1, n_2, 0 \rangle$ *to* $\mathcal{A}(q_3, \mathcal{T})$. *Note we do not add* $\mathsf{m}' = \langle 0, n_1, 0 \rangle$ *as* $\mathsf{m}'_3 = 3 > \tau$. *Finally we add the strings on the leaf descendants of* n_2, n_3, n_5, n_{11} *and* n_{12} *to* $\mathcal{R}_3$ *and output* $\mathcal{R}_3 = \{s_1, s_2, s_3, s_4, s_5\}$.

Note in the matching-based framework, for each $\mathsf{m}' = \langle i', n', ed' \rangle \in \mathcal{A}(q_{i-1}, \mathcal{T})$ we need to find all the descendants n of n' s.t. $n.char = q[i]$ and $\mathsf{m}'_{(i-1,|n|-1)} \leq \tau$. We can achieve this by binary searching $\mathcal{I}[d][q[i]]$ where $d \in [|n'| + 1, |n'| + \tau + 1]$ to get nodes n with id within $n'.range$ and $\mathsf{m}'_{(i-1,|n|-1)} \leq \tau$. This is because on the one hand these nodes have label same as $q[i]$ and are descendants of n'. On the other hand for any descendant n of n', $\mathsf{m}'_{(i-1,|n|-1)} = ed + max(i - 1 - i', |n| - 1 - |n'|) \geq |n| - 1 - |n'|$. To satisfy $\mathsf{m}'_{(i-1,|n|-1)} \leq \tau$, it requires $|n| \leq |n'| + \tau + 1$.

The matching-based framework satisfies correctness and completeness as stated in Theorem 1.

THEOREM 1. *The matching-based framework satisfies (1) correctness: for each string* s *found by the matching-based framework,* $\mathsf{PED}(q, s) \leq \tau$, *and (2) completeness: for each string* $s \in \mathcal{S}$ *satisfying* $\mathsf{PED}(q, s) \leq \tau$, *it must be reported by the matching-based framework.*

Complexity: The time complexity for answering query q_i is $\mathcal{O}\big(|\mathcal{A}(q_{i-1}, \mathcal{T})|\tau \log |\mathcal{S}| + |\mathcal{A}(q_i, \mathcal{T})|(\tau^2 + \log |\mathcal{S}|)\big)$ where binary searching inverted lists costs $\mathcal{O}(|\mathcal{A}(q_{i-1}, \mathcal{T})|\tau \log |\mathcal{S}|)$, getting matching set costs $\mathcal{O}(|\mathcal{A}(q_i, \mathcal{T})|\tau^2)$ and getting the results costs $\mathcal{O}(|\mathcal{A}(q_i, \mathcal{T})| \log |\mathcal{S}|)$ as we can store the data in the order of their *id*s and perform binary search for each active matching to get the results. The space complexity is $\mathcal{O}(|\mathcal{S}|)$ as $\mathcal{S}$ is larger than $\mathcal{T}$, $\mathcal{H}$ and the matching sets.

5. COMPACT TREE BASED METHOD

We have an observation that the matching-based framework has a large number of redundant computations. Consider two active matchings with the active nodes n and p as shown in Figure 3. If $n = p$, it leads to redundant computation on the descendants of n. We combine the active matchings with the same active node to avoid this type of redundant computations in Section 5.1. If p is an ancestor of n, it may lead to redundant computation on the overlap region of their descendants. We only check those descendants of n with depth in $[|p| + \tau + 2, |n| + \tau + 1]$ to eliminate this kind of redundant computations in Section 5.2. To efficiently identify the active nodes with ancestor-descendant relationship (such as p and n), we design a compact tree index in Section 5.3. Lastly, we discuss how to maintain the compact tree index in Section 5.4.

5.1 Combining Active Matchings

We have an observation that some active matchings may share the same active node n and we need to perform redundant binary searches on the descendants of n with depth in $[|n| + 1, |n| + \tau + 1]$. For example, consider the two active matchings $\langle 1, n_2, 0 \rangle$ and $\langle 2, n_2, 1 \rangle$ of q_2 in Example 2, we

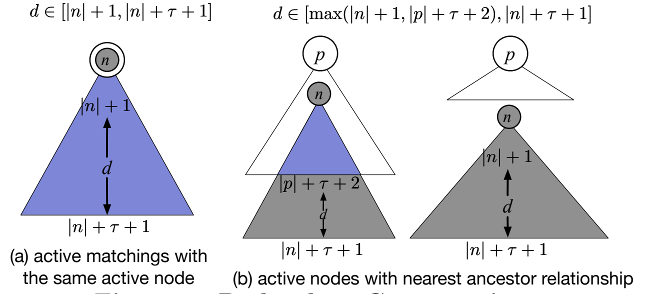


Figure 3: Redundant Computations.

need to check the descendants of n_2 twice. To address this issue, we combine the active matchings with the same active node. We use a hash map $\mathcal{F}$ to store all the active nodes where $\mathcal{F}[n]$ contains all the active matchings with the active node n. For each query q_i, for each node n s.t. $n.char = q[i]$, the matching-based framework enumerates every $\mathsf{m} \in \mathcal{A}(q_{i-1}, \mathcal{T})$ and uses $\mathsf{m}_{(i-1,|n|-1)}$ to update $\mathcal{H}[n]$. To achieve the same goal with $\mathcal{F}$, we enumerate every active node $n' \in \mathcal{F}$ and use $\min_{\mathsf{m} \in \mathcal{F}[n']} \mathsf{m}_{(i-1,|n|-1)}$ to update $\mathcal{H}[n]$. We discuss more details of utilizing $\mathcal{F}$ to answer the threshold-based query in Section 5.3.

5.2 Avoiding Redundant Binary Search

Both the matching-based framework and all the previous works [7,14,18,30] store the active nodes in a hash map and process them independently, and they involve many redundant computations. Using the matching-based framework as an example, consider the two active matchings $\langle 0, n_1, 0 \rangle$ and $\langle 1, n_2, 0 \rangle$ of q_2 in Example 2, as $[|n_1| + 1, |n_1| + \tau + 1] = [1, 3]$ and $[|n_2| + 1, |n_2| + \tau + 1] = [2, 4]$, it needs to perform duplicate binary searches on both $\mathcal{I}[2][q[3]]$ and $\mathcal{I}[3][q[3]]$.

Next we formally discuss how to avoid the redundant computations. Consider two active nodes n and p of the query q_{i-1} where p is an ancestor of n as shown in Figure 3(b). In the matching-based framework, we need to perfom binary search on the inverted lists $\mathcal{I}[d][q[i]]$ to find nodes with id within $n.range$ where $d \in [|n| + 1, |n| + 1 + \tau]$ and on the inverted lists $\mathcal{I}[d'][q[i]]$ to find nodes with id within $p.range$ where $d' \in [|p| + 1, |p| + 1 + \tau]$. As $p.range$ covers $n.range$, the binary search on the overlap region for $n.range$ is redundant. To avoid this, for the active node n, we do not perform binary searches on the overlap region, i.e., we only perform the binary search on $\mathcal{I}[d][q[i]]$ where $d \in [max(|n| + 1, |p| + 2 + \tau), |n| + 1 + \tau]$ to get nodes with id within $n.range$. Thus we can eliminate all the redundant binary search by maintaining the ancestor-descendant relationship between the active nodes. As $|p| + 2 + \tau$ is monotonically increasing with the depth $|p|$ and the nearest ancestor of n has the largest depth, we only need the nearest ancestor p of n to avoid the redundant binary searches. To this end, we design a compact tree index to keep the nearest ancestor relationship for the active nodes in Section 5.3.

5.3 The Compact Tree based Method

To effectively maintain the nearest ancestor for each active node, we design a compact tree index for the active nodes.

DEFINITION 8 (COMPACT TREE). *The compact tree of a query is a tree structure, which satisfies,*
(1) There is a bijection between the compact nodes and the active nodes of the query.

Table 4: A running example of the compact tree based method ($\tau = 2, q$='ssol', $\mathcal{T}$ in Figure 1).

i and query q_i	$i=1, q_1=$ s	$i=2, q_2=$ ss		$i=3, q_3=$ sso		$i=4, q_4=$ ssol		
$n \in \mathcal{F}$ ($n \in \mathcal{C}$)	n_1	n_1	n_2	n_1	n_2	n_2	n_3	$\cdots$
$\mathsf{m} \in \mathcal{F}[n]$	$\langle 0, n_1, 0\rangle$	$\langle 0, n_1, 0\rangle$	$\langle 1, n_2, 0\rangle$	$\langle 0, n_1, 0\rangle$	$\langle 1, n_2, 0\rangle$ $\langle 2, n_2, 1\rangle$	$\langle 1, n_2, 0\rangle$ $\langle 2, n_2, 1\rangle$	$\langle 3, n_3, 1\rangle$	$\cdots$
$\langle d, \mathcal{L}\rangle$	$\langle 1, (n_2)\rangle$	$\langle 1, (n_2)\rangle$		$\langle 2, (n_3)\rangle$, $\langle 3, (n_{12})\rangle$	$\langle 4, (n_5, n_{11})\rangle$	$\langle 3, (n_6)\rangle$		$\cdots$

Algorithm 3: CompactTreeBasedMethod

Input: $\mathcal{T}$: a trie; τ: a threshold; q: a continuous query;
Output: $\mathcal{R}_i = \{s \in \mathcal{S} \mid \mathsf{PED}(q_i, s) \le \tau\}$ for each $1 \le i \le |q|$;

1 $\mathcal{F}[\mathcal{T}.root] = \{\langle 0, \mathcal{T}.root, 0\rangle\}$ and add $\mathcal{T}.root$ to $\mathcal{C}$;
2 **foreach** *query* q_i *where* $1 \le i \le |q|$ **do**
3 traverse $\mathcal{C}$ in pre-order and get all its nodes;
4 **foreach** *compact node* n *in pre-order* **do**
5 **foreach** $d \in [max(|n|+1, |p|+2+\tau), |n|+1+\tau]$ *where* p *is the parent of* n *in* $\mathcal{C}$ **do**
6 $\mathcal{L} = \text{BinarySearch}(\mathcal{I}[d][q[i]], n.range)$;
7 $h = min_{\mathsf{m} \in \mathcal{F}[n]} \mathsf{m}_{(i-1,d-1)}$;
8 `AddIntoCompactTree`$(n, \mathcal{L}, d, i, h)$;
9 **foreach** *compact node* n *in* $\mathcal{C}$ **do**
10 **foreach** $\mathsf{m} \in \mathcal{F}[n]$ **do** remove m if $\mathsf{m}_i > \tau$;
11 **if** $\mathcal{F}[n] = \phi$ **then**
12 remove n from $\mathcal{F}$, and $\mathcal{C}$ by setting the parent of n as the parent of n's children;
13 **foreach** *first-level compact node* n *in* $\mathcal{C}$ **do**
14 add all the leaf descendants of n to $\mathcal{R}_i$;
15 output $\mathcal{R}_i$;

(2) For any two compact nodes p and n and their corresponding active nodes p' and n', p is the parent of n iff p' is the nearest ancestor of n' among all the active nodes.
(3) The children of a compact node are ordered by their ids.

For example, Figure 5 shows the compact tree of the query q_3 in Example 2. Note the active nodes of q_3 are shown in Figure 1 with bold border. For ease of presentation, we interchangeably use the compact node with its corresponding active node when the context is clear.

We discuss how to build and maintain the compact tree in Section 5.4. In this section we focus on utilizing the compact tree to answer the threshold-based query while avoiding the redundant binary search. The compact tree based method is similar to the matching-based framework except that the active nodes are processed in a top-down manner, we only binary search the non-overlapping region of the descendants of an active node and the active-node set $\mathcal{F}$ and the compact tree $\mathcal{C}$ are updated in-place. The pseudo code is shown in Algorithm 3. Initially, it sets $\mathcal{F}[\mathcal{T}.root] = \{\langle 0, \mathcal{T}.root, 0\rangle\}$ and inserts the active node $\mathcal{T}.root$ of q_0 to the empty compact tree $\mathcal{C}$ (Line 1). Then for each query q_i, instead of processing the active nodes independently, it processes them in a top-down manner. Specifically, it first traverses the compact tree in pre-order and gets all the compact nodes (Line 3). Then for each compact node n (in pre-order), for each depth $d \in [max(|n|+1, |p|+2+\tau), |n|+1+\tau]$ where p is the parent of n in the compact tree, it binary searches the inverted list $\mathcal{I}[d][q[i]]$ to get a list $\mathcal{L}$ of ordered nodes with id within $n.range$, i.e., the nodes in $\mathcal{L}$ are ordered, have label $q[i]$, with depth d and are descendants of n (Lines 4 to 6). Then it inserts the active nodes in $\mathcal{L}$ to the compact tree $\mathcal{C}$ and adds the corresponding active matchings, which are the second kind as described in Section 4, to $\mathcal{F}$ using the procedure `AddIntoCompactTree` which we discuss later in Section 5.4. Note h is used to keep the minimum deduced edit distance (Lines 7 to 8). Next for each compact node n, it removes the non-active matchings m in $\mathcal{F}[n]$ where $\mathsf{m}_i > \tau$ and thus all the first kind of active matchings are remained in $\mathcal{F}[n]$ (Line 10). If all the matchings in $\mathcal{F}[n]$ are removed, it also removes the node n from $\mathcal{F}$ and $\mathcal{C}$ by pointing the parent of n to the children of n (Line 12). Finally it adds all the leaf descendants of the first-level compact nodes in $\mathcal{C}$ to $\mathcal{R}_i$ and outputs $\mathcal{R}_i$ as the leaf descendants of all the other compact nodes are covered by those of the first-level (Lines 14 to 15).

Example 3. *Table 4 shows a running example of the compact tree based method. For query q_3, we traverse $\mathcal{C}$ and get two compact nodes n_1 and n_2. For n_1, as $[max(0+1, -\infty+2+2), 0+1+2] = [1,3]$ (Note if a compact node has no parent p in $\mathcal{C}$, we set $|p|$ as $-\infty$), we binary search $\mathcal{I}[1][\mathtt{o}]$, $\mathcal{I}[2][\mathtt{o}]$, and $\mathcal{I}[3][\mathtt{o}]$ for $n_1.range$ and get three lists $\mathcal{L}$ of ordered nodes ϕ, $\{n_3\}$ and $\{n_{12}\}$. For the compact node n_2, as $[max(1+1, 0+2+2), 1+1+2] = [4,4]$, we binary search the inverted list $\mathcal{I}[4][\mathtt{o}]$ for $n_2.range$ and get a list $\mathcal{L}$ of ordered nodes $\mathcal{L} = \{n_5, n_{11}\}$. Note we only show the non-empty lists in the table. We discuss later how to add the active nodes in these lists to $\mathcal{C}$ and $\mathcal{F}$. For node n_1, as $\langle 0, n_1, 0\rangle_3 = 3 > \tau$, we remove it from $\mathcal{F}[n_1]$ and have $\mathcal{F}[n_1] = \phi$. Thus we also remove n_1 from $\mathcal{F}$ and $\mathcal{C}$ and now $\mathcal{C}$ is that in Figure 5. Finally we add the leaves of the first-level compact node n_2 in $\mathcal{C}$ to $\mathcal{R}_3$ and have $\mathcal{R}_3 = \{s_1, s_2, s_3, s_4, s_5\}$.*

We can see that for each query, each trie node is accessed at most once by the active nodes. The compact tree based method is correct and complete as stated in Theorem 2.

Theorem 2. *The compact tree based method satisfies correctness and completeness.*

5.4 Adding Active Nodes to the Compact Tree

For a query q_i, for a compact node n and a depth d, the compact tree based method binary searches the inverted list $\mathcal{I}[d][q[i]]$ and get a list $\mathcal{L}$ of ordered nodes which are all descendants of n in the trie and with depth d. In this section we focus on inserting the active nodes in $\mathcal{L}$ to the compact tree and add the corresponding active matchings to $\mathcal{F}$. The basic idea is that for each node $a \in \mathcal{L}$ we find a proper position for it in $\mathcal{C}$. Then we compute $ed = \mathsf{ED}(q_i, a)$ if $\mathsf{ED}(q_i, a) \le \tau$. As $ed + (i - i) = ed \le \tau$, $\langle i, a, ed\rangle$ is an active matching and we add it to $\mathcal{F}[a]$. In addition, a is an active node and we insert a to $\mathcal{C}$ at the proper position.

We first discuss how to find a proper position in $\mathcal{C}$ for a node $a \in \mathcal{L}$ to insert to. The proper position for a in $\mathcal{C}$ should satisfy the three conditions in Definition 8. As the node $a \in \mathcal{L}$ is a descendant of n in the trie, based on condition 2 of Definition 8, we should also insert a as a descendant of n in the compact tree. Based on condition 3 of Definition 8, the children of n are already ordered and the proper position for a in $\mathcal{C}$ should keep the children of n ordered. To this end, we develop a procedure `AddIntoCompactTree` which sequentially compare a with each child c of n and try to find the proper position for a as a child of n. There are five cases in the comparison of a and c as shown in Figure 4. **Case 1:** a is on the left of c, i.e., $a.up < c.lo$. In this case the proper position of a is exactly the left of c and the child

Figure 4: Comparing nodes in $\mathcal{L}$ with children of the compact node n.

Figure 5: An example of adding active nodes.

of n as the comparisons are performed sequentially and the left siblings of c are all smaller than a.

Case 2: a is an ancestor of c, i.e., $c.range \subset a.range$. In this case, based on condition 2 of Definition 8, the proper position of a is the child of n and the parent of c and all the right siblings c' of c s.t. $c'.range \subset a.range$.

Case 3: $a = c$, i.e., $a.range = c.range$. In this case a already exists in the compact tree. To satisfy the condition 1 of Definition 8, we do not insert a to $\mathcal{C}$. Thus there is no proper position for a in the compact tree.

Case 4: a is a descendant of c, i.e., $a.range \subset c.range$. In this case, based on condition 2 of Definition 8, the proper position of a in $\mathcal{C}$ should be a descendant of c. Thus we recursively invoke the procedure `AddIntoCompactTree` which compares a with the children of c to find the proper position.

Case 5: a is on the right of c, i.e., $c.up < a.lo$. In this case, to keep the descendants of n ordered, we skip c and compare a with the next sibling of c.

Note if c reaches the end, a is larger than all the children of n and the proper position for a is the last child of n. Based on the structure of trie and compact tree, all the other cases cannot happen. Thus we can recursively find a proper position in $\mathcal{C}$ for each node in $\mathcal{L}$. Moreover, as the nodes in $\mathcal{L}$ are ordered and so are the children of n, instead of processing each node in $\mathcal{L}$ independently, we actually can perform the comparison between all the nodes in $\mathcal{L}$ and all the children of n in a merging fashion. Specifically, we first compare the first node in $\mathcal{L}$ with the first child of n. Whenever we get the proper position in $\mathcal{C}$ for a node $a \in \mathcal{L}$, we continuously compare the next node of a in $\mathcal{L}$ with the current child c of n and check the five cases until we reach the end of $\mathcal{L}$. In this way we can get all the proper positions for nodes in $\mathcal{L}$.

Next we discuss how to compute $ed = \mathsf{ED}(q_i, a)$ in the situation $\mathsf{ED}(q_i, a) \leq \tau$ where $a \in \mathcal{L}$ and $|a| = d$. Based on the discussion in Section 4, when $\mathsf{ED}(q_i, a) \leq \tau$, $\mathsf{ED}(q_i, a)$ is the minimum of $\mathsf{m}_{(i-1,|a|-1)}$, where $\mathsf{m} = \langle i', n', ed' \rangle$ is an active matching of q_{i-1} in $\mathcal{M}(q_{i-1}, a.parent)$ s.t. $\mathsf{m}_{(i-1,|a|-1)} \leq \tau$. As m is an active matching, n' is an active node and n' is in $\mathcal{C}$. Moreover, as $\mathsf{m} \in \mathcal{M}(q_{i-1}, a.parent)$, n' is an ancestor of a in the trie. Based on condition 2 of Definition 8, n' is also an ancestor of a in $\mathcal{C}$. In addition, as $\tau \geq \mathsf{m}_{(i-1,|a|-1)} = ed' + \max(i-1-i', d-1-|n'|) \geq d-1-|n'| \geq |p|+2+\tau-1-|n'|$ where p is the nearest ancestor of n among all active nodes, we have $|n'| > |p|$. Thus n' is on the path from n (included) to a (not included) in $\mathcal{C}$. Thus to get $\mathsf{ED}(q_i, a)$ we only need to enumerate every active node n' on the path from n to a and the minimum of $\min_{\mathsf{m} \in \mathcal{F}[n']} \mathsf{m}_{(i-1,d-1)}$ is $ed = \mathsf{ED}(q_i, a)$ if $\mathsf{ED}(q_i, a) \leq \tau$. We can achieve this simultaneously with the processing of finding the proper position for a by updating h as $\min_{\mathsf{m} \in \mathcal{F}[c]} \mathsf{m}_{(i-1,d-1)}$ if the latter is smaller whenever we invoke the procedure `AddIntoCompactTree` in Case 4. Finally if $ed \leq \tau$, $\langle i, a, ed \rangle$ is an active matching and we add it to $\mathcal{F}[a]$. In addition, a is an active node and we insert a to the proper position in $\mathcal{C}$.

EXAMPLE 4. *Figure 5 gives an example of adding a node to the compact tree $\mathcal{C}$ of the query q_3 =‘sso’. For $i = 4$ and q_4 =‘ssol’, for node n_2 and $d = 3$, we have $\mathcal{L} = \{n_6\}$. Next we add n_6 to $\mathcal{C}$ and calculate $\mathsf{ED}(q_4, n_6)$. We use h to keep the minimum deduced edit distance. As $\mathcal{F}[n_2] = \{\langle 1, n_2, 0\rangle, \langle 2, n_2, 1\rangle\}$, initially we set $h = \langle 1, n_2, 0\rangle_{(3,2)} = 2$. We first compare n_6 with the first child n_3 of n_2 and we have $n_3.range \subset n_6.range$ (Case 4). Thus we recursively invoke this procedure and update h as $\langle 3, n_3, 1\rangle_{(3,2)} = 1$ as $\mathcal{F}[n_3] = \{\langle 3, n_3, 1\rangle\}$. We compare n_6 with the first child n_5 of n_3 and have $n_5.u < n_6.l$ (Case 5). Thus we move forward and compare n_6 with next sibling n_{11} of n_5. As $n_{11}.range \subset n_6.range$ (Case 2) and $n_{12}.range \not\subset n_6.range$, the proper position for n_6 in $\mathcal{C}$ is the child of n_3 and the parent of n_{11}. Moreover as $h = 1 \leq \tau$, we add the active matching $\langle 4, n_6, 1\rangle$ to $\mathcal{F}[n_6]$ and insert n_6 to the proper position in $\mathcal{C}$.*

Complexity: The time complexity for answering query q_i is $\mathcal{O}(|\mathcal{C}|\tau \log|\mathcal{S}| + |\mathcal{A}(q_i, \mathcal{T})|\tau^2 + |\mathcal{C}'|\log|\mathcal{S}|)$ where $\mathcal{C}$ and $\mathcal{C}'$ are respectively the compact trees before and after processing the query q_i. Note $|\mathcal{C}| \leq |\mathcal{A}(q_{i-1}, \mathcal{T})|$ and $|\mathcal{C}'| \leq |\mathcal{A}(q_i, \mathcal{T})|$. Also in practice we can skip the redundant binary searches and the cost for binary search is smaller than $\mathcal{O}(|\mathcal{C}|\tau \log|\mathcal{S}|)$. The space complexity is $\mathcal{O}(|\mathcal{S}|)$.

6. SUPPORTING TOP-K QUERIES

Given two continuous queries q_i and q_{i+1}, suppose b_{i+1} (or b_i) is the maximum prefix edit distance between q_{i+1} (or q_i) and its top-k results. We prove that $b_{i+1} = b_i$ or $b_{i+1} = b_i + 1$ (Section 6.1). Thus we can first use the same techniques for threshold-based queries to find all strings with prefix edit distance to q_{i+1} equal to b_i. If there are not enough results, we expand the matching set and find those data strings with prefix edit distance to q_{i+1} equal to $b_i + 1$ until we get k results (Section 6.2). In this way, we can answer the top-k query using the matching-based method (Section 6.3).

6.1 The b-Matching Set

Given two queries q_{i-1} and q_i, where q_i is a new query by adding a keystroke after q_{i-1}, and let $\mathcal{R}_i$ denote the top-k answers of q_i and b_i denote the maximal prefix edit distance between q_i and the top-k answers of q_i (i.e., $b_i = \max_{s \in \mathcal{R}_i} \mathsf{PED}(q_i, s)$). We have an observation that either $b_i = b_{i-1}$ or $b_i = b_{i-1} + 1$. This is because on the one hand, for any $s \in \mathcal{R}_{i-1}$, we have $\mathsf{PED}(q_i, s) \leq \mathsf{PED}(q_{i-1}, s) + 1 \leq b_{i-1} + 1$ as we can first delete the last character of q_i and then

transform the rest of q_i to a prefix of s with $\mathsf{PED}(q_{i-1}, s)$ edit operations. Thus there are at least k strings in $\mathcal{S}$ with prefix edit distances to q_i no larger than $b_{i-1} + 1$ which leads to $b_i \le b_{i-1} + 1$. On the other hand, for any string s, based on the definition of prefix edit distance, we have $\mathsf{PED}(q_i, s) \ge \mathsf{PED}(q_{i-1}, s)$, i.e., the prefix edit distance from the continuous query to a string is monotonically increasing with the query length. This leads to $b_i \ge b_{i-1}$. Thus we have either $b_i = b_{i-1}$ or $b_i = b_{i-1} + 1$ as stated in Lemma 4.

LEMMA 4. *Given a continuous top-k query q, for any* $1 \le i \le |q|$ *we have either* $b_i = b_{i-1}$ *or* $b_i = b_{i-1} + 1$.

For example, consider the dataset in Table 1. For the top-3 queries q_1='s', q_2='ss', and q_3='sso', we have $b_1 = 0, b_2 = 1$ and $b_3 = 1$. Note $b_0 = 0$ for any top-k query q.

Based on Lemma 4 we give the basic idea of the matching-based method for top-k query. For each query q_i, as either $b_i = b_{i-1}$ or $b_i = b_{i-1} + 1$, we first find all the strings in $\mathcal{S}$ with prefix edit distance to q_i less than b_{i-1}. Then we find those equal to b_{i-1} until we get k results and set $b_i = b_{i-1}$. If there are not enough results, we continuous to find those equal to $b_{i-1}+1$ until we get k results and set $b_i = b_{i-1}+1$. In this way we can answer the top-k query. The challenge in the matching-based method is how to get the strings with specific prefix edit distance to a query. Before addressing this challenge, we introduce a concept.

DEFINITION 9 (b-MATCHING). *A matching* $\langle i, n, ed\rangle$ *is a* b*-matching iff* $ed \le b$. *It is an exact* b*-matching iff* $ed = b$.

For example, the matching $\langle 1, n_2, 0\rangle$ of q_2 ='ss' is a 1-matching. All the b-matchings of a query q compose its b-matching set $\mathcal{P}(q, b, \mathcal{T})$, abbreviated as $\mathcal{P}(q, b)$ if the context is clear. For example, $\mathcal{P}(q_2, 1)=\{\langle 0, n_1, 0\rangle, \langle 1, n_2, 0\rangle, \langle 2, n_2, 1\rangle\}$.

We have an observation that given a query q, for any string $s \in \mathcal{S}$, if $\mathsf{PED}(q, s) \le b$, there must exist a b-matching $\langle i, n, ed\rangle$ s.t. s is a leaf descendant of n. This is because based on Lemma 2, if $\mathsf{PED}(q, s) \le b$, there exists a matching $\mathsf{m} = \langle i, j, ed\rangle$ s.t. $\mathsf{m}_{|q|} = ed + (|q| - i) \le b$. Suppose n is the corresponding node of $s[1, j]$ in the trie $\mathcal{T}$, $\langle i, n, ed\rangle$ is a b-matching as $ed = \mathsf{ED}(q[1, i], s[1, j]) = \mathsf{ED}(q[1, i], n)$ and $ed \le ed + (|q| - i) \le b$. Thus we can use $\mathcal{P}(q, b)$ to get all the strings in $\mathcal{S}$ with prefix edit distance to q within b.

Moreover, given a continuous query q, for any integer b and $1 \le i \le |q|$, we find that we can (1) calculate $\mathcal{P}(q_i, b - 1)$ based on $\mathcal{P}(q_{i-1}, b)$ and (2) calculate $\mathcal{P}(q_i, b)$ based on $\mathcal{P}(q_i, b - 1)$. We discuss how to achieve these later in Section 6.2. Thus given a b-matching set $\mathcal{P}(q_{i-1}, b)$, we can calculate $\mathcal{P}(q_i, b - 1)$, $\mathcal{P}(q_i, b)$, and $\mathcal{P}(q_i, b + 1)$ as follows.

$$\mathcal{P}(q_{i-1}, b) \xrightarrow{(1)} \mathcal{P}(q_i, b - 1) \xrightarrow{(2)} \mathcal{P}(q_i, b) \xrightarrow{(2)} \mathcal{P}(q_i, b + 1)$$

Then we can answer the top-k query using the b-matching set as follows. Given a trie $\mathcal{T}$ and a continuous query q, initially we have $b_0 = 0$ and $\mathcal{P}(q_0, 0) = \{\langle 0, \mathcal{T}.root, 0\rangle\}$. Then for each $1 \le i \le |q|$, we can use $\mathcal{P}(q_{i-1}, b_{i-1})$ to calculate $\mathcal{P}(q_i, b_i)$ and answer the query q_i. This is because on the one hand, we can use $\mathcal{P}(q_{i-1}, b_{i-1})$ to calculate $\mathcal{P}(q_i, b_{i-1} - 1)$, $\mathcal{P}(q_i, b_{i-1})$ and $\mathcal{P}(q_i, b_{i-1} + 1)$. On the other hand, we can use $\mathcal{P}(q_i, b_{i-1} - 1)$, $\mathcal{P}(q_i, b_{i-1})$ and $\mathcal{P}(q_i, b_{i-1} + 1)$ to get all the strings with prefix edit distance to q_i less than b_{i-1}, equal to b_{i-1} and equal to $b_{i-1} + 1$. Based on Lemma 4, $\mathcal{R}_i$ can be achieved from these strings and $\mathcal{P}(q_i, b_i)$ is either $\mathcal{P}(q_i, b_{i-1})$ or $\mathcal{P}(q_i, b_{i-1}+1)$. In this way we can answer the continuous query q. Next we calculate the b-matching sets.

6.2 Calculating the b-Matching Set

We first discuss calculating $\mathcal{P}(q_i, b-1)$ based on $\mathcal{P}(q_{i-1}, b)$, which is (almost) all the same as the incremental active matching set calculation when $\tau = b - 1$ in Section 4. There are two kinds of $(b-1)$-matchings $\mathsf{m}''=\langle i'', n'', ed''\rangle$ in $\mathcal{P}(q_i, b-1)$. Those with $i'' > i$ and those with $i'' = i$. Based on Definition 9, $ed'' \le b - 1$. For the first case, m'' is also a b-matching in $\mathcal{P}(q_{i-1}, b)$ as $ed'' \le b-1 \le b$. Thus we can get all of them from $\mathcal{P}(q_{i-1}, b)$. To get all the $(b-1)$-matchings where $i = i''$, for each node n'' s.t. $n''.char = q[i]$, we enumerate every $\mathsf{m}' = \langle i', n', ed'\rangle \in \mathcal{P}(q_{i-1}, b)$ where n' is an ancestor of n'' and $\mathsf{m}'_{(i-1,|n''|-1)} \le b-1$, and have the minimum of $\mathsf{m}'_{(i-1,|n''|-1)}$ is ed''=$\mathsf{ED}(q_i, n'')$ if $\mathsf{ED}(q_i, n'') \le b-1$. This is because based on Lemma 3 $\mathsf{ED}(q_i, n'')$ is the minimum of $\mathsf{m}'_{(i-1,|n''|-1)}$ where $\mathsf{m}' = \langle i', n'.ed'\rangle \in \mathcal{M}(q_{i-1}, n''.parent)$ while m' should also be a b-matching and thus in $\mathcal{P}(q_{i-1}, b)$ as $ed' \le \mathsf{m}'_{(i-1,|n|-1)} \le b-1 \le b$. In this way we can also get all the second kind of $(b-1)$-matchings based on $\mathcal{P}(q_{i-1}, b)$.

Next we discuss calculating $\mathcal{P}(q_i, b)$ based on $\mathcal{P}(q_i, b - 1)$. As $\mathcal{P}(q_i, b-1) \subseteq \mathcal{P}(q_i, b)$, we only need to calculate those exact b-matchings in $\mathcal{P}(q_i, b) - \mathcal{P}(q_i, b-1)$. Consider any exact b-matching $\mathsf{m}'' = \langle i'', n'', ed'' = b\rangle$ in $\mathcal{P}(q_i, b) - \mathcal{P}(q_i, b - 1)$. Based on Lemma 3, as $q[i''] = n''.char$, there exists a matching $\mathsf{m}' = \langle i', n', ed'\rangle \in \mathcal{M}(q_{i''-1}, n''.parent)$ s.t. $ed'' = \mathsf{m}'_{(i''-1,|n''|-1)}$. This leads to $ed' \le \mathsf{m}'_{(i''-1,|n''|-1)} = ed'' = b$. If $ed' < b$, m' is a $(b-1)$-matching in $\mathcal{P}(q_i, b-1)$. Otherwise, $ed' = b$ and m' is an exact b-matching in $\mathcal{P}(q_i, b) - \mathcal{P}(q_i, b - 1)$. Thus to get all the exact b-matchings, we can enumerate every $(b - 1)$-matching $\mathsf{m}' = \langle i', n', ed'\rangle$ in $\mathcal{P}(q_i, b-1)$ and find all the descendants n'' of n' and $i'' > i'$ s.t. $q[i''] = n''.char$ and $\mathsf{m}'_{(i''-1,|n''|-1)} = b$. If $\langle i'', n'', *\rangle \notin \mathcal{P}(q_i, b - 1)$ where $*$ denotes an arbitrary integer[6], we have $\mathsf{ED}(q_{i''}, n'') = b$. This is because $\langle i'', n'', *\rangle \notin \mathcal{P}(q_i, b - 1)$ indicates $\mathsf{ED}(q_{i''}, n'') \ge b$ while $\mathsf{m}'_{(i''-1,|n''|-1)} = b$ indicates $\mathsf{ED}(q_{i''}, n'') \le b$. Thus $\langle i'', n'', ed'' = b\rangle$ is an exact b-matching. For the newly generated exact b-matchings, we repeat the process above until there is no more new exact b-matchings. In this way we can get all the exact b-matchings in $\mathcal{P}(q_i, b)-\mathcal{P}(q_i, b-1)$ and achieve $\mathcal{P}(q_i, b)$ using $\mathcal{P}(q_i, b-1)$.

6.3 Matching-based Method for Top-k Queries

Based on Lemma 4, it is easy to see that $\mathcal{P}(q_{i-1}, b_{i-1}) \subseteq \mathcal{P}(q_i, b_i)$ for any $1 \le i \le |q|$. Thus we calculate the b-matching set in-place using $\mathcal{P}$. The pseudo-code of the matching-based method for top-k queries is shown in Algorithm 4. It first initializes $b_0 = 0$ and the 0-matching set $\mathcal{P}$ as $\{\langle 0, \mathcal{T}.root, 0\rangle\}$ (Line 1). Then for q_i, it first gets all the $(b_{i-1} - 1)$-matchings and adds them to $\mathcal{P}$ by the procedure `FirstDeducing` (Line 3). Then it gets all the strings with prefix edit distance to q_i less than b_{i-1} using $\mathcal{P}$ and adds them to $\mathcal{R}_i$ (Lines 4 to 5). Next it invokes the procedure `SecondDeducing` to get the rest of answers, sets b_i and outputs $\mathcal{R}_i$ (Lines 6 to 9). The procedure `FirstDeducing` is the same as the inner loop of Algorithm 2. `SecondDeducing` takes $\mathcal{P}$, $\mathcal{R}_i$, query length i, and two integers b and k as input and outputs true if it finds enough results whose prefix edit distance to q_i are b. For each m' in $\mathcal{P}$, if $\mathsf{m}'_i = b$, it adds the leaves of n' to $\mathcal{R}_i$ until $|\mathcal{R}_i| = k$ and returns true (Lines 2 to 4). If there is not enough results, it finds all the descendants n'' of n' and $i'' > i'$ s.t. $q[i''] = n''.char$

[6] We can achieve this by implementing $\mathcal{P}(q_i, b)$ as a hash map and use $\langle i', n'\rangle$ as the key of the b-matching $\langle i', n', ed'\rangle$ in it.

Table 5: A Running Example of the Matching-based Method for Top-k Queries ($k=3, q=$‘sso’, $\mathcal{T}$ in Figure 1).

i, query q_i and b_{i-1}	$i=1,\ q_1=\texttt{s},\ b_0=0$		$i=2,\ q_2=\texttt{ss},\ b_1=0$		$i=3,\ q_3=\texttt{sso},\ b_2=1$			
$\mathcal{P}$	$\langle 0,n_1,0\rangle$	$\langle 1,n_2,0\rangle$	$\langle 1,n_2,0\rangle$	$\langle 0,n_1,0\rangle$	$\langle 1,n_2,0\rangle$	$\langle 3,n_3,1\rangle$	$\langle 3,n_{12},1\rangle$	$\langle 0,n_1,0\rangle$
$< b_{i-1}$	Φ		Φ	Φ	Φ			Φ
$= b_{i-1}$	$\langle 1,n_2,0\rangle$	s_1,s_2,s_3	Φ	Φ	$\langle 3,n_3,1\rangle$ $\langle 3,n_{12},1\rangle$	s_1,s_2,s_3		
$= b_{i-1}+1$			s_1,s_2,s_3					

Algorithm 4: MatchingBasedMethodForTopK

Input: $\mathcal{T}$: a trie; k: an integer; q: a continuous query;
Output: $\mathcal{R}_i=\{$top-k answers for $q_i\}$ for each $1\le i\le|q|$;

```
1 b_0 = 0, P = {⟨0, T.root, 0⟩};
2 foreach query q_i where 1 ≤ i ≤ |q| do
3     FirstDeducing(P, b_{i-1}, i);
4     foreach matching m' = ⟨i', n', ed'⟩ ∈ P do
5         if m'_i < b_{i-1} then add the leaves of n' to R_i;
6     if SecondDeducing(P, i, R_i, b_{i-1}, k) then
7         set b_i = b_{i-1} and output R_i
8     else if SecondDeducing(P, i, R_i, b_{i-1}+1, k) then
9         set b_i = b_{i-1} + 1 and output R_i;
```

Procedure SecondDeducing($\mathcal{P}, i, \mathcal{R}_i, b, k$)

Input: $\mathcal{P}$: A matching set; i: query length;
$\mathcal{R}_i$: the result set; b,k: two integers.
Output: `true` if $|\mathcal{R}_i|=k$. Otherwise, `false`.

```
1 foreach m' = ⟨i', n', ed'⟩ ∈ P do
2     if m'_i = b then
3         add the leaves of n' to R_i until |R_i| = k;
4         if |R_i| = k then return true
5     find all the descendants n'' of n' and i'' > i' s.t.
      q[i'']=n''.char and m'_(i''-1,|n''|-1)=b and append
      ⟨i'', n'', b⟩ to P for looping if ⟨i'', n'', *⟩ ∉ P;
6 return false;
```

and $\mathsf{m}'_{(i''-1,|n''|-1)}=b$ and appends $\langle i'',n'',b\rangle$ to $\mathcal{P}$ to get new exact b-matchings if $\langle i'',n'',*\rangle\notin\mathcal{P}$ (Line 5). Note this can be achieved by binary searching $\mathcal{I}[|n''|][q[i'']]$, where $|n''|=|n'|+1+b-ed'$ and $i''\in[i'+1,i'+1+b-ed']$ or $i''=i'+1+b-ed'$ and $|n''|\in[|n'|+1,|n'|+1+b-ed']$ (less than $2b$ binary searches). Finally it returns false (Line 6).

Example 5. *Table 5 shows a running example for top-k queries. The last three rows show the achieved answers or matchings with deduced (prefix) edit distance less than b_{i-1}, equalling b_{i-1} and equalling $b_{i-1}+1$ respectively. For the query q_3, we have $b_2=1$. At first, $\mathcal{P}=\{\langle 1,n_2,0\rangle,\langle 0,n_1,0\rangle\}$. As the deduced (prefix) edit distances based on these matchings are not less than $b_2=1$, we do not find any results. Then we invoke SecondDeducing to find answers and matchings with (prefix) edit distance equalling b_2. For $\mathsf{m}=\langle 1,n_2,0\rangle$, as $\mathsf{m}_3=2\neq b$, we do not get any answers. However we have the descendants n_3 and n_{12} of n_2 and $i''=3>1$ s.t. the deduced edit distance based on m equal to $b_2=1$. Thus we add the two matchings $\langle 3,n_3,1\rangle$ and $\langle 3,n_{12},1\rangle$ to $\mathcal{P}$. We then process $\mathsf{m}=\langle 3,n_3,1\rangle$. As $\mathsf{m}_3=1$, we add the leaves of n_3 to $\mathcal{R}_3$ and get $\mathcal{R}_3=\{s_1,s_2,s_3\}$.*

The matching-based method for top-k queries satisfies correctness as stated in Theorem 3.

Theorem 3. *The matching-based method for top-k queries correctly finds the top-k results.*

Complexity: For the top-k query q_i, as `FirstDeducing` is the same as the inner loop of Algorithm 2 by setting τ as $b_{i-1}-1$, it costs $\mathcal{O}\big(|\mathcal{P}(q_{i-1},b_{i-1})|(b_{i-1}-1)\log|\mathcal{S}|+|\mathcal{P}(q_i,b_{i-1}-1)|((b_{i-1}-1)^2+\log|\mathcal{S}|)\big)$. `SecondDeducing` costs $\mathcal{O}(|\mathcal{P}(q_i,b_i)|b_i\log|\mathcal{S}|)$ as for each matching in $\mathcal{P}(q_i,b_i)$ it conducts at most $2b_i$ binary searches. Based on Definition 9, $\mathcal{P}(q_{i-1},b_{i-1})\subseteq\mathcal{P}(q_i,b_i)$ and $\mathcal{P}(q_i,b_{i-1}-1)\subseteq\mathcal{P}(q_i,b_i)$. Thus the overall time complexity is $\mathcal{O}\big(|\mathcal{P}(q_i,b_i)|(b_i^2+b_i\log|\mathcal{S}|)\big)$. The space complexity is $\mathcal{O}(|\mathcal{S}|)$.

Discussion: When there are a lot of data strings with the same maximum prefix edit distance to the query, we can re-rank them by the other scoring functions, such as TF/IDF.

Table 6: Datasets.

Datasets	Cardinality	Avg Len	Max Len	Min Len	$\lvert\Sigma\rvert$
Word	146,033	8.77	30	1	27
Querylog	1,000,000	19.06	32	2	56

7. EXPERIMENTS

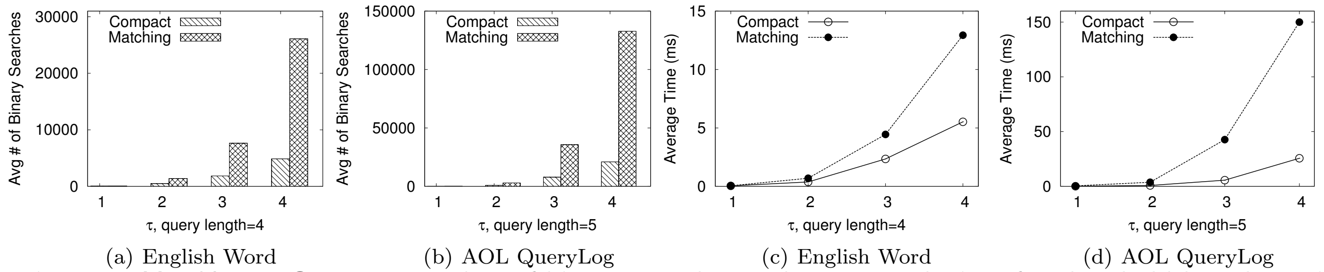
We conducted extensive experiments to evaluate the efficiency and scalability of our techniques. We compared our method META with state-of-the-art approaches IncNGTrie [30], ICAN [14] and IPCAN [18]. As state-of-the-art methods cannot answer top-k queries, we extended them to support top-k queries as follows. Based on Lemma 4, for each query q_i, either $b_i{=}b_{i-1}$ or $b_i{=}b_{i-1}{+}1$. Thus we first invoked the state-of-the-art methods to calculate all the strings with prefix edit distance to the query not larger than b_{i-1} based on active node sets. If the number of returned strings is no smaller than k, we returned the smallest k results from them. Otherwise, we increased the threshold by 1 and invoked the state-of-the-art methods with the new threshold $b_i{=}b_{i-1}{+}1$ to calculate k results from scratch. We also compared with the string similarity search methods HSTree [27] and Pivotal [8] for threshold-based queries and HSTree [27] and TopK [9] for top-k queries using the adaptation as described in Section 2.2. We obtained all the source codes from the authors. All the methods were implemented using C++ and compiled using g++ 4.8.2 with -O3 flag. All the experiments were conducted on a machine running on 64-bit Ubuntu Server 12.04 LTS version with an Intel Xeon E5-2650 2.00 GHz processor and 48 GB memory.

Dataset: We used two real datasets Word and Querylog[7]. Word contained 146,033 English words. Querylog contained 1 million query logs from AOL. The details are given in Table 6. We used 1000 common misspellings in Wikipedia[8] as queries for Word and randomly chose 1000 strings from Querylog as the queries for Querylog. We did not make any change for queries and data on Querylog. These queries were representative as they contained typos in real world. We reported the average querying time where τ and k were evenly distributed in [1,6] and [10,40].

7.1 Evaluating the Compact Tree based Method

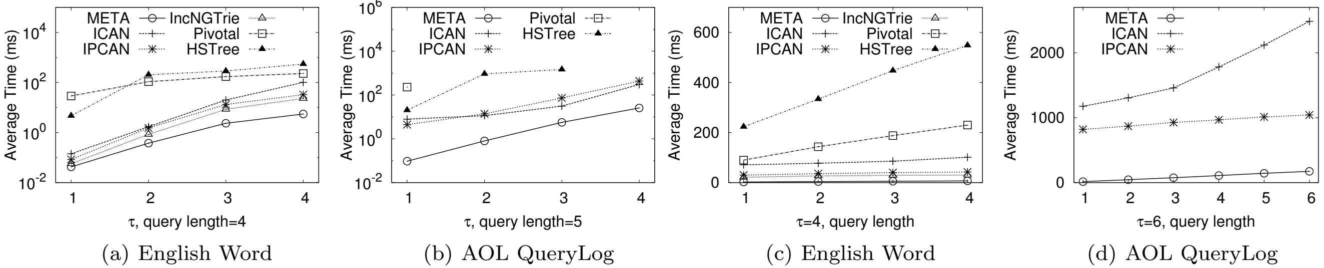
In this section we evaluated the compact tree. We implemented two methods Matching and Compact. Matching utilized the matching-based framework while Compact used compact tree to remove redundant computations. We first varied the threshold τ and fixed the query length of 4 and 5 for Word and Querylog respectively. We reported the average number of binary searches used by the two methods

[7] http://www.gregsadetsky.com/aol-data
[8] https://en.wikipedia.org/wiki/Wikipedia:Lists_of_common_misspellings



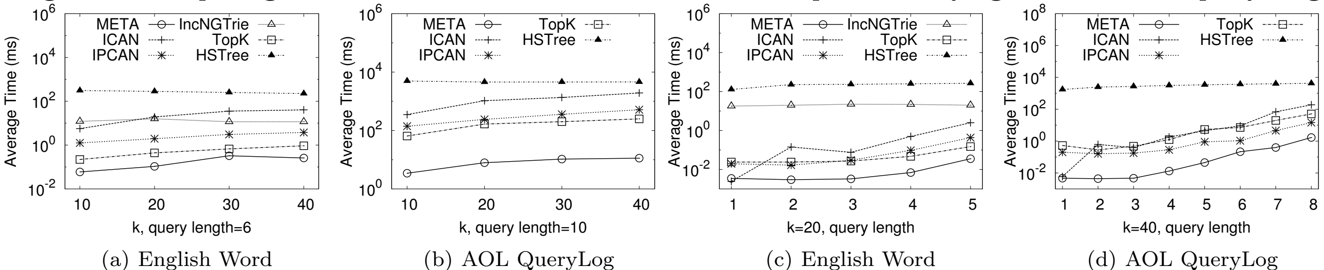
(a) English Word (b) AOL QueryLog (c) English Word (d) AOL QueryLog

Figure 6: Matching vs Compact: number of binary searches and avg. search time for threshold-based queries.



(a) English Word (b) AOL QueryLog (c) English Word (d) AOL QueryLog

Figure 7: Comparing with state-of-the-arts for threshold-based queries: varying threshold and query length.



(a) English Word (b) AOL QueryLog (c) English Word (d) AOL QueryLog

Figure 8: Comparing with state-of-the-arts for top-k queries: varying k and query length.

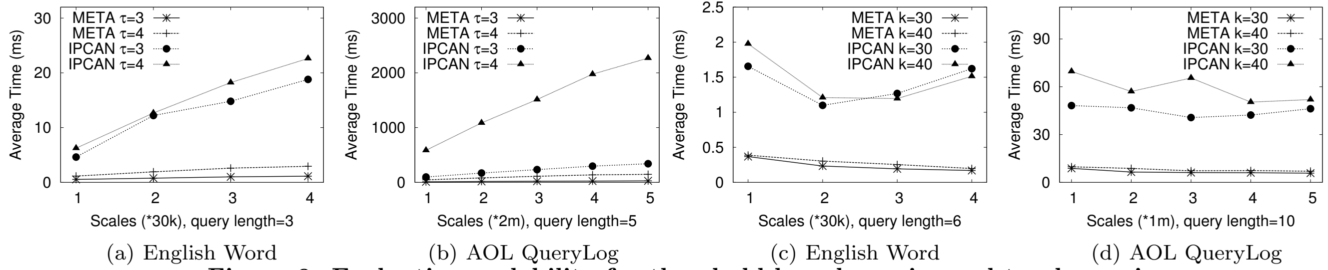
on the two datasets. Figure 6(a) and 6(b) show the results. We can see that Compact only took about one sixth binary searches of Matching. For example, on QUERYLOG dataset, for τ=4, Matching took about 130,000 binary searches for each query while Compact only took about 22,000 binary searches. This is because the compact tree can avoid a large number of redundant binary searches for the active matchings with same active node and for the active nodes with ancestor-descendant relationships. We also compared the average search time for the two methods. Figure 6(c) and 6(d) show the results. We can see that Compact outperformed Matching by 6 times. For example, on QUERYLOG, for τ=4, the average time for Matching and Compact were 150ms and 31ms respectively, because the search time depended on the number of binary searches (more than 80%) and Compact took fewer binary searches than Matching.

7.2 Comparison with State-of-the-art Methods

Threshold-Based Query. We compared META with ICAN [14], IPCAN [18], IncNGTrie [30], Pivotal [8] and HSTree [27] for threshold-based queries. We first varied the threshold τ and fixed the query length of 4 and 5 for WORD and QUERYLOG respectively. We reported the average search time and Figure 7(a) and 7(b) show the results. Note IncNGTrie took huge memory to store the deletion neighborhoods and active nodes, and it ran out of memory on QUERYLOG datasets. The similarity search methods Pivotal and HSTree could not finish in reasonable time on QUERYLOG dataset for large thresholds. META achieved the best performance and outperformed existing methods by an order of magnitude. For example, on WORD dataset, for $\tau = 4$, the average time for IncNGTrie, ICAN, IPCAN, META, Pivotal and HSTree were about 23ms, 78ms, 33ms, 5ms, 230ms and 548ms respectively. This is because our method META can save a lot of redundant binary searches compared with the state-of-the-art approaches. The string similarity search methods were slower as they had poor pruning power for extremely short queries and they generated huge number of prefixes. Note on QUERYLOG when $\tau = 1$, the average result set size for query prefixes with length 10 was around 87. It was 38 for query prefixes with length 6 on WORD when $\tau = 1$.

We then varied the query length and fixed τ as 4 and 6 for WORD and QUERYLOG respectively. We reported the average search time and Figure 7(c) and 7(d) show the results. META also achieved the best performance. For example, on WORD dataset, for query length of 3, the average search time for IncNGTrie, ICAN, IPCAN, META, Pivotal and HSTree were 22ms, 66ms, 31ms, 4ms, 188ms and 447ms respectively.

Top-k Query. We compared META with ICAN [14], IPCAN [18], IncNGTrie [30], HSTree [27], and TopK [9] for top-k queries. We reported the average search time by varying k and query length. Figure 8 shows the results. Note that the y-axles are log scale. We can see that META outperformed the other methods by 1-4 orders of magnitudes. For example, as shown in Figure 8(c), on WORD dataset, for k=20 and query length of 4, the average time for IncNGTrie, ICAN, IPCAN, META, HSTree and TopK were respectively 22ms, 0.50ms, 0.10ms, 0.007ms, 250ms and 0.047ms. This is because META can answer the top-k query incrementally. IncNGTrie was slower than others as it took more time for initialization. Our method outperformed the string similarity search methods as we shared the computations between the continuous queries typed in letter by letter while TopK and HSTree cannot. On QUERYLOG when k=10, there were 34% and 5% of the query prefixes with lengths 10 and 8 whose maximum prefix edit distance in their top-k results were no smaller than 3; there were 58% and 18% when $k = 40$.



(a) English Word (b) AOL QueryLog (c) English Word (d) AOL QueryLog

Figure 9: Evaluating scalability for threshold-based queries and top-k queries.

7.3 Scalability

We evaluated the scalability of our method. We used the same queries and varied the dataset sizes. Figure 9 shows the results for both threshold-based queries and top-k queries. For threshold-based queries, we used a fixed query length of 3 and 5 for WORD and QUERYLOG respectively and reported the average time for different thresholds. We can see that META scaled very well on the two datasets. For example, on WORD dataset, for $\tau = 4$, the average time for 30,000 strings, 60,000 strings, 90,000 strings and 120,000 strings were respectively 1.1ms, 1.9ms, 2.6ms and 2.9ms. This is because with the increase of dataset sizes, the size of the trie index only slightly increased as many strings shared common prefixes. We also evaluated the most efficient existing method IPCAN on the large dataset. IPCAN took more than 1 second on the QUERYLOG dataset with 4 million strings for $\tau = 4$ and could not meet the high-performance requirement. For top-k queries, we used a fixed query length of 6 and 10 for WORD and QUERYLOG respectively, and reported the average time for different k's. We can see that META still had good scalability for top-k queries. The average time slightly decreased as with the increases of dataset sizes, the maximum prefix edit distance for the top-k queries decreased while the number of active nodes slightly increased.

8. CONCLUSION

We study the threshold-based and top-k error-tolerant autocompletion problems. We propose a matching-based framework. To the best of our knowledge, this is the first study on answering top-k queries. We design a compact tree index to effectively maintain the active nodes. We propose an efficient method to incrementally answer the top-k queries. Experimental results showed our method significantly outperformed state-of-the-art methods.

Acknowledgements. This work was partly supported by the 973 Program of China (2015CB358700), NSF of China (61272090, 61373024, 61422205), Huawei, Shenzhou, Tencent, FDCT/116/2013/A3, MYRG105 (Y1-L3)-FST13-GZ, 863 Program (2012AA012600), and Chinese Special Project of Science & Technology(2013zx01039-002-002).

9. REFERENCES

[1] A. V. Aho. Algorithms for finding patterns in strings. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 255–300. 1990.

[2] A. Behm, S. Ji, C. Li, and J. Lu. Space-constrained gram-based indexing for efficient approximate string search. In *ICDE*, pages 604–615, 2009.

[3] F. Cai, S. Liang, and M. de Rijke. Time-sensitive personalized query auto-completion. In *CIKM*, pages 1599–1608, 2014.

[4] I. Cetindil, J. Esmaelnezhad, T. Kim, and C. Li. Efficient instant-fuzzy search with proximity ranking. In *ICDE*, pages 328–339, 2014.

[5] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD Conference*, pages 313–324, 2003.

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

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

[8] D. Deng, G. Li, and J. Feng. A pivotal prefix based filtering algorithm for string similarity search. In *SIGMOD Conference*, pages 673–684, 2014.

[9] D. Deng, G. Li, J. Feng, and W.-S. Li. Top-k string similarity search with edit-distance constraints. In *ICDE*, pages 925–936, 2013.

[10] H. Duan and B. P. Hsu. Online spelling correction for query completion. In *WWW*, pages 117–126, 2011.

[11] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[12] J. Guo, X. Cheng, G. Xu, and H. Shen. A structured approach to query recommendation with social annotation data. In *CIKM*, pages 619–628, 2010.

[13] S. Ji and C. Li. Location-based instant search. In *SSDBM*, pages 17–36, 2011.

[14] S. Ji, G. Li, C. Li, and J. Feng. Efficient interactive fuzzy keyword search. In *WWW*, pages 433–439, 2009.

[15] G. Li, D. Deng, J. Wang, and J. Feng. Pass-join: A partition-based method for similarity joins. *PVLDB*, 5(3):253–264, 2011.

[16] G. Li, J. Feng, and C. Li. Supporting search-as-you-type using sql in databases. *IEEE Trans. Knowl. Data Eng.*, 25(2):461–475, 2013.

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

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

[19] G. Li, J. Wang, C. Li, and J. Feng. Supporting efficient top-k queries in type-ahead search. In *SIGIR*, pages 355–364, 2012.

[20] Y. Li, A. Dong, H. Wang, H. Deng, Y. Chang, and C. Zhai. A two-dimensional click model for query auto-completion. In *SIGIR*, pages 455–464, 2014.

[21] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

[22] 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.

[23] S. B. Roy and K. Chakrabarti. Location-aware type ahead search on spatial databases: semantics and efficiency. In *SIGMOD*, pages 361–372, 2011.

[24] E. Sadikov, J. Madhavan, L. Wang, and A. Y. Halevy. Clustering query refinements by user intent. In *WWW*, 2010.

[25] S. K. Tyler and J. Teevan. Large scale query log analysis of re-finding. In *WSDM*, pages 191–200, 2010.

[26] E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64(1-3):100–118, 1985.

[27] J. Wang, G. Li, D. Deng, Y. Zhang, and J. Feng. Two birds with one stone: An efficient hierarchical framework for top-k and threshold-based string similarity search. In *ICDE*, pages 519–530, 2015.

[28] J. Wang, G. Li, and J. Feng. Trie-join: Efficient trie-based string similarity joins with edit-distance constraints. *PVLDB*, 3(1):1219–1230, 2010.

[29] S. Whiting and J. M. Jose. Recent and robust query auto-completion. In *WWW*, pages 971–982, 2014.

[30] C. Xiao, J. Qin, W. Wang, Y. Ishikawa, K. Tsuda, and K. Sadakane. Efficient error-tolerant query autocompletion. *PVLDB*, 6(6):373–384, 2013.

[31] C. Xiao, W. Wang, and X. Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB*, 1(1):933–944, 2008.

[32] Z. Zhang, M. Hadjieleftheriou, B. C. Ooi, and D. Srivastava. Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In *SIGMOD*, 2010.

[33] Y. Zheng, Z. Bao, L. Shou, and A. K. H. Tung. INSPIRE: A framework for incremental spatial prefix query relaxation. *IEEE Trans. Knowl. Data Eng.*, 27(7):1949–1963, 2015.