BEVA: An Efficient Query Processing Algorithm for Error-Tolerant Autocompletion

XIAOLING ZHOU and JIANBIN QIN, University of New South Wales
CHUAN XIAO, Nagoya University
WEI WANG, University of New South Wales
XUEMIN LIN, University of New South Wales and East China Normal University
YOSHIHARU ISHIKAWA, Nagoya University

Query autocompletion has become a standard feature in many search applications, especially for search engines. A recent trend is to support the *error-tolerant autocompletion*, which increases the usability significantly by matching prefixes of database strings and allowing a small number of errors.

In this article, we systematically study the query processing problem for error-tolerant autocompletion with a given edit distance threshold. We propose a general framework that encompasses existing methods and characterizes different classes of algorithms and the minimum amount of information they need to maintain under different constraints. We then propose a novel evaluation strategy that achieves the minimum active node size by eliminating ancestor-descendant relationships among active nodes entirely. In addition, we characterize the essence of edit distance computation by a novel data structure named *edit vector automaton* (EVA). It enables us to compute new active nodes and their associated states efficiently by table lookups. In order to support large distance thresholds, we devise a partitioning scheme to reduce the size and construction cost of the automaton, which results in the *universal partitioned EVA* (UPEVA) to handle arbitrarily large thresholds. Our extensive evaluation demonstrates that our proposed method outperforms existing approaches in both space and time efficiencies.

Categories and Subject Descriptors: H.3.3 [**Information Search and Retrieval**]: Search Process, Information filtering; F.1.1 [**Models of Computation**]: Automata

General Terms: Algorithm, Performance

Additional Key Words and Phrases: Edit distance, error-tolerant autocompletion, query processing, query optimization, edit vector automaton

ACM Reference Format:
Xiaoling Zhou, Jianbin Qin, Chuan Xiao, Wei Wang, Xuemin Lin, and Yoshiharu Ishikawa. 2016. BEVA: An efficient query processing algorithm for error tolerant autocompletion. ACM Trans. Database Syst. 41, 1, Article 5 (March 2016), 44 pages.
DOI: http://dx.doi.org/10.1145/2877201

5

X. Zhou, J. Qin, and W. Wang are supported by ARC DP130103401 and DP130103405. C. Xiao is supported by the FIRST Program, Japan. X. Lin is supported by NSFC61232006, DP150102728, and DP140103578. Y. Ishikawa is supported by JSPS Kakenhi (#25280039); in this grant, C. Xiao was also a team member.
Authors' addresses: X. Zhou, J. Qin, W. Wang, and X. Lin, School of Computer Science and Engineering, the University of New South Wales, Sydney, Australia; emails: {xiaolingz, jqin, weiw, lxue}@cse.unsw.edu.au; C. Xiao and Y. Ishikawa, Graduate School of Information Science, Nagoya University, Nagoya, Japan; emails: chuanx@nagoya-u.jp, ishikawa@is.nagoya-u.ac.jp.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2016 ACM 0362-5915/2016/03-ART5 $15.00
DOI: http://dx.doi.org/10.1145/2877201

1. INTRODUCTION

Nowadays, autocompletion has become a standard feature for search applications (especially for search engines). Not only does it reduce the number of keystrokes when a user launches a query, but also it reduces the possibility of erroneous queries and helps improve the throughput of the system as query or intermediate results can be effectively cached and reused.

Traditional autocompletion does not allow errors. Essentially, it performs a prefix search using the currently input query against a database of strings, typically from query logs or database records [Ji et al. 2009]. A recent trend is to allow a small amount of errors (usually controlled by an edit distance threshold τ) in the input query when performing the prefix matching [Chaudhuri and Kaushik 2009; Ji et al. 2009; Li et al. 2011, 2012]. For example, even if a user types in an incorrect prefix of Schwarzenegger such as Shwarz, Schwarzenegger may still be autocompleted or suggested if we allow one edit error in the prefix (underlined) matching [Chaudhuri and Kaushik 2009]. It is well known that misspelling is a common phenomenon for search engine queries, where 10% to 20% of the queries are found to be misspelt [Cucerzan and Brill 2004; Broder et al. 2009]. It has been found that error-tolerant autocompletion can save users' typing efforts by 40% to 60% [Ji et al. 2009], whereas typically no meaningful query completion will be provided if no error is allowed.

Query processing efficiency is an important aspect for this problem, as allowing errors drastically increases the inherent complexity of the problem. The system typically needs to handle many concurrent error-tolerant autocompletion queries against an ever-growing database of strings. Each query needs to be completed in no more than 100ms [Ji et al. 2009] to avoid noticeable delays during interactive search sessions. To address these performance challenges, most existing approaches [Ji et al. 2009; Chaudhuri and Kaushik 2009; Li et al. 2011] are based on indexing database strings in a trie and *incrementally* maintaining a set of trie nodes (called *active nodes*) for each input character, such that the query results can be readily computed once requested. More specifically, they maintain all nodes in the trie that have edit distances no larger than τ with respect to the current query, where τ is the given edit distance threshold. The number of active nodes is a dominant factor for the performance of the method, as both the amount of computation and memory consumption are proportional to the cumulative number of active nodes. The number can be as high as 300,000 for large U.S. address datasets with $\tau = 3$. Yet another deficiency in existing approaches is that there are plenty of ancestor-descendant relationships among active nodes due to the definition. This increases the complexity and overhead during incremental maintenance and query result reporting.

Realizing the importance of reducing the number of active nodes, Li et al. [2011] improve the previous work [Ji et al. 2009] by keeping only a carefully selected subset of active nodes. For the same U.S. address datasets and settings, it still needs to process over 80,000 active nodes, in addition to repeated visits of other trie nodes. Another recent approach [Xiao et al. 2013] achieves a substantial reduction in the active node size at the expense of a substantially larger index size. As a result, it is hard to be applied on large datasets like U.S. addresses as the index size is more than 20 times larger than the traditional trie index.

In this article, we provide a systematic study of the efficient query processing for error-tolerant autocompletion. We first characterize different classes of algorithms for the problem according to the properties they obey. We show the minimum-sized intermediate results any algorithm must maintain in each class. We characterize the correct conditions for identifying such nodes to be maintained in our algorithm based on the novel concept of *edit vectors*. In addition, we show that we can precompute an *edit vector automaton* (EVA) from all edit vectors with a given parameter of τ. This

enables us to both represent and incrementally update the edit vector of a node in $O(1)$ space and time. We propose a new algorithm, BEVA, that efficiently maintains just the *boundary active prefix set*, which is of the minimum size under reasonable constraints, incrementally with the help of the EVA. We also propose an algorithm to efficiently support different output options, as well as several optimizations to further speed up the execution. In order to reduce the transition number and the construction time of the automaton when τ is large, we propose a partitioning scheme (PEVA) that divides each edit vector into smaller edit vectors of a tunable length. In addition, we exploit the delta between the values in an edit vector and propose the notion of *universal partitioned EVA* (UPEVA) that supports arbitrarily large τs without constructing automata for predefined thresholds. Finally, we perform extensive experiments with state-of-the-art methods [Ji et al. 2009; Li et al. 2011; Xiao et al. 2013]. We demonstrate that our method outperforms existing methods in both space and query efficiencies. For the same U.S. address dataset, our BEVA algorithm processes less than 40,000 active nodes and achieves up to 11× and 6× speedups against ICAN and ICPAN, respectively.

Our contributions can be summarized as follows:

—We introduce a general framework for the problem and show the minimum set of information to maintain by any algorithm in this framework under different settings (Section 3).
—We propose the novel concepts of edit vectors and edit vector automata. They enable our algorithms to maintain the minimum amount of information and substantially improve the efficiency of our query processing algorithms (Section 4). EVA also compares favorably against state-of-the-art universal deterministic Levenshtein automata [Mihov and Schulz 2004] in several important aspects and may be of independent interest (Section 9).
—We propose a novel and efficient query processing technique for error-tolerant autocompletion, which achieves the minimum-sized active nodes by removing the ancestor-descendant relationships in existing approaches entirely (Section 5). Additional optimization techniques are also proposed to achieve further speedup in query processing and outputting results (Section 6).
—We devise a technique to cope with large edit distance thresholds by partitioning EVAs, and on top of that the universal partitioned EVA is proposed to handle arbitrarily large thresholds (Section 7).
—We demonstrate the superiority of our proposed method over the state of the art in our extensive experiments. The speedup is typically 4 to 10× faster than ICPAN [Li et al. 2011] and ICAN [Ji et al. 2009], respectively, and it outperforms IncNGTrie [Xiao et al. 2013] when the query string is short (Section 8).

2. PRELIMINARIES

2.1. Problem Definition

Let Σ be a finite alphabet of characters. A string d is an ordered array of characters drawn from Σ. An empty string is denoted as ϵ. $|d|$ denotes the length of d. $d[i]$ is the ith character of d, starting from 1. $d[i\,..\,j]$ denotes its substring starting from i and ending at j. Given two strings d and d', "$d' \preceq d$" denotes that d' is a prefix of d; that is, $d' = d[1..|d'|]$. Let $\mathcal{P}(d)$ denote all of d's prefixes; that is, $\mathcal{P}(d) = \{d[1\,..\,i] \mid 1 \le i \le |d|\}$.

Edit distance is a distance metric between two strings d and Q, denoted $ed(d, Q)$. It is the minimum number of operations, including insertion, deletion, and substitution of a character, to transform d to Q, or vice versa. We define the *prefix edit distance* between two strings as the minimum edit distance between any prefix of d and Q; that is, $ped(d, Q) = \min_{p \in \mathcal{P}(d)}\{ed(p, Q)\}$.

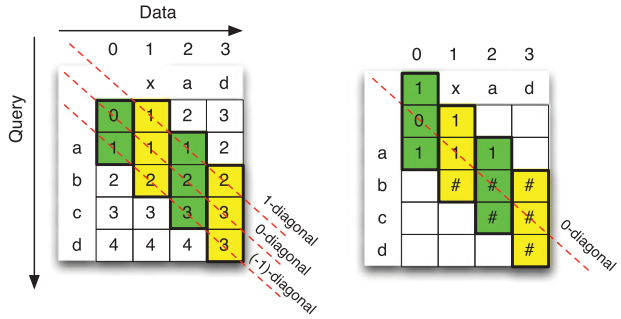


Fig. 1. Edit vector ($\tau = 1$).

Definition 2.1 (*Error-Tolerant Autocompletion*). Given a collection of data strings $\mathcal{D}$, a query string Q, and an edit distance threshold τ, the error-tolerant query autocompletion task is to (1) return all the strings $d \in \mathcal{D}$, such that their prefix edit distances to the query are no more than τ, and (2) be able to efficiently process the subsequent queries when additional characters are appended to Q.

We call the strings that satisfy the previous definition for the current query Q *qualified strings*, denoted as R_Q. There are several variations of the basic autocompletion task, notably (1) returning both the strings in R_Q as well as their prefix edit distances, and (3) returning only the top-k strings in R_Q (typically assuming a scoring function that is monotonic in both static scores of the strings and their prefix edit distances [Chaudhuri and Kaushik 2009; Ji et al. 2009; Li et al. 2012]). We focus on the basic definition, that is, returning all the qualified strings, and present details and experiment results with other output variants in Section 5.2 and Section 8.

2.2. Threshold Edit Distance Computation

The standard method to compute the edit distance between two strings d and Q (of length n and m, respectively) is the dynamic programming algorithm that fills in a matrix M of size $(n+1) \cdot (m+1)$. Each cell $M[i, j]$ records the edit distance between the prefixes of lengths i and j of the two strings, respectively.[1] The cell values can be computed in one pass in row-wise or column-wise order based on the following equation:

$$M[i, j] = \min(M[i-1, j-1] + \delta(d[i], Q[j]), \\ M[i-1, j] + 1, M[i, j-1] + 1), \quad (1)$$

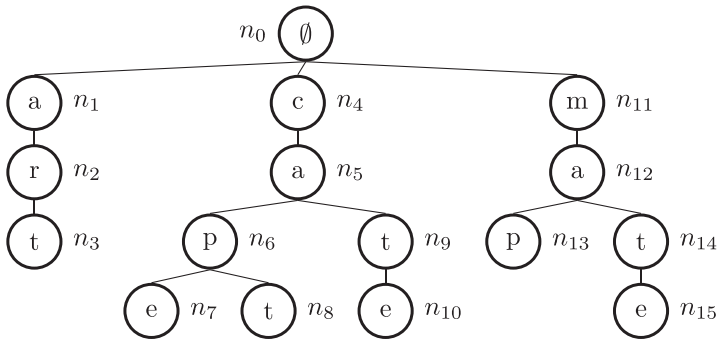
where $\delta(x, y) = 0$ if $x = y$, and 1 otherwise. The boundary conditions are $M[0, j] = j$ and $M[i, 0] = i$. The time complexity is $O(n \cdot m)$. In this article, we use the convention of placing the query string vertically and the data string horizontally in the matrix, as shown in Figure 1(a).

Define a k-diagonal of the matrix as all the cells $M[i, j]$ such that $j - i = k$, and it is well known that cell values on the same diagonal do not decrease (Theorem 2.2). To determine if the edit distance from d to Q is within τ, the *threshold edit distance algorithm* in Ukkonen [1985a] only needs to compute the k-diagonals of the matrix,

[1]For ease of exposition, the rows and the columns of the edit distance matrix M start from 0. All the other subscripts in the article start from 1. The prefix of length 0 of any string is an empty string.

ID	String
1	art
2	cape
3	capt
4	cate
5	map
6	mate

(a) Dataset strings



(b) Trie-based Index

Fig. 2. Example dataset.

where $k \in [-\tau, \tau]$, as shown in the green and yellow shaded area in Figure 1(a). The complexity is $O(\tau \cdot \min(n, m))$.

THEOREM 2.2 (LEMMA 2 IN UKKONEN [1985B]). $\forall M[i, j]$, $M[i, j] \geq M[i-1, j-1]$.

2.3. Overview of Existing Solutions

The prevalent approach to error-tolerant query autocompletion is based on incrementally maintaining a set of *active nodes* on a trie index built from the string data [Chaudhuri and Kaushik 2009; Ji et al. 2009]. This approach supports two essential operations: maintain and output. The data strings are indexed in a trie. During query processing, maintain *incrementally* maintains a set of prefixes of data strings that are within edit distance τ from the *current* query. The corresponding nodes in the trie are called *active nodes*. Whenever a new input character c is appended to the query, the new set of active nodes are computed based on c and the previous set of active nodes. The time complexity of each maintenance step is $O(\tau \cdot (|A| + |A'|))$ for Ji et al. [2009] and $O(|A| + |A'|)$ for Chaudhuri and Kaushik [2009], where $|A|$ and $|A'|$ are the numbers of active nodes before and after the maintenance, respectively. When results need to be reported, output returns the leaf nodes that can be reached from current active nodes as resulting strings.

There are two main issues with these approaches. The first lies in the large number of active nodes, which can be up to $O((|Q| + \tau)^{\tau} \cdot |\Sigma|^{\tau})$. The cost of carrying out maintenance of active nodes hence is large. To alleviate the issue, the notion of *pivotal* active nodes was proposed [Li et al. 2011] to only consider the subset of active nodes with the last characters being neither substituted nor deleted; in other words, the last character reaching the node must be a match in an alignment that yields the edit distance between the query and the prefix. Nevertheless, the number of (pivotal) active nodes can still be up to $O((|Q| + \tau - 1)^{\tau} \cdot |\Sigma|^{\tau})$. The second issue is that the output is made complex due to the existence of ancestor-descendant relationships among active nodes. For example, when only qualified strings need to be output, duplicate elimination is required when results need to be reported, because if an active node is an ancestor of another, it subsumes the string IDs under the latter. This procedure is costly yet has not been well studied.

Example 2.3. Consider a trie of six data strings (Figure 2(b)), a query `cat`, and $\tau = 1$. Table I shows the active nodes maintained by the ICAN algorithm [Ji et al. 2009] and the edit distances from the query to the prefixes they represent. Each active node represents a prefix whose edit distance to the current query is within τ. For example,

Table I. Example of Running the Algorithm in Chaudhuri and Kaushik [2009]

Step	Query	Active Nodes & Their Edit Distances
1	∅	$\{n_0, 0\}, \{n_1, 1\}, \{n_4, 1\}, \{n_{11}, 1\}$
2	c	$\{n_0, 1\}, \{n_1, 1\}, \{n_4, 0\}, \{n_5, 1\}, \{n_{11}, 1\}$
3	ca	$\{n_1, 1\}, \{n_4, 1\}, \{n_5, 0\}, \{n_6, 1\}, \{n_9, 1\}, \{n_{12}, 1\}$
4	cat	$\{n_5, 1\}, \{n_6, 1\}, \{n_8, 1\}, \{n_9, 0\}, \{n_{10}, 1\}, \{n_{14}, 1\}$

when $Q = \texttt{cat}$, $ed(\texttt{ca}, Q) = 1$ for n_5, $ed(\texttt{cap}, Q) = 1$ for n_6, and $ed(\texttt{mat}, Q) = 1$ for n_{14}. As can be seen, (1) a trie node can be an active node in several steps (e.g., n_0), and (2) there are plenty of ancestor-descendant relationships among active nodes (e.g., n_5 and n_9 in Step 4, both reaching the same data string cate).

There are also techniques that speed up the query processing by using additional space. Chaudhuri and Kaushik [2009] propose to partition all possible queries at a certain length into a limited number of equivalent classes (via reduction of the alphabet size) and precompute the answer active nodes for all these classes. Most recently, Xiao et al. [2013] proposed to build a trie for all the τ-deletion variants of the data strings and process the query by a simple matching procedure. Since this approach is based on the neighborhood enumeration method, there is no need to calculate edit distance for intermediate nodes as other trie-based methods do. However, it builds a substantially larger index (e.g., 15.9× larger on MEDLINE) to achieve good speedup and may not work when the edit distance threshold is large or space is limited, as evidenced in our experiment.

3. A PREFIX-BASED FRAMEWORK FOR EDIT PREFIX SEARCH

In this section, we develop a general framework to process error-tolerant autocompletion queries that encompasses most of the existing solutions and our proposed solution. We identify the minimal number of prefixes to be maintained by any scheme in this framework subject to certain constraints. This motivates us to develop a scheme that maintains a minimal number of prefixes with the help of edit vector automata in Section 5.

The Prefix-Based Framework. We consider a framework based on the set of *prefixes* produced from all data strings. Although these prefixes can be mapped to nodes of a trie, the framework is generic and independent of any physical implementation.

Given a query Q, a dataset $\mathcal{D}$, and an edit threshold τ, an algorithm in the framework produces and maintains a set of prefixes of data strings $P_Q \subseteq \mathcal{P}(\mathcal{D})$. It supports two basic operations: one is to output the query result, and the other is to incrementally maintain the prefix set when a new character is appended to the current query. Hence, the framework can be summarized by the following three phrases:

—*Indexing*: A trie is usually used as the index structure to organize the data strings to support fast prefix matching. Each node is associated with a range indicating the first and the last strings that share the prefix represented by the node.
—*Maintenance*: At each step, the framework maintains a set of prefixes P_Q for the query Q. When a new character c is appended to Q (denote the new query as Q'), the new prefix set $P_{Q'}$ can be incrementally calculated based solely on P_Q and c.
—*Result Fetching*: For each prefix ρ in P_Q, we return all strings that have ρ as a prefix.

Obviously, the following two conditions are necessary to ensure that the *result fetching* outputs the correct results:

—**C1** [Completeness] $\forall d \in R_Q, \mathcal{P}(d) \cap P_Q \neq \emptyset$
—**C2** [Soundness] $\forall d \notin R_Q, \mathcal{P}(d) \cap P_Q = \emptyset$

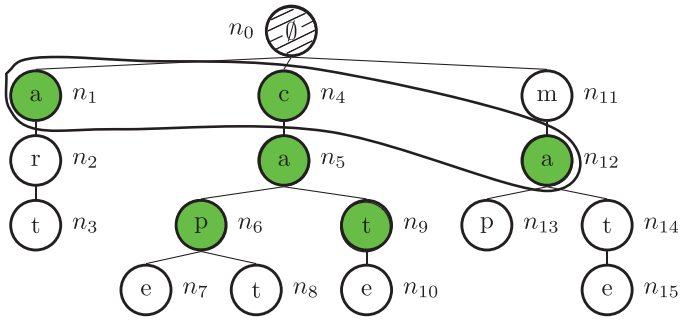


Fig. 3. Consider the query ca and the trie in Figure 2(b). The active prefix set is $\{n_1, n_4, n_5, n_6, n_9, n_{12}\}$, the boundary active prefix set is $\{n_1, n_4, n_{12}\}$, and the compressed boundary active prefix set is $\{n_0\}$.

If P_Q satisfies both conditions **C1** and **C2**, we call it an *answer prefix set*, denoted by A_Q.

In addition to the aforementioned two necessary conditions, we define the following condition, which enables us to efficiently output the results from prefixes maintained in P_Q:

—**C3** [Validity] $\forall \rho \in P_Q, ed(\rho, Q) \leq \tau$

All the existing algorithms satisfy **C3**. For example, both Chaudhuri and Kaushik [2009] and Ji et al. [2009] maintain a subset of the *active prefix set* (defined later) and it is obvious that it, and any of its subsets, satisfies **C3**.

Definition 3.1 (*Active Prefix Set* $\mathcal{V}_Q$). $\mathcal{V}_Q = \{\rho \mid \rho \in \mathcal{P}(\mathcal{D}) \wedge ed(\rho, Q) \leq \tau\}$.

Minimum Answer Prefix Sets. While the algorithm in Chaudhuri and Kaushik [2009] and ICAN [Ji et al. 2009] maintains exactly the $\mathcal{V}_Q$, the improved algorithm ICPAN [Ji et al. 2009] maintains a subset of $\mathcal{V}_Q$. As a result, the ICPAN algorithm achieves better efficiency both in terms of space and time complexities.

It is natural to ask what is the minimum answer prefix set an algorithm must maintain for the error-tolerant autocompletion problem. We answer this question in Theorems 3.3 and 3.4 later.

Given a $\mathcal{V}_Q$, it may contain two prefixes ρ and ρ' such that $\rho' \preceq \rho$. If we remove all such ρs repeatedly, we can obtain the **Boundary Active Prefix Set**, that is:

Definition 3.2 (*Boundary Active Prefix Set* $\mathcal{B}_Q$). $\mathcal{B}_Q = \{\rho \mid \rho \in \mathcal{V}_Q \wedge (\nexists \rho' \in \mathcal{V}_Q \wedge \rho' \preceq \rho)\}$.

Given a boundary active prefix set $\mathcal{B}_Q$, we can *repeatedly* apply two *compress* operators (defined later) to obtain a **compressed boundary prefix set** $\mathcal{B}^*_Q$. We first define the *parent* of a prefix ρ as ρ' such that $\rho' \prec \rho$ and $|\rho'| = |\rho| - 1$. Alternatively, we call ρ a *child* of ρ'. If ρ is the only child of ρ', ρ' is ρ's *sole parent*. The first compress operator replaces ρ with ρ' if ρ' is ρ's sole parent and ρ' does not correspond to the complete data string, and the second compress operator replaces all of ρ's children with ρ if all of ρ's children are in the active prefix set $\mathcal{V}_Q$. An example is given in Figure 3 to showcase the active prefix set, the boundary active prefix set, and the compressed boundary active prefix set.

The following two theorems show that $\mathcal{B}_Q$ and $\mathcal{B}^*_Q$ are the smallest answer prefix sets for all algorithms satisfying conditions **C1**, **C2,** and **C3** and conditions **C1** and **C2**, respectively.

THEOREM 3.3. *Boundary active prefix set* $\mathcal{B}_Q$ *is the smallest answer prefix set that satisfies* ***C1, C2,*** *and* ***C3****.*

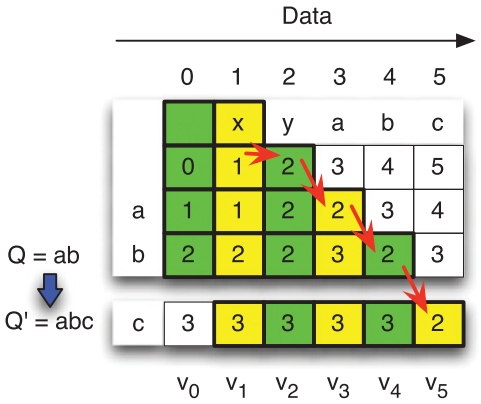


Fig. 4. Motivation for edit vectors ($\tau = 2$).

THEOREM 3.4. *Compressed boundary active prefix set $\mathcal{B}^*_Q$ is the smallest answer prefix set that satisfies **C1** and **C2**.*

Discussion. Although the compressed boundary active set is guaranteed to be no larger than the boundary active set, it is unlikely that it can be efficiently maintained in an incremental manner due to repeated visit and computation of edit distances in the descendant nodes. Therefore, considering the additional condition **C3** let us focus on the subclass of algorithms that can process the query efficiently in an incremental manner.

Note that the size of the prefix set maintained by an algorithm directly affects its space and time efficiency. A smaller prefix set reduces the amount of traffic between the client and the server at each step [Ji et al. 2009]. It also contributes directly to the maintenance cost. The ICPAN method can be seen as benefiting from reducing the size of the prefix set by maintaining only the *pivot active prefix set*. Theorem 3.3 points out that the minimum prefix set is the boundary active prefix set, and this observation drives us to design efficient algorithms that maintain only the boundary active prefix set (see Section 5).

4. EDIT VECTORS AND EDIT VECTOR AUTOMATA

As we analyzed in Example 2.3, the root problem that causes much overhead in existing solutions is due to their definition of active nodes, which inherently allows ancestor-descendant relationships among active nodes. The essential reason for keeping such redundancy in these methods is to ensure that edit distance information can be *easily* and correctly passed on to the descendant node.

For example, consider the example in Figure 4 with $\tau = 2$ and the current query Q is `ab`. Active nodes on the path `xyabc` are {$\emptyset$, `x`, `xy`, `xyab`}. The inclusion of `xyab` as an active node makes it easy to compute the edit distance of `xyabc` according to Equation (1).

Now what if we keep only the top-most nodes for the current query as active nodes? In this example, we will keep only root node $\emptyset$ (and its current edit distance, which is 2). Consider that the query becomes $Q' =$ `abc`. Root node $\emptyset$ and its only child node `x` both have an edit distance of more than 2 and will no longer be active nodes. However, one of its descendant nodes, `xyabc`, should be in the active node. Since we have not kept `xyab` and its edit distance to Q in the previous active node set, we have to perform costly and nontrivial computation of such information.

Our key idea to solve this difficulty is that, if we keep for each node all its edit distance values between its $(-\tau)$- and τ-diagonals, we can guarantee that all the query

results can be computed correctly. In the previous example, these values for xy with respect to Q' are $[2, 2, 2, 3, \#]^T$ (shown in the green cells in Column 2).[2] By running the threshold edit distance algorithm (Section 2.2) in column-wise order, we can still compute the cells in the subsequent columns between the $\pm\tau$-diagonals and finally obtain the edit distance between the Q' and xyabc.

In the rest of this section, we formalize this idea as *edit vector*s and show that it can be encoded as a state in an *edit vector automaton*. This not only speeds up the active node maintenance by a factor of $O(\tau)$ but also enables us to maintain only the minimum subset of active nodes (see Section 5).

4.1. Edit Vectors

A *raw edit vector* v_j with respect to τ is a column vector of size $2\tau + 1$ at the jth column of M such that $v_j[i] = M[j - \tau - 1 + i, j]$; that, it is centered around the 0-diagonal. When applying the threshold edit distance algorithms of τ, it is unnecessary to keep the actual value of cells whose value is larger than τ. Therefore, we replace those cells with a special symbol # to generate the *edit vector*. We also define three natural rules regarding the essential computations on #: (i) $\tau + 1 = \#$, (ii) $\# + 1 = \#$, and (iii) $\# > \tau$.

Example 4.1. Consider $\tau = 1$. We show all the *raw edit vectors* and *edit vectors* $(0 \le j \le 3)$ in the matrix in Figures 1(a) and 1(b), respectively. We use alternating green and yellow shades to visually distinguish two adjacent vectors. The out-of-boundary elements and those larger than τ are appropriately initialized and marked by #, respectively. Finally, we have four edit vectors: $[1, 0, 1]^T$, $[1, 1, \#]^T$, $[1, \#, \#]^T$, and $[\#, \#, \#]^T$ in this example.

Note that the edit vector of column 0 in any matrix is always the same in the form of $[\underbrace{\tau, \tau - 1, \ldots, 1}_{\tau}, 0, \underbrace{1, 2, \ldots, \tau}_{\tau}]^T$, as the data string at column 0 is an empty string. We name such a vector the ***initial edit vector***. Similarly, the vector with all #, that is, $[\underbrace{\#, \#, \ldots, \#}_{2\tau+1}]^T$, is named the ***terminal edit vector***.

4.2. From Edit Vectors to Edit Vector Automata

It is straightforward to compute the subsequent edit vector from the current one, given the query and the next character in the data string. Nevertheless, the significance of edit vectors mainly lies in the fact that their transitions can be precomputed and hence collectively form an automaton.

4.2.1. Edit Vector Computation. With the definition of edit vectors, it can be easily verified that the threshold edit distance computation in column-wise fashion is essentially computing the subsequent jth edit vector starting from $j = 0$. Finally, the edit distance between Q and a data string d is either $v_{|d|}[\tau + 1 + (|Q| - |d|)]$ when $|Q| \in [|d| - \tau, |d| + \tau]$ or more than τ otherwise.

Given the edit vector v_j at the jth column, our job is to compute the subsequent edit vector v_{j+1}. The computation is performed from the top-row cell to the bottom cell of the column, using the equation adapted from Equation (1) (see Figure 5):

$$v_{j+1}[i] = \min(v_j[i] + \delta(d[j + 1], Q[j - \tau + i]),\\ v_j[i + 1] + 1, v_{j+1}[i - 1] + 1), \quad \forall 1 \le i \le 2\tau + 1.$$

[2]Here, we use the special symbol # to denote out-of-boundary cell values.

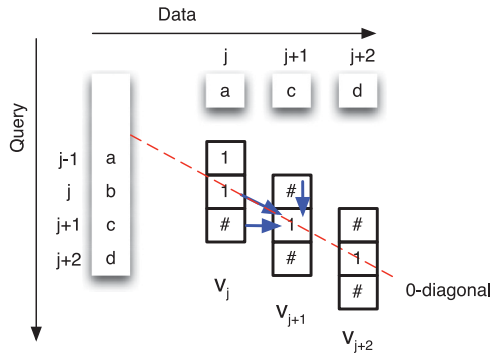


Fig. 5. Edit vector transition ($\tau = 1$).

4.2.2. Precomputation of All Edit Vectors. First, we observe that there are only a limited number of possible edit vectors when τ is small. Let $\mathcal{V}(\tau)$ denote the set of all possible edit vectors with respect to τ.

LEMMA 4.2. $2.25 \cdot 2^{2\tau} \leq |\mathcal{V}(\tau)| \leq 2 \cdot 3^{2\tau}$.

PROOF. We first prove the upper bound. The length of the edit vector with respect to τ is $2\tau + 1$. Consider its first value. As it belongs to the $(-\tau)$-diagonal, its value must be at least τ. Hence, there are only two choices: τ or #.

It is well known that any two adjacent cells in an edit distance matrix differ at most by 1 [Masek and Paterson 1980]. Therefore, for any edit vector v, we can equivalently rewrite it to $[v[1], v[1]+\zeta_1, \ldots, v[2\tau]+\zeta_{2\tau}]^T$, where $\zeta_i \in \{-1, 0, 1\}$. Obviously, the distinct number of rewritten sequences only depends on $v[1]$ and ζ_i. Hence, we have $|\mathcal{V}(\tau)| \leq 2 \cdot 3^{2\tau}$.

We then prove the lower bound. Let $g(\tau) = |\mathcal{V}(\tau)|$. For any edit vector v, we can construct a vector for threshold $\tau + 1$ by adding either $\tau + 1$ or # to the beginning and the end of v. Hence, $g(\tau + 1) \geq 2^2 \cdot g(\tau)$. We can easily find $g(1) = 9$ by enumeration. Hence, $g(\tau) \geq 9 \cdot 2^{2\tau-2}$, from which the lower bound stated in the lemma can be easily obtained. □

In practice, we find that $|\mathcal{V}(\tau)|$ is much smaller than the upper bound, as shown in Table VIII in Section 8.11.

The other key observation is that, since v_{j+1} can be computed from v_j and other inputs, we can think of v_{j+1} as a *function* of the following inputs: (i) v_j, (ii) $d[j+1]$, and (iii) $Q[(j-\tau+1)..(j+\tau+1)]$.

For a fixed alphabet Σ, we can precompute the results of the function for every possible combination of input values and store them in a table. We define the number of distinct states as $|\mathcal{V}(\tau)|$, and the total number of entries in the precomputed tables will be $|\mathcal{V}(\tau)| \cdot |\Sigma|^{2\tau+2}$. As such, this bound can be deemed as a special case of the bound obtained in the Four-Russians technique [Masek and Paterson 1980].

Due to the dependency on the alphabet size, the table will be enormous for an English alphabet of at least 26 even for small τ—the number of entries will be 6.74×10^{13} for $\tau = 3$. Fortunately, we observe that no matter what characters $d[j+1]$ and $Q[j-\tau+i]$ are, it is whether they match or not that affects the resulting edit vector. Consequently, we can model the computation of the subsequent edit vector as a *function* on a different set of input parameters. Specifically, we model v_{j+1} as $f(v_j, B)$, where B is a binary bitmap of $2\tau + 1$ bits, and its ith bit, $B[i]$, is $\neg\delta(d[j+1], Q[j-\tau+i])$, for $1 \leq i \leq 2\tau + 1$.

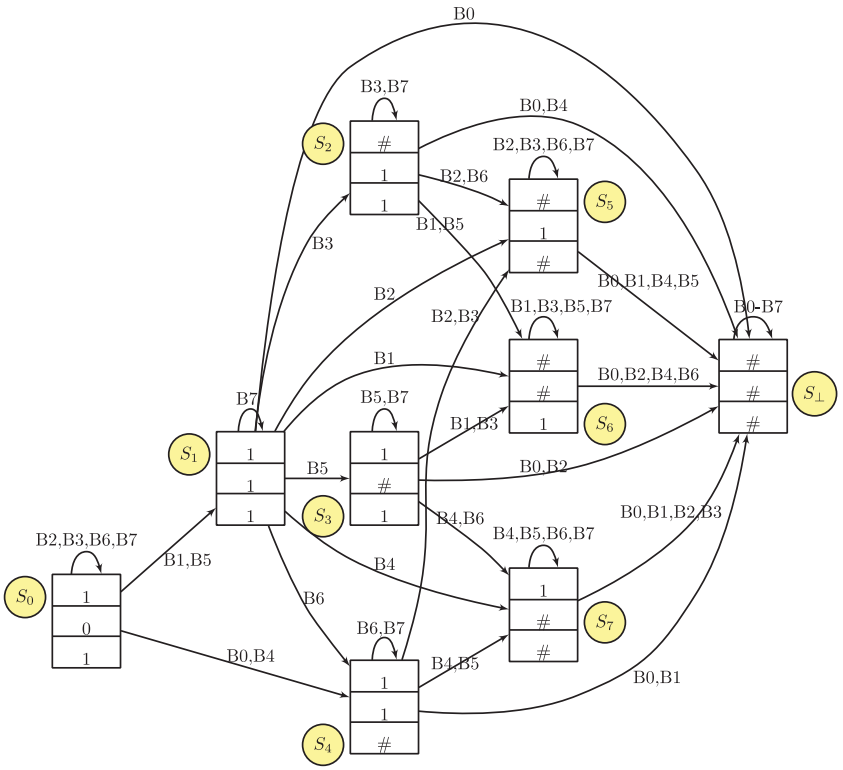


Fig. 6. Edit vector automata.

Example 4.3. Consider the transition from vector v_j to vector v_{j+1} in the example of Figure 5. Since $\delta(\mathtt{c}, \mathtt{b}) = 1$, $\delta(\mathtt{c}, \mathtt{c}) = 0$, and $\delta(\mathtt{c}, \mathtt{d}) = 1$, the bitmap $B = \mathtt{010}$. So we have $f([1, 1, \#]^T, \mathtt{010}) = [\#, 1, \#]^T$.

The total number of precomputed table entries will be upper bounded by $|\mathcal{V}(\tau)| \cdot 2^{2\tau+1}$ and hence does *not* depend on the size of the alphabet Σ—the number of entries is only 41, 344 for $\tau = 3$. It also means the precomputation only depends on τ and is **universal** with respect to the alphabet or the strings—the $f(\cdot, \cdot)$ function characterizes all the *subsequent* transitions between edit vectors.[3] If we model each edit vector as a state, and $f(\cdot, \cdot)$ as the transition function, we can construct the following automaton, named *edit vector automaton,* to characterize *all* the computation on edit vectors.

Definition 4.4. An edit vector automaton with respect to τ is a 5-tuple $(\mathcal{S}, \mathcal{B}, f, \{S_0\}, \{S_\perp\})$, where $\mathcal{S}$ is the set of states and each state is associated with a unique edit vector, $\mathcal{B} = \{\mathtt{0}, \mathtt{1}\}^{2\tau+1}$ is the set of all bitmaps of length $2\tau + 1$ that drives the transition of states, f is the edit vector transition function discussed earlier, $S_0 \in \mathcal{S}$ is the only *initial state* associated with the *initial edit vector*, and $S_\perp \in \mathcal{S}$ is the only *terminal state* associated with the *terminal edit vector*.

We show the edit vector automaton for $\tau = 1$ in Figure 6. We represent each state using its associated edit vector and number the state as S_i in the yellow shaded circle

[3]In Section 7, we present a method to make the automaton truly universal and hence it can be built and work for any τ.

placed nearby. S_0 and $S_\perp$ are the initial and terminal states, respectively. We represent the B values as integers on the edges. For example, the transition from S_0 to S_1 is via an edge labeled with $B1$ and $B5$, which represent `001` and `101`, respectively.

It is well known that any two adjacent cells in an edit distance matrix differ at most by 1 [Masek and Paterson 1980]. Therefore, we have the following property of edit vectors:

LEMMA 4.5. *The values of two adjacent cells in an edit vector differ by at most 1.*

4.2.3. Precomputation of Edit Vector Automata (EVA). We give a simple algorithm to construct the automaton, which starts from the initial state and keeps growing the automaton by feeding all possible B values to each newly discovered state. More specifically:

(1) We push the initial state S_0 associated with the initial vector into an empty queue.
(2) While the queue is not empty, we pop one state S_i from the queue. We compute $S' = f(S_i, B)$ for each of the $2^{2\tau+1}$ possible values of B (i.e., from 0 to $2^{2\tau+1} - 1$). Record these transitions in the automaton. Finally, if S' is a new state, we push it into the queue.

Note that each new state is pushed into the queue exactly once. For each state popped from the queue, we compute $2^{2\tau+1}$ transitions, each taking $O(\tau)$ time. Testing if a state is new takes $O(1)$ time. Together with the upper bound on the number of states in Lemma 4.2, the complexity of the algorithm is $O(\tau \cdot 6^{2\tau})$. The actual running time is less than 0.3 second for τ up to 4 (see Section 8.11).

5. BOUNDARY QUERY PROCESSING WITH EDIT VECTOR AUTOMATA

We introduce a new algorithm, named BEVA, that maintains as *active nodes* the *boundary active prefixes* with the help of edit vector automata. As a result, the algorithm maintains the minimum amount of active nodes among all algorithms satisfying **C1**, **C2**, and **C3** due to Theorem 3.3. We detail the implementation of the maintain and output methods, followed by an optimization to avoid repeated bitmap construction.

5.1. The Maintenance Algorithm

In our BEVA algorithm, we choose to maintain as "active nodes" exactly those *boundary active prefixes*, as defined in Section 3. Hence, we do not disambiguate these two terms in the rest of the section.

Recall from the definition (of the boundary active prefix set) that an active node itself satisfies the edit distance constraint with the current query, and none of its prefixes (or ancestors in the trie) satisfies the edit distance constraint.

An interesting consequence is that if n is an active node of the current query Q, then it cannot be an active node of the subsequent query Q' (Lemma 5.1), and only its first descendant nodes in paths of the subtrie that satisfy the edit distance constraint will be active nodes of Q' (Lemma 5.2). Therefore, the straightforward way to maintain the active nodes when the query is updated to Q' is to start with an empty set and, for each node in the current active node set, compute the edit distances of each of its child nodes. There are two outcomes: (1) for the nodes that satisfy the edit distance constraint, we mark them as active nodes of Q', and (2) for the other nodes, we recursively examine their next-level child nodes.

LEMMA 5.1. *For a given τ, none of the active nodes of a query Q is an active node of the query $Q' = Q \circ c$, where $\circ$ denotes a concatenation, and c is an appended character.*

LEMMA 5.2. *For a given τ, a node n is an active node of a query $Q' = Q \circ c$ if and only if (1) one of n's ancestors is an active node of the query Q, (2) none of its ancestor nodes*

ALGORITHM 1: Maintain($\mathcal{A}$, Q, c)

```
   Input: Q is the last query, and Q' = Q ∘ c is the current query
1  Initialize(Q, c);
2  if |Q| = 0 then
3      A ← ⟨r, S_0⟩;                    /* r is the root of the trie */;
4  else if |Q| ≥ τ then
5      A' ← ∅;
6      for each ⟨n, S⟩ in A do
7          A' ← A' ∪ FindActive(c, ⟨n, S⟩);
8      A ← A';
9  return A;
```

ALGORITHM 2: FindActive(c, $\langle n, S\rangle$)

```
1  A ← ∅;
2  for each child n' of n do
3      k ← |Q'| − n'.len;  /* n'.len is the length of n''s */ /* corresponding prefix */;
4      B_n' ← BuildBitmap(Q', n');
5      S' ← f(S, B_n');
6      if S' ≠ S_⊥ then
7          if S'[τ + 1 + k] ≤ τ then
8              A ← A ∪ ⟨n', S'⟩;
9          else
10             A ← A ∪ FindActive(c, ⟨n', S'⟩);
11 return A;
```

is an active node of the query Q', and (3) it satisfies the edit distance constraint with respect to Q'.

In the following, we discuss the details of implementing the maintenance algorithm leveraging edit vector automata.

We use the edit vector automaton to drive the traversal on the trie. Hence, an active node n of Q is always associated with a state S in the edit vector automaton and is represented by a pair $\langle n, S\rangle$. Initially, the only active node is the root node of the trie, associated with the initial state S_0.

The pseudo-code of maintain is shown in Algorithm 1. Initially, we start from the root node of the trie, which is associated with the initial state of the edit vector automaton (Line 3). The algorithm does nothing for the first τ characters of the input query. When the length of current query $|Q'| \geq \tau$ (Line 4), we compute the set of *active nodes* for it.

Once a new character c is appended to Q to form new query Q', we need to construct a new active node set from the descendants of current active nodes. Hence, we next iterate all the current active nodes and call Algorithm 2 to traverse its descendants. In Algorithm 2, for each active node represented in $\langle n, S\rangle$, it expands its children, and for each child node n' of n, it calls the BuildBitmap function to construct the corresponding bitmap $B_{n'}$ necessary to drive the automaton. Then, the new active state S' is computed using S and $B_{n'}$ via the edit vector automaton (Line 5). Finally, (1) no further action will be taken if S' is a terminal state, (2) $\langle n', S'\rangle$ is added into the new active node set $\mathcal{A}'$ if it satisfies the edit distance constraint (Line 8), or (3) Algorithm 2 is called recursively to look for active nodes amid the descendants of n' (Line 10), otherwise.

In order to check if the new node n' meets the edit distance constraint, we compare τ with the edit distance between Q' and the prefix ending at n'. The latter is captured by the $\tau + 1 + k$th element of the state S', that is, the value on the $(-k)$-diagonal of the edit distance matrix, where $k = |Q'| - n'.len$ (Line 3). Take Figure 5 as an example, assuming $Q = \ldots$abc and the node n ending with a is an active node with the state $[1, 1, \#]^T$. When the query changes to $Q' = \ldots$abcd, the state of its child node n' (ending with ac) can be computed to be $[\#, 1, \#]^T$. Since the last cell value that indicates the edit distance between Q' and n' is larger than the threshold, we need to recursively test n''s child nodes. In this example, the child node ending with acd satisfies the edit distance constraint and hence becomes an active node for Q'.

The function BuildBitmap is supposed to compare n''s label with each character of $Q'[(|n'| - \tau)..(|n'| + \tau)]$. Since the function FindActive can be called recursively, it is possible that query characters beyond the current length of Q' may need to be accessed. In this case, we set these bits to 0. We will show a more efficient way to obtain the bitmap in Section 5.1.1.

The cost of each invocation of Algorithm 1 is $O(\tau + |\mathcal{A}| + |\mathcal{A}'|)$. If we run it for every character of the query of length $|Q|$, the total cost is $O(|Q| \cdot \tau + M), M = \min(N, O(|Q|^{\tau+1}|\Sigma|^{\tau}))$, where N is the total number of nodes in the trie and $|Q|^{\tau+1}|\Sigma|^{\tau}$ is the upper bound of total neighborhood size of each prefix of query Q in terms of alphabet Σ and threshold τ. This compares favorably with the existing methods where the total cost is $O(\tau \cdot M)$ [Chaudhuri and Kaushik 2009] or $O(\tau^2 \cdot M)$ [Ji et al. 2009].

5.1.1. Maintaining Global Bitmaps. An inefficiency with the previous algorithm is in the repeated construction of bitmaps for descendant nodes of the active nodes of the last query. The cost per node is $\Theta(\tau)$ time. In the following, we present a technique to reduce the processing time per child node to $O(1 + \frac{\tau}{C})$, where C is the total number of descendant nodes of all the current active nodes considered in an invocation of FindActive. This complexity is $O(1)$ when $C \gg \tau$, which is usually the case.

Our idea is to reuse the bitmaps among all nodes encountered. A key observation is that given the last $2\tau + 1$ characters of the current query Q, there are at most $2\tau + 2$ distinct bitmaps[4] regardless of which node and which of its child nodes we are processing. Define the *effective query window*, $w(Q)$, as the last $2\tau + 1$ characters of Q. There are at most $2\tau + 1$ distinct characters in $w(Q)$. Each of them will lead to a distinct bitmap, and this bitmap will be used if a child node's label matches it. For the case when the child node's label does not match any character in $w(Q)$, the bitmap is all 0s. We call these $2\tau + 2$ bitmaps the *global bitmaps* for the current query. To use these global bitmaps, we only need to additionally have a dynamic table $\mathcal{H}$ that maps a character to one of these bitmaps in $O(1)$ time.

To use the global bitmaps, we make the following changes to Algorithm 1:

—Assume there is a global hash table $\mathcal{H}$.
—The Initialize function (Line 1 of Algorithm 1) will call the UpdateBitmap function (see Algorithm 3).
—The BuildBitmap function will be implemented as Algorithm 4.

Algorithm 3 gives the pseudo-code of the algorithm to update the global bitmap table $\mathcal{H}$ upon each increment of the query. Supposing $\tau = 1$ and the last $2\tau + 1$ characters of Q are abc, the dynamic table $\mathcal{H}$ maps a to 100, b to 010, and c to 001, and all the other characters to 000. If a new character d is appended to query Q, we will delete entry $\mathcal{H}$(a) (Line 8) and update $\mathcal{H}$(b) and $\mathcal{H}$(c) by shifting 1 bit leftwards (Line 6) and then add $\mathcal{H}$(d) = 001 to $\mathcal{H}$ (Line 9).

[4]Assume without loss of generality that $|\Sigma| \geq 2\tau + 2$.

Table II. BEVA0 Example

Step	Query	Active Node Set $\mathcal{A}$
1	$\emptyset$	$\{\langle n_0, S_0\rangle\}$
2	c	$\{\langle n_0, S_0\rangle\}$
3	ca	$\{\langle n_1, S_1\rangle, \langle n_4, S_0\rangle, \langle n_{12}, S_5\rangle\}$
4	cat	$\{\langle n_5, S_0\rangle, \langle n_{14}, S_5\rangle\}$

ALGORITHM 3: UpdateBitmap(Q, c)

```
1 if |Q| = 0 then
2     H ← empty map from a key ∈ Σ to a bitmap ∈ {0, 1}^(2τ+1);
3 for each key e ∈ H do
4     new ← H[e] ≪ 1;
5     if new ≠ 0^(2τ+1) then
6         H[e] ← new;
7     else
8         delete H[e] from H;
9 H[c] ← H[c] | 1;                    /* H[c] ← 1, if c ∉ H */;
```

ALGORITHM 4: BuildBitmap(Q', n')

```
1 B_n' ← H(n'.char);                  /* n'.char is n''s label */;
2 k ← |Q'| − n'.len;        /* n'.len is the length of n''s corresponding prefix */;
3 B_n' ← B_n' ≪ (τ − k);
4 return B_n';
```

5.1.2. An Example.

Example 5.3. Consider the trie shown in Figure 2(b) and a query cat. We show the active nodes and their edit vector states for each step of the query in Table II. In the first two steps, the only node is the root node (associated with the initial state). From step 2 to 3, we first update the global bitmaps and the map $\mathcal{H}$, where $\mathcal{H}(\mathtt{a}) = 001$, $\mathcal{H}(\mathtt{c}) = 010$, and any character otherwise is mapped to 000. For the child a of the root, it finds the bitmap via the $\mathcal{H}$ and looks up in the edit vector automaton (Figure 6). We obtain $f(S_0, 001) = S_1$ and mark $\langle n_1, S_1\rangle$ as an active node. Likewise, $\langle n_4, S_0\rangle$ becomes an active node. As $\mathcal{H}(\mathtt{m}) = 000$, the state of the node n_{11} is S_4. Since S_4's (-1)-diagonal value ($ed(\mathtt{ca}, \mathtt{m})$) exceeds threshold τ, FindActive is called to search for active nodes under n_{11}. Then we have n_{12}, whose state is S_5 and its 0-diagonal value ($ed(\mathtt{ca}, \mathtt{ma})$) is within the threshold. Therefore, we mark node $\langle n_{12}, S_5\rangle$ as an active node.

Following the previous maintaining process, in step 4, there are two active nodes n_5 and n_{14} maintained.

The previous example illustrates the basic BEVA algorithm, that is, without the optimizations to be introduced in Section 6, and hence is referred to as BEVA0. Note that it already compares favorably with the ICAN algorithm by maintaining only seven instead of 21 active nodes in total (see Example 2.3).

5.2. Fetching Results and Generating Output

In this subsection, we first discuss how to output qualified strings only, then their corresponding prefix edit distances, and finally the top-k results.

5.2.1. Fetch Qualified Strings Only. When only qualified strings are to be returned, in our BEVA algorithm, we simply return all the strings reachable from the current active nodes, which can be efficiently supported by keeping the starting and ending string IDs for each node (denoted as *n.start* and *n.end*, respectively) in the trie.

Compared with the standard implementation of output in existing methods, the major advantage in our method is that there is *no need to perform duplicate elimination*, as there is no ancestor-descendant relationships among our active nodes. Note that duplicate elimination of existing methods requires extra costs, since they need to perform ancestor-descendant checking on a trie (as active nodes may not always form direct parent-child relationships, such as `xy` and `xyab` in Figure 4, and this is generally true for *pivotal active nodes* [Li et al. 2011]).

We adopt a simple result fetching algorithm for Ji et al. [2009] that performs ancestor-descent checking by comparing string ID ranges stored in each trie node, which incurs a constant cost for each active node. This algorithm requires the input nodes to be sorted in preorder.

ALGORITHM 5: FetchResultICAN($\mathcal{A}$)

```
   Input: A maintained in the preorder
1  R ← ∅;                                   /* result set */;
2  lastID ← −1;                             /* Track last ID fetched */;
3  for each active node n in A do
4  |  if lastID < n_start then
5  |  |  R = R ∪ { i | i ∈ [n_start, n_end] };
6  |  |  lastID ← n_end;
7  return R;
```

Continue with Example 5.3: assume we need to return results after step 4; that is, the current query is cat. If only qualified strings need to be returned, for BEVA, we just simply return all the strings that can be reached by active nodes n_5 and n_{14}, which are strings with ids 2, 3, 4, and 6 respectively; for ICAN and ICPAN, we use Algorithm 5. Among all the current active nodes, the ones that are used to report results are also n_5 and n_{14}, but with the extra cost of traversing through all the six current active nodes compared with BEVA.

5.2.2. Fetch Qualified Strings with Prefix Edit Distance. Next we consider returning the prefix edit distance of each qualified string. This is useful to applications that employ a general ranking function, which entails scoring every qualified string. As BEVA's active nodes are all boundary active nodes, we need to perform further state transitions until the prefix edit distance of the current node is no larger than the minimum value of the current edit vector.

The pseudocode is shown in Algorithm 6. In Line 2, $S[\tau + 1 + k]$ is the edit distance of current node n with respect to Q and we use it to update the prefix edit distance of the current node. By Theorem 2.2, it can be shown that when the current prefix edit distance value is no larger than $\min(S)$, which denotes the minimum value of the cells in S, all the strings under n will have the same prefix edit distance. Hence, the algorithm only performs additional state transitions recursively (Line 6) if this condition is not satisfied. The algorithm employs the function report, which outputs the final query results. Its input parameters are a range of string IDs and a prefix edit distance value; these can also be deemed as a compressed representation of the output. For convenience, we assume report will ignore invalid input values (e.g., when the range's start value is larger than its end value).

ALGORITHM 6: FetchResultEdBEVA($\langle n, S\rangle$, ped)

```
Input: ⟨n, S⟩ is the trie node and its state. ped is the prefix edit distance of n.
1  k ← |Q| − n.len;
2  ped′ ← min(S[τ + 1 + k], ped);
3  if ped′ > min(S) then
4  |   report([n.start, n.firstchild.start − 1], ped′);
5  |   for each child n′ of n do
6  |   |   FetchResultEdBEVA(⟨n′, f(S, B_n′)⟩, ped′);
7  else
8  |   report([n.start, n.end], ped′);
```

Since there is no known algorithm for ICAN or ICPAN to output all qualified strings and their prefix edit distances, we devise Algorithm 7 as an efficient algorithm to implement these functionalities. The algorithm requires that the active nodes in $\mathcal{A}$ are maintained in preorder, and hence the start IDs of the nodes are in increasing order. The algorithm works in a line-sweeping manner by maintaining the following invariants: (i) all the qualified strings whose ID is smaller than *lastID* have been output; (ii) all the active nodes whose string ID range intersects with *lastID* have been maintained in the stack, where the top of the stack contains the lowest such active node in the trie; (iii) *lastID* equals the start ID of the top node in the stack (except in the initialization stage). A subtlety is that in Line 10, we take the minimum of active node n's edit distance and that of its parent-to-be node in the stack. This is because n's edit distance may be larger.

ALGORITHM 7: FetchResultEdICAN($\mathcal{A}$)

```
Input: Active nodes in A are in preorder.
1  Append a sentinel active node whose n.start is ∞ to A;
2  for each active node ⟨n, ed⟩ in A do
3  |   while S.empty() = false and S.top().end < n.start do
4  |   |   ⟨n′, ed′⟩ ← S.pop();
5  |   |   report(⟨[max(n′.start, lastID), n′.end], ed′⟩);
6  |   |   lastID ← n′.end + 1;
7  |   if S.empty() = false and lastID < n.start then
8  |   |   report(⟨[lastID, n.start − 1], S.top().ed⟩);
9  |   lastID ← n.start;                          /* update lastID */;
10 |   S.push(⟨n, min(ed, S.top().ed)⟩);          /* take the min */;
   |   /* if S is empty, S.top().ed returns τ + 1                    */
11 return R;
```

Continue with Example 5.3, assuming the current query is cat, and both qualified strings and their prefix edit distances are to be returned. For BEVA, we call Algorithm 6 using each current active node (n_5 and n_{14}) as input. For n_5, as the minimum edit value in S_0 is 0, which is smaller than n_5's current edit distance value 1 ($S_0[3]$), we recursively compute the next states for n_5's two child nodes n_6 and n_9, which are $\langle n_6, S_4\rangle$, $\langle n_9, S_0\rangle$, and use them as input respectively to call Algorithm 6. For n_{14}, since $min(S_5) = S_5[2] = 1$, we do not need to go down further and directly report all strings under n_{14} as a result with a prefix edit distance of 1. Table III shows the nodes that are processed by Algorithm 6 and results reported using them.

For ICAN, we call Algorithm 7 using the current active node set as input. First, n_5, n_6, and n_8 are added to the stack one by one, and before n_8 is added, the range

Table III. BEVA0 FetchResultsEd Example

Nodes Processing	Results Reported
n_5	∅
n_6	([2, 3], 1)
n_9	([4, 4], 0)
n_{14}	([6, 6], 1)

Table IV. ICAN FetchResultsEd Example

Nodes Processing	Results Reported
before add n_8	([2, 2], 1)
pop n_8	([3, 3], 1)
pop n_6	∅
pop n_{10}	([4, 4], 0)
pop n_9	∅
pop n_5	∅
pop n_{14}	([6, 6], 1)

$[lastID, n_8.start - 1]$ (i.e., [2, 2]) is reported with a prefix edit distance of 1. When processing n_9, all nodes in the stack that are no higher than n_9 (in terms of trie level), that is, n_8 and n_6, are popped out with corresponding string ID intervals reported ([3, 3], 1). Similarly, when processing n_{14}, n_{10}, n_9, and n_5 are popped, and ([4, 4], 0) reported. Finally, n_{14} is popped with ([6, 6], 1) reported. In summary, all current active nodes are processed in order to correctly report all result strings and their prefix edit distance. Table IV shows all the result intervals reported with the corresponding processing actions on related active nodes.

It can be seen clearly from Tables III and IV that although the prefix edit distance is required, BEVA accesses fewer nodes than ICAN for reporting all the results.

5.2.3. Fetch Top-K Results. Finally, we consider top-k retrieval. For the special case when the scoring function is monotonic in both the edit distances and static scores of the strings, it is easy to support pruning and early stopping in the previous output routines that output edit distance (i.e., Algorithms 6 and 7). For example, given the inputs of the report function, we can compute the maximum score among all the strings whose IDs fall within the input interval, provided that we can efficiently obtain the maximum static score of these strings. These strings can be discarded entirely if the maximum score is no larger than the score of the current kth result (we call it the early-stop condition).

For our BEVA algorithm, since the input intervals of the report function are nonoverlapping and each of them always corresponds to a trie node, we can precompute and materialize for each trie node the top-k list of string IDs with the highest static scores. Whenever an interval of strings associated with a trie node is input to the report function, we traverse the top-k list of the node to retrieve strings with the highest scores and use them to update the top-k result list until the early stop condition is met.

This method does not apply to ICAN or ICPAN, for example, when some but not all of the child nodes of an active node are not active nodes, which violates the nonoverlapping trie node interval property and results in the failure of the one-to-one corresponding relationship between input intervals of the report function and the trie nodes. Therefore, we either need an additional range maximum index [Hsu and Ottaviano 2013] (capable of returning the strings with maximum static scores given an input ID range) or additional duplicate removal of strings in the top-k result list in order to utilize the precomputed maximum static scores for each trie node. To be more specific about the

second point, assume that we store the top-k list of string IDs with the highest static scores for each trie node, similar to our BEVA method. During the fetching process presented in Algorithm 7, whenever a node n' is popped out from the stack and part of its interval is passed to the report function, we traverse the static top-k list of n' to update the top-k result list until the early-stop condition is met. Note that the updating process may add into the result list true-positive entries but with false scores (strings whose IDs are not in the current input interval). This case occurs when there are ancestor-descendant relationship among active nodes and the current processing node n' is an ancestor node. Due to the carefully designed node processing order in Algorithm 7, it can be proved easily that strings in this case must have been encountered when the descendant nodes of n' are processed, and the descendant nodes are processed before n'. Therefore, it can be resolved by checking whether the current entry has been encountered and processed previously before adding it into the result list. Note that this duplicate checking is nontrivial when the result size is large.

Using the same Example 5.3, we assume only the top-k result strings with their final scores are to be reported. For BEVA, at each time an interval of results is to be reported (shown in the second column in Table III), we traverse the top-k list of the corresponding nodes (shown in the first column in Table III), aggregate final scores for each traversed string, and use them to update the top-k result list until the early-stop condition is met. For ICAN, each time an active node is popped out from the stack (shown in the first column in Table IV), the top-k list of the node is traversed, and the final scores of traversed strings are aggregated and added to the top-k result list until the early stop condition is met. However, during the traversal process, strings that have been processed and scores aggregated before should be skipped. For example, string cate with an ID of 4 will occur in the static top-k list of n_{10}, n_9, and n_5, and all of them are in the current active node set. Whenever one of them is popped out from the stack, string cate will be traversed, but only the first traversal (at n_{10}) is needed since the final scores computed at later traversal (such as n_5) may be incorrect due to the wrong prefix edit distance value.

6. FURTHER OPTIMIZATIONS

In this section, we introduce two optimizations that can further speed up our proposed algorithm.

6.1. Depth-First Computation

There are several cases when the autocompletion system receives multiple query characters at a time. This could happen when the user types in the query very fast, when the query is pasted into the search box, or when the user hits several backspaces.

The standard way to deal with this in existing algorithms is to process these query characters one by one. This essentially follows the breadth-first search (BFS), which calculates $\mathcal{A}_{Q[1..i]}$ before calculating $\mathcal{A}_{Q[1..i+1]}$. This may cause substantial overhead as nodes that are ultimately disqualified at a later query length may be added and maintained in the intermediate steps.

We optimize for this scenario with a depth-first search (DFS) strategy, which calculates $\mathcal{A}_{Q[1..i+k]}$ directly from $\mathcal{A}_{Q[1..i]}$ without maintaining $\mathcal{A}_{Q[1..i+t]}, \forall t \in [1, k)$. We iterate over all the active nodes in $\mathcal{A}_Q$, and for each node, we visit its child nodes in a depth-first fashion (i.e., extract and shift the B into the correct bitmap and call the DFS on the child node recursively if its edit vector is not the terminal vector). To support this procedure, we modify the input parameters of Algorithm 2 to $(x, i, \langle n, S\rangle)$ and replace Line 10 with FindActive$(x, i + 1, \langle n, S\rangle)$, where i indicates the ith character of x.

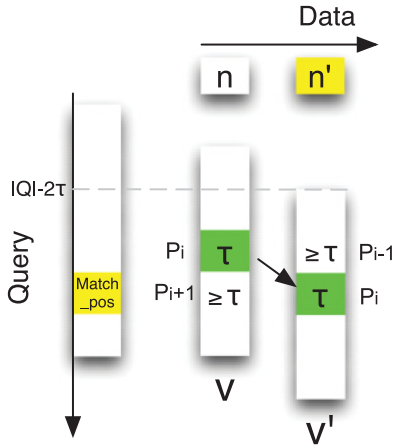


Fig. 7. The ND state transition.

Note that if only one character is appended to the current query, the DFS strategy is the same as the original one. Therefore, we can safely use the DFS strategy under all the cases.

6.2. Optimizations for Nearly Disqualified States

Consider the edit vector automaton in Figure 6. We observe that some states contain only self-loop and the transition to the terminal state (e.g., states S_5, S_6, and S_7). We call such states *nearly disqualified states*, or *ND states*.

Definition 6.1 (*Nearly Disqualified (ND) states*). A state is called an ND state if all the values in its vector are no smaller than τ, and the *degree* of the state is the number of τ values it contains. An ND state with degree d is denoted as ND_d. For example, S_5 in Figure 6 is an ND_1 state.

Given a query Q, an active node n whose state is ND_d, and its edit vector v, without loss of generality, we assume the ith τ value ($i \in [1, d]$) in v is at position p_i ($p_i \in [1, 2\tau + 1]$) and consider computing the next edit vector v' for n's child nodes.

LEMMA 6.2. *Given an ND state ND_d with edit vector v, if $v[p_i] = \tau$, the only option for $v'[p_i]$ not to be # is that there exists a child node n' of n whose label matches $Q[|Q|-2\tau+p_i]$.*

PROOF. A general example can be seen from Figure 7. Because the values on the diagonals of the edit distance matrix will never decrease, and all values in v are no smaller than τ, we have $v[p_i] = \tau$, $v[p_i + 1] \geq \tau$, $v'[p_i - 1] \geq \tau$. Since $v'[p_i] = min\{v[p_i]+\delta(n', Q[|Q|-2\tau+p_i]), v[p_i+1]+1, v'[p_i-1]+1\}$, the only chance for $v'[p_i] = \tau$ is $\delta(n', Q[|Q| - 2\tau + p_i]) = 0$, which means the label of n' matches $Q[|Q| - 2\tau + p_i]$. □

Therefore, we name $Q[|Q| - 2\tau + p_i](i \in [1, d], p_i \in [1, 2\tau + 1])$ as ND_d's *survival characters* with respect to Q and have the following property.

PROPOSITION 6.3. *Let n be a node in an ND state ND_d. To prevent a child node n' of n from transiting to the terminal state, its label must match one of the survival characters of ND_d with respect to Q.*

PROOF. According to Lemma 6.2, if n' does not match any of the survival characters of ND_d, then all the values in v' will be #, rendering it a terminal state. □

Therefore, the optimization for nodes in the ND_d states is *not* to access every child node of n, but just check for all the survival characters if there exists a child node with a label matching it when d is smaller than the number of child nodes of n. We achieve this by modifying Line 2 of Algorithm 2: we first perform a comparison between d and n's number of child nodes if n is in an ND_d state, and then iterate through either the survival characters or n's child nodes accordingly.

If the trie implementation supports random access to its child nodes, then this optimization uses an $O(\tau)$ time child test rather than the $O(\Sigma)$ time iteration through child nodes to process each node in the ND states.

This optimization is quite useful as we found that a large percentage (measured about 90%) of nodes during the query processing belongs to one of the ND states.

Example 6.4. Consider when a query is changed from `ca` to `cat` and the active state S_5 with its extent $\{n_{12}\}$ in Table II. Since S_5 is ND_1, its only survival character for the query is `t`. So we can directly access the `t` child node of n_{12} using this optimization.

7. UNIVERSAL EDIT VECTOR AUTOMATA

The EVA described in Section 4 (we call it original EVA in this section) has two limitations. First, its size is exponential in the edit distance threshold τ, as τ dictates the edit vector length, possible bitmap number, and maximum value in the edit vector. For example, when $\tau = 5$, the total number of transitions is around 35 million. Therefore, a large τ will eventually prevent the building and the use of the original EVA. Second, although it can be built without the knowledge of the query or the alphabet, its precomputation depends on the value of τ. A naive workaround is to build multiple original EVA s, each for a different τ value between 1 and some prespecified $\tau_{\max}$ value.

In this section, we present the universal partitioned edit vector automaton, which does not have the previous two limitations. We first present the partitioned EVA, or PEVA, which works for specific τ (Section 7.1), and then extend it to the universal PEVA, which can be constructed without knowing the τ value, hence being truly *universal* (Section 7.2).

7.1. Partitioned EVA (PEVA)

To support a large τ, our initial idea is to partition the *initial edit vector* into several edit vectors with shorter length (the length of EVA that we can afford) and use them as the initial state to compute several partitioned EVAs. Then each state in the original EVA can be represented by several states each from one of the partitioned EVAs. The detailed algorithms are presented next.

7.1.1. PEVA Construction. Given the maximum edit vector length l of the EVA that we can afford and a threshold τ, we partition the *initial edit vector* of length $2\tau + 1$ to r shorter edit vectors with length l, where $r = \lceil (2\tau + 1)/l \rceil$. The last partition is padded with # when $2\tau + 1$ is not divisible by l. For example, assume $\tau = 5, l = 4$; the initial edit vector $[5, 4, 3, 2, 1, 0, 1, 2, 3, 4, 5]^T$ is partitioned into three initial edit vectors $[5, 4, 3, 2]^T$, $[1, 0, 1, 2]^T$, and $[3, 4, 5, \#]^T$.

Using the r partitioned edit vectors ($I_i, 1 \le i \le r$) as the *initial edit vector*, respectively, r automata (denoted as $\mathcal{PA}_i^\tau, 1 \le i \le r$) are precomputed using the transition function $f_p(v_j, B, c_u, c_l)$ similar to $f(v_j, B)$ described in Section 4.2 but with two extra inputs, c_u and c_l. Consider the vectors v_j and v_{j+1} in Figure 8, assuming they are edit vectors in the partitioned EVAs. In order to correctly compute v_{j+1} from v_j, not only is bitmap B needed, but also the values of the cell above v_{j+1} (c_u, the green cell) and the cell below v_j (c_l, the red cell) are required. The reason is that v_j and v_{j+1} are only part of the edit vectors in original EVA, so cells c_u and c_l may contain values no larger than τ, which affects the computation of the first and last values of vector v_{j+1}. By starting

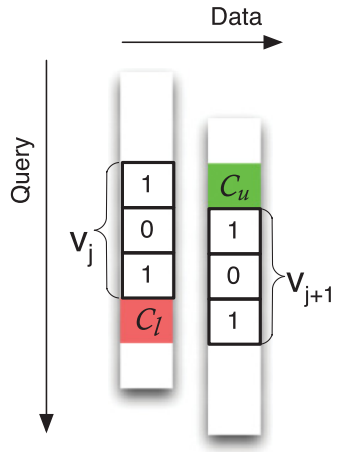


Fig. 8. Partitioned edit vector transition.

from the initial state (with the initial edit vector $I_i, 1 \leq i \leq r$) and continuing to feed combinations of all possible bitmaps of length l and all possible values for c_u and c_l to newly created states, $\mathcal{PA}_i^\tau$ $(1 \leq i \leq r)$ can be successfully precomputed. A state S_j in the original EVA is now represented as r small states $S_j[i]$ $(1 \leq i \leq r)$, where $S_j[i]$ is a state in $\mathcal{PA}_i^\tau$. We call the EVA computed in this way the partitioned edit vector automata (PEVA) with respect to τ.

Regarding to the complexity of the construction of each partitioned EVA $\mathcal{PA}_i^\tau$, the possible number of bitmaps is 2^l. The possible number of values for c_u is three for each v_j, that is, $v_j[1]-1$, $v_j[1]$, and $v_j[1]+1$. We observe for the latter two cases that $v_{j+1}[1]$ is independent of c_u, since $v_{j+1}[1] = \min(c_u+1, v_j[1]+\neg B[1], v_j[2]+1) = \min(v_j[1]+\neg B[1], v_j[2]+1)$. Therefore, we only need to keep two values for c_u: $v_j[1]-1$ and $v_j[1]$, indicating whether it affects $v_{j+1}[1]$ or not. Similarly, the values of c_l that need consideration are $v_j[l]-1$ and $v_j[l]$. Although this simple idea does not change the asymptotic construction complexity, it reduces the total transition numbers by 5/9. The number of states of each $\mathcal{PA}_i^\tau$ can be shown to be bounded from upper by $(\tau+2)\cdot 3^{l-1}$.[5] Therefore, the time complexity of constructing each $\mathcal{PA}_i^\tau$ is $O(\tau \cdot 6^l)$ and the complexity for constructing the $\mathcal{PA}^\tau$ is $O(\lceil \frac{2\tau+1}{l} \rceil \cdot \tau \cdot 6^l)$ in total.

7.1.2. Implementing the Maintenance Step Using PEVA. In the maintenance step of Algorithm 2, to compute the state S' associated with node n' from (n, S) and bitmap $B_{n'}$ (Line 5), r number of next state lookup operations from the r number of PEVAs are performed instead of only one using original EVA. That is, given the r state $S[i], (1 \leq i \leq r)$ associated with the node n and the bitmap $B_{n'}$, we first partition the bitmap into r partitions $(B_{n'}[i], (1 \leq i \leq r))$ with length l (the last partition is padded with 0 when $2\tau+1$ is not divisible by l). Then each of the r next state $S'[i]$ is obtained from $\mathcal{PA}_i^\tau$ by the following: $S'[i] = f_p(S[i], B_{n'}[i], S'[i-1][l], S[i+1][1])$, where $S[i][j]$ represents the j^{th} cell value of the edit vector associated with state $S[i]$. The r initial states associated with the root node are the initial states of the r PEVAs.

A nice property of using partitioned EVA is that whenever a partitioned edit vector $S[i]$ reaches the terminal edit vector, we no longer need to compute the transition of

[5]To show this, one needs to refer to the proof of Lemma 7.8 (page 26) where it shows that there are at most 3^{l-1} number of *difference vectors* of length $l-1$. Each difference vector determines at most $\tau+2$ edit vectors in PEVA as the cell values of edit vectors in $\mathcal{PA}_i^\tau$ are bounded by 0 and τ.

Table V. Comparison of PEVA and UPEVA

Length	Initial Edit Vector	# of States		
		PEVA		UPEVA
5	[2, 1, 0, 1, 2]	$\tau = 4$	199	81
		$\tau = 5$	280	
		$\tau = 6$	361	

this vector in the subsequent maintaining process due to the nondecreasing nature of diagonal values stated in Theorem 2.2.

7.2. Universal PEVA (UPEVA)

The PEVA method reduces the edit vector length but still needs to build different automata to accommodate all possible different thresholds; these automata are different because the maximum cell values in the edit vectors are different.

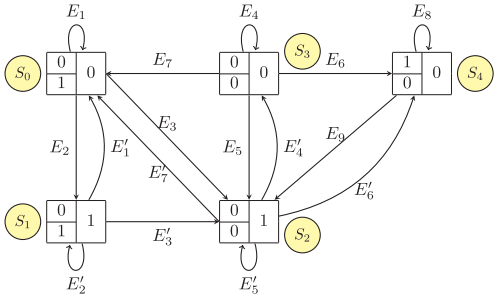
Our idea to build a single automaton for all possible threshold values is based on the observation that edit vectors can be grouped into equivalence classes. For example, although $[5, 6, 7]^T$ and $[6, 7, 8]^T$ are two different edit vectors, their transitions with respect to any given bitmap are identical in terms of the relative changes to the values in the edit vectors. Therefore, we can put them into in an equivalence class represented by vector $[0, 1, 2]^T$ by subtracting 5 and 6 from the two vectors, respectively. Since the vector length is l and any two adjacent cells in an edit vector differ by at most 1, we are able to reduce the maximum cell value in any edit vectors to a bounded value. This leads to our universal partitioned EVA method, that is, to precompute an automaton that has a controllable size and supports arbitrary thresholds—a truly universal automaton for edit distance computation.

In the following, we first introduce how to construct such an automaton for a given length and initial edit vector and then discuss how to obtain the initial edit vectors.

7.2.1. UPEVA Construction. Given a length l and an initial edit vector (v_0) of length l, an automaton is computed by starting from v_0 and using the transition function $f_u(v_j, B, c_u, c_l)$, which involves two steps. The first step is to transit v_j to a new vector v using the same transition function $f_p(v_j, B, c_u, c_l)$ described in Section 7.1.1. The second step is to check, after the new vector v is generated, whether all the values in v are no less than 1. If so, we reduce all the values in v by 1 to be a new vector v' and record $(v', 1)$ as the next state in the automaton. Otherwise, we record $(v, 0)$ as the next state. Hence, the output of the transition function $f_u(v_j, B, c_u, c_l)$ is a vector and flag value pair (v, a), where $a \in \{0, 1\}$ is the flag value indicating whether the reduction happens: 1 means reduction occurred and 0 means it did not.[6] For example, a vector $[1, 2, 3]^T$ will be reduced as $[0, 1, 2]^T$, and we record $([0, 1, 2]^T, 1)$ as the new state.

Using this idea, the size of the automaton computed is substantially reduced in terms of numbers of states and transitions compared with the PEVA constructed in Section 7.1.1, and they are applicable to any threshold τ. We call PEVAs computed using this idea universal partitioned edit vector automaton (UPEVA, and denote $\mathcal{U}^l$ as a UPEVA of length l). Table V gives the comparison of numbers of states in PEVA and UPEVA for a given vector length $l = 5$; the initial edit vector is $[2, 1, 0, 1, 2]^T$ and the threshold value τ varies from 4 to 6. We note that UPEVA is a special type of automaton where there is no start or accept states; however, we abuse the term here for ease of exposition. The edit vectors in $\mathcal{U}^l$ have the following property.

[6]Note in original EVA or PEVA, we use the concepts *edit vector* and automata *state* exchangeably since each state is associated with only an edit vector, but in this section, each state is associated with an edit vector and flag value pair.



(a) UPEVA $\mathcal{U}^2$

Edge	transition labels(* means 0 or -1)
E_1, E_1'	$[\langle 0,-1,*\rangle, \langle 1,-1,*\rangle, \langle 2,*,*\rangle, \langle 3,*,*\rangle]$
E_2, E_2'	$[\langle 0,0,0\rangle]$
E_3, E_3'	$[\langle 0,0,-1\rangle, \langle 1,0,*\rangle]$
E_4, E_4'	$[\langle 0,-1,-1\rangle, \langle 1,-1,*\rangle, \langle 2,*,-1\rangle, \langle 3,*,*\rangle]$
E_5, E_5'	$[\langle 0,0,0\rangle]$
E_6, E_6'	$[\langle 0,0,-1\rangle, \langle 1,0,*\rangle]$
E_7, E_7'	$[\langle 0,-1,0\rangle, \langle 2,*,0\rangle]$
E_8	$[\langle 0,*,-1\rangle, \langle 1,*,*\rangle, \langle 2,*,-1\rangle, \langle 3,*,*\rangle]$
E_9	$[\langle 0,*,0\rangle, \langle 2,*,0\rangle]$

(b) Edge labels

Fig. 9. Universal partitioned EVA with length 2.

PROPOSITION 7.1. *For every edit vector v in $\mathcal{U}^l$, there must exist at least one 0 in v, and 0 is its smallest value. The maximum possible value in v is $l-1$. There is no "#" in v.*

PROPOSITION 7.2. *Every edit vector v in $\mathcal{U}^l$ represents an equivalence class of edit vectors v' from PEVA, where each v' is an edit vector that can be transferred to v by subtracting* $\min_{i\in[1,l]}(v'[i])$ *from all the values in v'.*

Figure 9 shows the UPEVA for $l=2$ and initial edit vector $v_0=[0,1]^T$. We represent each state using its state number S_j in a yellow shaded circle placed nearby, and its associated edit vector and flag value are organized in a table. For each transition from S_j to S_{j+1}, we represent the B, $c_u - v_j[1]$ and $c_l - v_j[2]$ values as integers on the edges, where v_j is the edit vector associated with S_j. For example, the transition from S_0 to S_2 is via an edge labeled with $\langle 1,\ 0,\ -1\rangle$, which represents $B=001, c_u - v_0[1]=0$ and $c_l - v_0[2]=(-1)$, respectively.

It is worth noting that if two states are associated with the same edit vector but different flag values (called *paired states*), for example, S_0 and S_1 in Figure 9, their outgoing transitions are always the same. Therefore, we optimize our implementation by computing and storing the outgoing transitions only once for those paired states. Note that it is not necessarily that all states have paired states; for example, state S_4 in Figure 9 does not have a paired state as none of the three $\mathcal{U}^2$ edit vectors $([0,1]^T, [0,0]^T, [1,0]^T)$ can reach the state $([1,0]^T, 1)$.

Obtaining initial edit vectors. For a given threshold τ, we need to first partition the original initial edit vector of length $2\tau+1$ into r number of PEVA initial edit vectors with length l, where $r=\lceil\frac{2\tau+1}{l}\rceil$. Then, for each of the r PEVA initial edit vectors $(I_i, 1\le i\le r)$, we check if the minimum value in I_i $(\min(I_i))$ is no less than 1. If so, we reduce all values in I_i by $\min(I_i)$ to be new vector I_i', record $\min(I_i)$ as a *delta value*, and use I_i' as the initial edit vector to construct UPEVA. If not, we use I_i as the initial edit vector and record 0 as the delta value. For the last partition when $2\tau+1$ is not divisible by l, we replace the padded values in PEVA's initial edit vectors by actual edit distance values, that is, increase by 1 from the value in the previous cell. Using the same example as in Section 7.1.1, assuming $\tau=5, l=4$, the last partition $[3,4,5,\#]^T$ in PEVA becomes $[3,4,5,6]^T$ in UPEVA and is then reduced to $[0,1,2,3]^T$ with a delta value 3. The first two partitions become initial edit vectors $[3,2,1,0]^T$ and $[1,0,1,2]^T$ with delta value 2 and 0, respectively.

7.2.2. Precomputing the UPEVA. Similar to precomputing EVA introduced in Section 4.2.3, we precompute UPEVA of length l $(\mathcal{U}^l)$ by starting with the initial state and continuing to feed all possible bitmap values to generate new states. However, UPEVA is different from the original EVA in construction in two aspects:

—There is more than one initial edit vector in $\mathcal{U}^l$.
—There is no terminal edit vector in $\mathcal{U}^l$.

We show that we can use any edit vector in $\mathcal{U}^l$ as the initial edit vector to generate $\mathcal{U}^l$, and the termination condition is that there is no more new edit vector yet to be discovered.

First, we introduce a few definitions and lemmas to show that all the edit vectors in $\mathcal{U}^l$ form a *connected* graph.

Definition 7.3 (*Subsequent Edit Vector*). Given two edit vectors v_1 and v_2, we call v_2 a subsequent edit vector of v_1 if there exists exactly one i, s.t. $v_2[i] = v_1[i] + 1$, and $\forall j \neq i$, $v_2[j] = v_1[j]$.

For example, assume $v_1 = [1, 0, 1, 2]^T$; then edit vector $[1, 1, 1, 2]^T$ is a subsequent edit vector of v_1, but $[1, 1, 2, 2]^T$ is not.

LEMMA 7.4. *Given two edit vectors v_1 and v_2, if v_2 is a subsequent edit vector of v_1, there exists a bitmap B such that $v_2 = f(v_1, B)$.*

PROOF. Let i be the only position where v_1 and v_2 differs. We construct B as the bit vector that only the ith bit is 0 and all the other bits are 1. Let $v = f(v_1, B)$, where f is defined in Equation (1). We shall show that $v_2 = v$,

Consider $v_2[j]$ where $j \neq i$. By definition, $v_2[j] = v_1[j]$. According to Equation (1) and the fact that adjacent cells in the edit matrix differ by at most 1, we have

$$\begin{aligned} v[j] &= \min(v_1[j] + \neg B[j], v_1[j+1] + 1, v[j-1] + 1) \\ &= \min(v_1[j], v_1[j+1] + 1, v[j-1] + 1) \\ &= v_1[j] && \because v[j-1] \geq v_1[j] - 1 \text{ and } v_1[j+1] \geq v_1[j] - 1. \end{aligned}$$

Now consider $v_2[i]$. By definition, $v_2[i] = v_1[i]+1$. According to Equation (1), we have

$$\begin{aligned} v[i] &= \min(v_1[i] + \neg B[i], v_1[i+1] + 1, v[i-1] + 1,) \\ &= \min(v_1[i] + 1, v_1[i+1] + 1, v[i-1] + 1) \\ &= \min(v_1[i], v_1[i-1], v_1[i+1]) + 1 && \because v[i-1] = v_2[i-1] = v_1[i-1] \text{ proved earlier.} \end{aligned}$$

There are two possible cases:

Case I: $v_1[i] \leq \min(v_1[i-1], v_1[i+1])$, then $v_2[i] = v_1[i] + 1$.
Case II: $v_1[i] > \min(v_1[i-1], v_1[i+1])$. This case is impossible, as this means $v_2[i] - v_2[i-1] = (v_1[i] + 1) - v_1[i-1] \geq 2$, and contradicts the fact that adjacent cell values in an edit vector must not differ by more than 1.

The lemma then follows. □

Definition 7.5 (*Reachable Edit Vectors*). Given two edit vectors v_1 and v_2, if $\forall i$, $v_1[i] \leq v_2[i]$, we call v_2 a reachable edit vector of v_1.

LEMMA 7.6. *Given two edit vectors v_1 and v_2, if v_2 is a reachable edit vector of v_1, there exists a finite number of bitmap sequences $B_1, B_2, \ldots, B_k$, so that v_1 can be transformed into v_2 in k transitions, where $k = \sum_i (v_2[i] - v_1[i])$.*

PROOF. We construct a sequence of $k+1$ edit vectors $u_0, u_1, u_2, \ldots, u_{k-1}, u_k$ such that

—$u_0 = v_1$ and $u_k = v_2$, and
—u_{i+1} is the edit vector of u_i constructed in the following way: let $p = \min(\arg\min_j u_i[j])$, then $u_{i+1}[p] = u_i[p] + 1$ and $u_{i+1}[j] = u_i[j]$, $\forall j \neq p$.

It is easy to check that the previous sequence exists with $k = \sum_i (v_2[i] - v_1[i])$, and also u_{i+1} is a subsequent edit vector of u_i $(0 \leq i \leq k-1)$. Based on Lemma 7.4, we can

conclude that there exists a sequence of k bitmaps that transform u_0 to u_k. Hence, the lemma is proved. □

LEMMA 7.7. *For any two UPEVA edit vectors v_1 and v_2, there exists a finite number of transitions that transform v_1 to v_2.*

PROOF. Since both v_1 and v_2 are UPEVA edit vectors, they are edit vectors too. If v_2 is a reachable edit vector of v_1, then the lemma follows from Lemma 7.6. Otherwise, consider v_2' such that $v_2'[i] = v_2[i] + \delta$, where $\delta = \max_i(v_1[i] - v_2[i])$. Obviously, v_2' is a reachable edit vector of v_1, and hence there exists a sequence of transformation, $\mathcal{B}$, from v_1 to v_2'. v_2' and v_2 are in the same equivalence class with respect to a UPEVA. Therefore, in a UPEVA, $\mathcal{B}$ will transform v_1 to v_2. □

Lemma 7.7 essentially reveals that the UPEVA forms a *connected* graph, as there is a path of transitions between any two states. Therefore, we can precompute a single UPEVA by starting from *any* UPEVA edit vector and feeding it with all possible bitmaps. We refer to the resulting UPEVA $\mathcal{U}^l$. It also implies that all edit vectors satisfying both Proposition 7.1 and Lemma 4.5 will be included in the $\mathcal{U}^l$.

The number of nodes is the total number of edit vectors of length l that satisfy both Proposition 7.1 and Lemma 4.5, which we denote as $|\mathcal{U}^l|$. We can also bound the number of states of $\mathcal{U}^l$ by the following lemma.

LEMMA 7.8. *The number of states in $\mathcal{U}^l$ is at most $2 \cdot 3^{l-1}$.*

PROOF. Let $v = [a_1, a_2, \ldots, a_l]^T$ be a UPEVA edit vector in $\mathcal{U}^l$. Define its corresponding length $l-1$ *difference vector* $D(v)$ as $[a_2 - a_1, a_3 - a_2, \ldots, a_l - a_{l-1}]^T$.

We claim that there is a one-to-one mapping between the set of length l UPEVA edit vectors and the set of length $l-1$ difference vectors. It is obvious that a v uniquely determines $D(v)$. Given a $D(v) = [b_1, b_2, \ldots, b_{l-1}]^T$, it can be generated by the set of edit vectors in the form of $\mathcal{S} = [\lambda, \lambda + b_1, \lambda + b_1 + b_2, \ldots, \lambda + \sum_{i=1}^{l-1} b_i]^T$, where λ is taken from an appropriate subset of integers. Since the minimum value of a UPEVA edit vector v must be 0 according to Proposition 7.1, there is only one such $v \in \mathcal{S}$ that can produce $D(v)$; therefore, $D(v)$ uniquely determines v.

According to Lemma 4.5, all the entries in the difference vector $D(v)$ have three possible values, -1, 0, or 1, and hence there are at most 3^{l-1} difference vectors, and this is also the upper bound for the number of UPEVA edit vectors. Since each UPEVA edit vector generates at most two states in $\mathcal{U}^l$, the Lemma follows. □

Based on this analysis, the total number of $\mathcal{U}^l$ states is at most $2 \cdot 3^{l-1}$, and all possible transitions for each state is 2^{l+2}, and we can conclude that the precomputation cost of $\mathcal{U}^l$ is $O(6^l)$.

Discussions. Note that in Lemma 7.6, we do not assert that there are cases where v_1 is actually transformed into v_2; in other words, it is possible that the sequence of bitmaps required by the proof actually contains conflicts. An example of conflicts is that if $B_i = 111$, then it is impossible for B_{i+1} to be 011. This means the UPEVA (also original EVA) generated by the precomputation algorithm may not be the minimum. We leave the problems of precomputing the minimum UPEVA as future research.

7.2.3. Implementing the Maintenance Step Using UPEVA. The maintenance process using UPEVA is similar to using PEVAs presented in Section 7.1.2 with only two differences: (1) maintaining and accumulating delta values for each state and (2) initializing initial vectors.

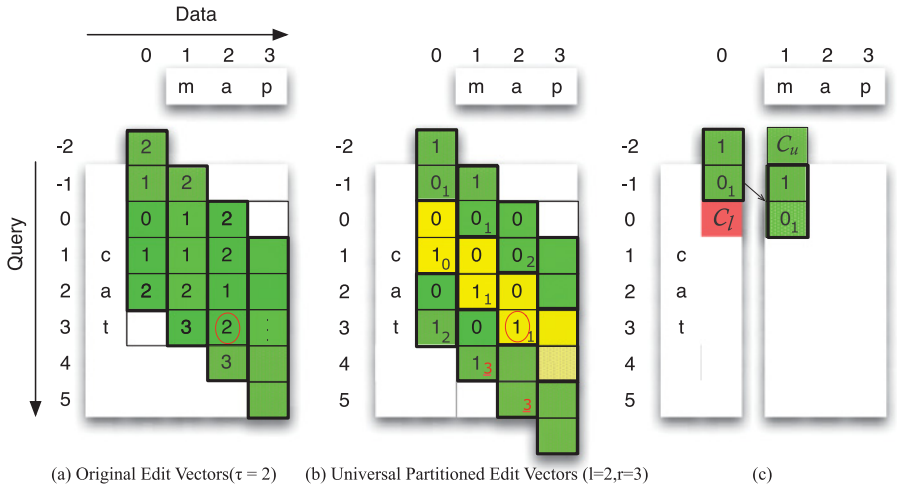


Fig. 10. Original edit vector to universal partitioned edit vector ($\tau = 2, l_{max} = 2$).

In the initialization step, we compute the r partitioned initial edit vectors and their delta values using the method presented in Section 7.2.1 and associate them with the root node.

In the maintenance step, given the r states $S[i], (1 \leq i \leq r)$ and their delta values $\Delta[i]$ associated with node n, for the child node n' of n, we get its state $S'[i]$ together with a flag value a' using the transition function $(S'[i], a') = f_u(S[i], B_{n'}[i], S'[i-1][l], S[i+1][1])$. The computation of the bitmap $B_{n'}$ is the same as PEVA. Then we accumulate the delta value by recording $\Delta'[i] = \Delta[i] + a'$ for $S'[i]$.

The main reason for accumulating a delta value is to maintain the real edit distance values for each edit vector. Hence, after each transition, we can decide if all edit distance values in a state $S[i]$ are larger than τ by checking its corresponding delta value. If its delta value is larger than τ, we no longer need to compute the transition of this state in the subsequent maintaining process.

7.2.4. An Example. Figure 10 shows an example of edit vector transition using UPEVA compared with the original EVA. Assume the query string is cat, the data string is map, $\tau = 2$, and $l = 2$. As can be seen from the figure, each original edit vector of length 5 is represented by three partitioned edit vectors of length 2. Accumulative delta values are shown at the bottom right corner of each partitioned vector. Padded and out-of-matrix boundary cells are shown in the dotted background.

In the initialization step, we compute the initial UPEVA states from the initial edit vector $[2, 1, 0, 1, 2]^T$ by partitioning it to $r = \lceil \frac{2\tau+1}{l} \rceil = 3$ parts, $[2, 1]^T$, $[0, 1]^T$, $[2]^T$, and we extend the last partition to length l by padding a value 3, so it becomes $[2, 3]^T$. Then we reduce each vector by its minimum cell value and record the minimum cell value as its corresponding delta value, so they become $([1, 0]^T, 1)$, $([0, 1]^T, 0)$, $([0, 1]^T, 2)$. Finally, we find the state number of these reduced edit vectors from the UPEVA $\mathcal{U}^2$ shown in Figure 9(a) and associate the state numbers and corresponding delta values to the empty data string (i.e., the root node if the data string is indexed in a trie). Therefore, the initial state is $\{(S_4, 1), (S_0, 0), (S_0, 2)\}$; to differentiate this state with the r states inside it, we call this state the *super state*, denote it as SS, and call the r states inside it the *single state*, denoted as $SS[i], 1 \leq i \leq r$. The delta values are denoted as $\Delta[i], 1 \leq i \leq r$.

In the maintaining step, when the query length is smaller than τ, we stay at the initial state and do nothing. When the query length is increased to $\tau + 1$, that is, letter t comes, we (1) read the next data string letter m, compute the bitmap of m against the query ∅∅cat as $B = 00000$, and pad and partition it into r parts {00, 00, 00} so that each partition corresponds to one of the r single states; and (2) transit from the initial state SS to the next state SS' by computing the next state $SS'[i]$ for each single state $SS[i]$ and accumulating the delta value $\Delta[i]$.

For the transition of $SS[1] = S_4$, the corresponding bitmap partition is $B[1] = 00$; c_u is an out-of-boundary cell as shown in Figure 10(c), so we set it as $\tau + 1 = 3$. The real edit distance value of cell $M[-2, 0]$ (i.e., $v_j[1]$) in Figure 10(b) is $S_4[1] + \Delta[1] = 1 + 1 = 2$. Therefore, $c_u = M[-2, 0] + 1$; as we mentioned in Section 7.1.1, when c_u is $v_j[1]$ or $v_j[1] + 1$, the transition output is the same, so here we set the transition input value $c_u - M[-2, 0]$ as 0 rather than 1. Similarly, $c_l = SS[2][1] + \Delta[2] = S_0[1] + \Delta[2] = 0$. The real edit distance value of cell $M[-1, 0] = SS[1][2] + \Delta[1] = 0 + 1 = 1$, so the transition input value $c_l - M[-1, 0] = -1$. Therefore, following the transition edge $\langle 0, 0, -1\rangle$ (E_8) from state S_4 in $\mathcal{U}^2$, we get the next state $SS'[1] = S_4$, and $\Delta'[1] = \Delta[1] + 0 = 1$ since the output flag value is 0. Using the same way to compute the next state of $SS[2]$ and $SS[3]$, we finally get the next super state $SS' = \{(S_4, 1), (S_1, 1), (S_1, 3)\}$.

The last delta value $\Delta'[3] = 3 > \tau$; we say this single state is crashed and we no longer need to compute the transition of this state in the subsequent transitions. The edit distance value of the current query prefix cat and the current data prefix m resides in cell $M[3, 1]$, which is larger than τ as $\Delta'[3] > \tau$, and there exist no crashed single states in SS', so we need to keep reading data string letters and computing state transitions as the same maintaining process presented in Section 5. We finally stop at data prefix ma, and as its edit distance with respect to query cat in cell $M[3, 2]$ is $2 \leq \tau$, we record the data string map as a result string.

7.2.5. ND Optimization Using UPEVAs. To use the ND optimization technique, for each vector v in the UPEVA precomputed, we store the number of 0 values and their positions in v. During the transition step, we check if a node n is nearly disqualified by checking if $\min(\Delta[i]), (1 \leq i \leq r)$ associated with n equals τ; if so, all the 0 value positions stored for the states $S[i]$ whose $\Delta[i] = \tau$ are nearly dead positions. Survival characters can be found accordingly.

8. EXPERIMENTS

In this section, we report and analyze experimental results.

8.1. Experiments Setup

The following algorithms are compared in the experiment:

—**ICAN** and **ICPAN** are two trie-based algorithms for error-tolerant autocompletion [Li et al. 2011].
—**IncNGTrie** is a recent algorithm for error-tolerant autocompletion [Xiao et al. 2013], which trades space for query efficiency. Therefore, we cannot run it for large τ values or very large datasets.
—**BEVA** is our proposed method based on the Boundary maintenance strategy and edit vector automata with both the depth-first and nearly disqualified optimizations. The default automaton used in BEVA is EVA, presented in Section 4. We will state explicitly if PEVA or UPEVA is used.
—**UDLA** results from replacing the EVA used in our BEVA algorithm with the universal deterministic Levenshtein automata [Mihov and Schulz 2004]. Since the characteristic vectors used by UDLA have length $2\tau + 2$, we keep as active nodes those trie nodes such that (1) their states are not the terminal state, and (2) their corresponding

Table VI. Dataset Statistics

Dataset	Num of Strings	Avg String Len	$\|\Sigma\|$
MEDLINE	1,782,517	10	26
UMBC	3,072,292	8	26
USADDR	9,485,981	40	97

string is of length $|Q| - \tau - 1$. Besides these active nodes, we also keep the boundary trie nodes (i.e., the active nodes in BEVA) so that results can be returned directly as BEVA if result fetching is called without the need to return edit distance.

All the experiments were carried out on a server with a Quad-Core AMD Opteron 8378@2.4GHz Processor and 96GB RAM, running Ubuntu 12.04. We implemented all the algorithms in C++ and compiled with `gcc` 4.4.

We select three publicly available datasets:

—**MEDLINE** is a repository of about 4 million journal citations and abstracts of biomedical literature.[7]
—**UMBC** is a collection of English paragraphs with over 3 billion words processed from the Stanford WebBase project.[8]
—**USADDR** is about 10 million real US POI addresses extracted from the SimpleGeo CC0 collection.[9]

For MEDLINE and UMBC, we tokenize them into terms using white spaces and punctuation. For USADDR, we treat each address as a data string. Statistics about the preprocessed datasets are provided in Table VI.

For MEDLINE and USADDR, we follow Chaudhuri and Kaushik [2009] to randomly sample 1,000 strings as queries. For UMBC, we obtain the set of terms that appear in the AOL query log [Pass et al. 2006] and having a similar (but not identical) term in the UMBC collection. We then randomly sample 1,000 query terms from this set.

By default, all algorithms return qualified strings. We evaluate two other variations (all or top-k qualified strings with their prefix edit distances) in Sections 8.5 and 8.7.

We measure and (1) the **query response time**, which consists of the **searching time** and the **result fetching time**—the former accounts for the total time to maintain active nodes or states, and the latter for fetching query results; (2) the **active node size**, which is the number of active nodes for ICAN and IncNGTrie, the number of pivotal active nodes for ICPAN, or the total number of nodes in the extents of active states for BEVA and UDLA. All the measures are averaged over 1,000 queries.

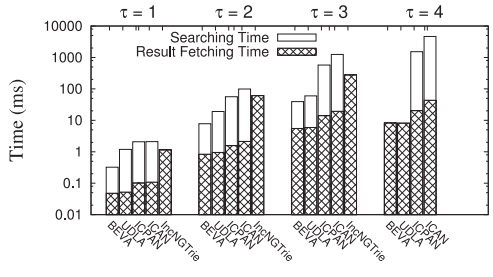
8.2. Varying Edit Distance Threshold

Figures 11(a) through 11(d) plot the query response times for all algorithms with edit distance threshold τ between 1 and 4, at fixed query lengths of 4 and 7.

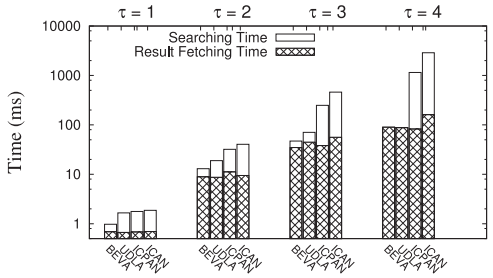
From the figures, we can observe the following:

—BEVA is fastest when the query length is small (when $|Q| = 4$, as in Figures 11(a)–11(b)). This is because result fetching time dominates, and BEVA has the best result fetching time. We analyze this in more detail in Section 8.5. In addition, BEVA's performance advantage is even more substantial when τ is large. For example, at query length 4 on USADDR, when $\tau = 1$, BEVA (0.97ms) is 1× faster than ICAN

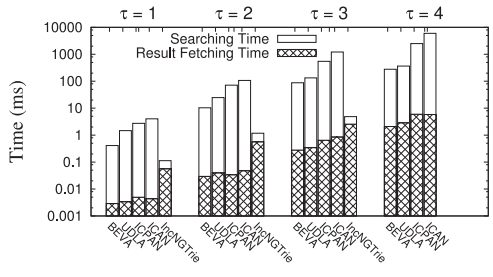
[7]http://mbr.nlm.nih.gov/Download/index.shtml.
[8]http://ebiquity.umbc.edu/resource/html/id/351.
[9]http://ia600809.us.archive.org/25/items/2011-08-SimpleGeo-CC0-Public-Spaces/.



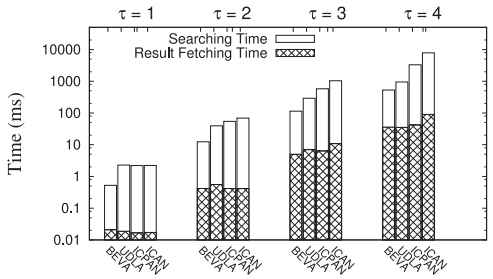
(a) MEDLINE, Query Response Time ($|Q| = 4$)



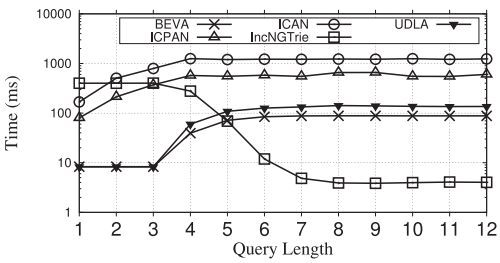
(b) USADDR, Query Response Time ($|Q| = 4$)



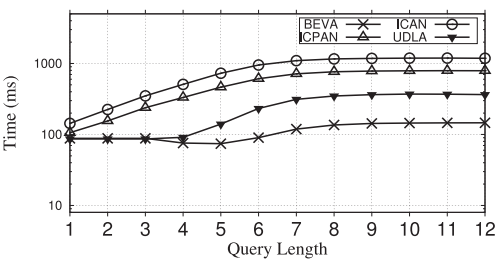
(c) MEDLINE, Query Response Time ($|Q| = 7$)



(d) USADDR, Query Response Time ($|Q| = 7$)



(e) MEDLINE, Query Response Time ($\tau = 3$)



(f) USADDR, Query Response Time ($\tau = 3$)

Fig. 11. Experimental results 1.

(1.86ms) and ICPAN (1.76ms), whereas when $\tau = 4$, BEVA (89.85ms) is 32 and 12× faster than ICAN (2883.75ms) and ICPAN (1147.79ms), respectively.

—When the query length is large ($|Q| = 7$ as in Figures 11(c)–11(d)), BEVA performs the second best; it is 6 and 10× faster than ICPAN and ICAN at $\tau = 1$, and 9 and 21× faster than ICPAN and ICAN, respectively, at $\tau = 4$ on MEDLINE.

—IncNGTrie outperforms all other algorithms by up to three orders of magnitude at the cost of using an index that is 15 to 19× larger than the trie index used by all other algorithms. Such a huge index size prevents IncNGTrie from working on large datasets like USADDR and large τ (e.g., 4).

—BEVA always has the smallest fetching time (shaded bar at the bottom) compared with ICAN, ICPAN, and IncNGTrie. The main reason is that our algorithm keeps a boundary active nodes set, and we do not need to perform duplicate elimination at the active node level required by ICAN and ICPAN. The result fetching time of IncNGTrie is the slowest in most of the settings as it requires deduplication at both active nodes and final results levels.

8.3. Varying Query Length

We plot the query response times for all algorithms with query length varying from 1 to 12 with a fixed $\tau = 3$. The results are shown in Figures 11(e)– 11(f). We can observe the following:

—The trends on the three datasets are similar. When query length is small ($\leq$3), BEVA and UDLA have a constant time and perform the best among all algorithms, as both BEVA and UDLA keep only the root node as the only active node.
—When the query length increases (from 4 to 8), the query response times of BEVA, UDLA, ICAN, and ICPAN increase, while IncNGTrie decreases. The main reason is that when the query length exceeds τ, the result size drops significantly, leading to a faster result fetching time. Result fetching requires much work for IncNGTrie, and hence the reduction in the amount of result fetching leads to a faster performance. This also makes IncNGTrie the most efficient when the query length exceeds 4, and BEVA is the second best (e.g., up to 31$\times$ faster than ICAN and 14$\times$ faster than ICPAN on MEDLINE). BEVA is slightly faster than UDLA (up to 3$\times$ on USADDR) since the ND states' optimization cannot be used in UDLA.
—When query is long enough (>8), all algorithms' query response times become stable. This is because query response time is dominated by searching time; the latter becomes stable as only few active nodes remain and need maintenance.

8.4. Searching Time and Active Node Size

We plot the search times for all algorithms in Figures 12(a) and 12(b), and active node sizes in Figures 12(c) and 12(d).

The searching time of BEVA is always less than those of ICPAN, ICAN, and UDLA. The advantage is more significant when the query length is small; for example, when $|Q| = 2$ on MEDLINE, the searching times are 0.001ms, 0.001ms, 195.372ms, and 483.872ms for BEVA, UDLA, ICPAN, and ICAN, respectively. This is because BEVA and UDLA do not perform any maintenance when the query is shorter than τ. BEVA and UDLA's time jumps at query length $\tau + 1$, due to the need to maintain active nodes. IncNGTrie outperforms BEVA when the query length is larger than τ. The reason is that IncNGTrie keeps significantly fewer active nodes than other algorithms during query length between 4 and 9.

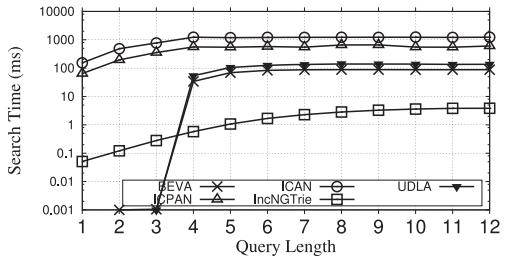
The behavior of the searching time is closely related to the number of active nodes for each query length, and the trend can be roughly observed by the cumulative active node sizes estimated from Figures 12(c) and 12(d). The main reasons BEVA has less searching time than ICPAN and ICAN are that BEVA keeps fewer active nodes and its maintenance cost per node is very small thanks to the use of EVA. The reason UDLA has a hump when the query length is around 8 is because its active node resides on the fifth level of the trie, which is very large.

8.5. Result Fetching Time

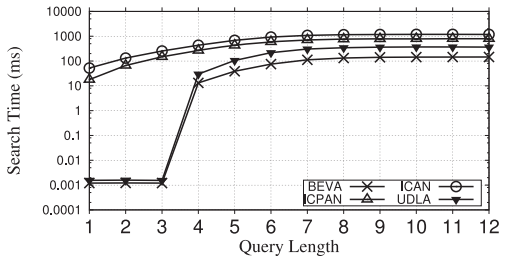
We evaluate the result fetching times and plot them in Figures 12(e) and 12(f). We distinguish algorithms with outputting edit distances by adding $_{\text{ED}}$ to the algorithm name.

As BEVA requires no extra process to remove ancestor-descendant active nodes, BEVA performs the best on all datasets and under most of settings.

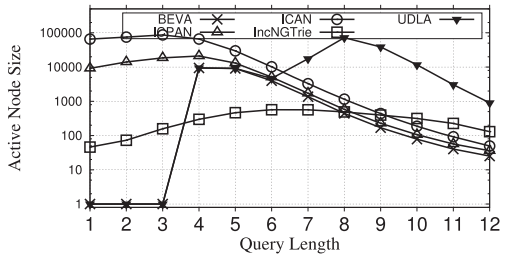
The advantage of BEVA's result fetching time is that it is around 3 to 4$\times$ faster than other algorithms if not returning the prefix edit distance. For example, at query length 6 on MEDLINE, the result fetching times are 0.85ms, 2.09ms, 2.51ms, and 10.22ms for BEVA, ICPAN, ICAN, and IncNGTrie, respectively. The result fetching time gap between BEVA and other algorithms tends to be smaller when the dataset is larger. The reason



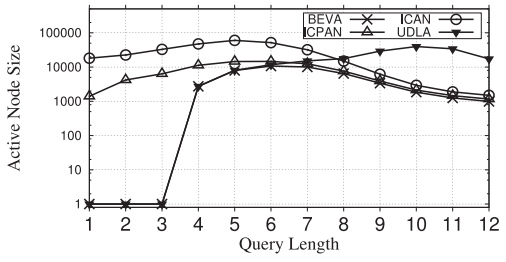
(a) MEDLINE, Searching Time ($\tau = 3$)



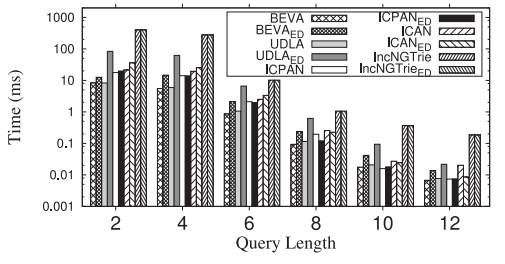
(b) USADDR, Searching Time ($\tau = 3$)



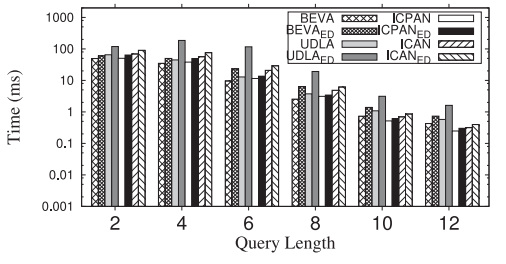
(c) MEDLINE, Active Node Size ($\tau = 3$)



(d) USADDR, Active Node Size ($\tau = 3$)



(e) MEDLINE, Result Fetching Time ($\tau = 3$)



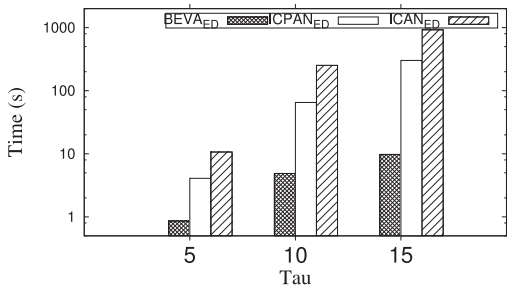
(f) USADDR, Result Fetching Time ($\tau = 3$)

Fig. 12. Experimental results 2.

is that the result fetching time depends on both active node sizes and number of results. The former does not tend to increase much with the dataset size, while the latter does. UDLA has a similar fetching time with BEVA since it uses exactly the same set of boundary nodes as BEVA to retrieve results.

We can also observe that additional time is needed when an algorithm (except IncNGTrie) needs to return prefix edit distances. This is because all algorithms need to perform additional computation (cf. Algorithms 6 and 7). For IncNGTrie, it uses a hashmap to remove duplicates, and during the process, the edit distance can be obtained free of additional cost. The main overhead for ICAN and ICPAN is the deduplication required at the active node level; even with an efficient algorithm (Algorithm 7) that avoids deduplication, it still introduces noticeable overhead due to iterating through a whole active node set. The index used by IncNGTrie contains much more redundancy, and its duplicate elimination has to be performed at both the active node and query results level. Therefore, it has the greatest fetching time among all algorithms.

UDLA reveals a significant increase in fetching time when the prefix edit distance is required since it has to recursively traverse down the trie from current active nodes until the trie node is crashed in order to correctly compute the prefix edit distance for result strings. Although BEVA also needs to perform this extra traversing of trie nodes,



(a) USADDR-1M, Query Response Time with ED ($|Q| = 30$)

Fig. 13. Large threshold experimental results.

thanks to the early stop condition used (in Line 2 of Algorithms 6), it outperforms other methods under most of the settings, especially when the query length is small. When the query length is large (e.g., >8), BEVA tends to be slower than ICPAN and ICAN since the traversing costs in BEVA outweigh active nodes' iterating cost in the other two methods due to the decrease of the active node set size. However, the result fetching time for all methods under such cases is quite small (<1*ms*), which makes the inefficiency insignificant.

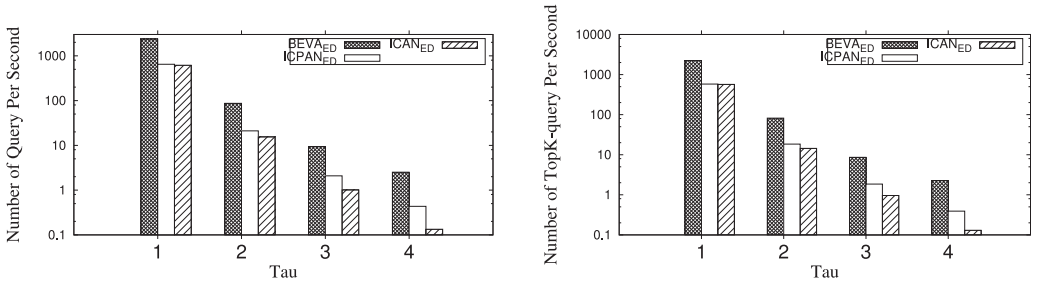
8.6. Large Threshold Values

We test the performance of BEVA, ICAN, and ICPAN for large threshold values where τ is set to 5, 10, and 15. We randomly select 1 million strings from the USADDRESS dataset as data strings with an average length of 41.9 and randomly choose 10 strings whose length is cut to 30 as query strings. We use the UPEVA presented in Section 7.2 for the BEVA algorithm, and l_{max} is set to 7. The total query time (including result fetching with edit distance) is shown in Figure 13. The following can be seen: (i) The query time for all algorithms increases along with τ value, but BEVA has the slightest slope. For example, along with the increase of τ from 5 to 15, the total query time of BEVA increases from 0.88s to 10.29s, while ICPAN is increased from 3.48s to 282.97s, and ICAN starts with 10.60s and goes up to 991.63s. (ii) BEVA outperforms the other two methods by one to two orders of magnitude in terms of the total query time, and the margin grows rapidly with the increase of τ.

8.7. Query Throughput

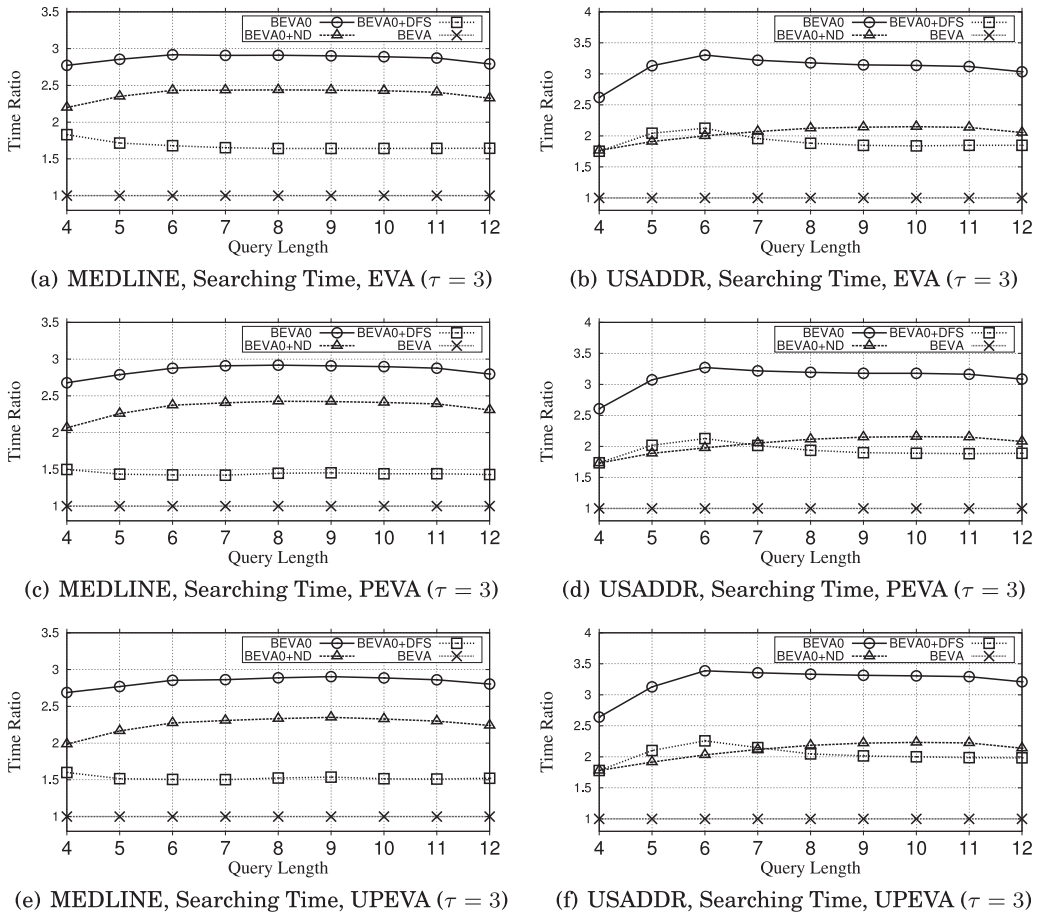
In this experiment, we test the query throughput of different algorithms as measured by query per second (QPS). We use the UMBC dataset and the queries carefully created from the AOL query log to simulate a real query workload; thresholds from 1 to 4 are used. We also evaluate two output variations: all or top-k qualified strings with their prefix edit distances or scores. We use a monotonic aggregate function using randomly assigned scores for strings and the edit distance. The results are shown in Figures 14(a) and 14(b).

We observe the following: (i) In both settings, BEVA consistently outperforms other algorithms, and the margin grows with the increase of τ. The improved QPS means a server equipped with BEVA can serve up to 17 and 5× more users/queries per physical machine than ICAN and ICPAN, respectively. (ii) The performances of all the algorithms do not differ much with different output options. Results with other k values are similar.



(a) Query Per Second UMBC (b) Query Per Second UMBC TOP-5

Fig. 14. Query throughput experimental results.



(a) MEDLINE, Searching Time, EVA ($\tau = 3$) (b) USADDR, Searching Time, EVA ($\tau = 3$)

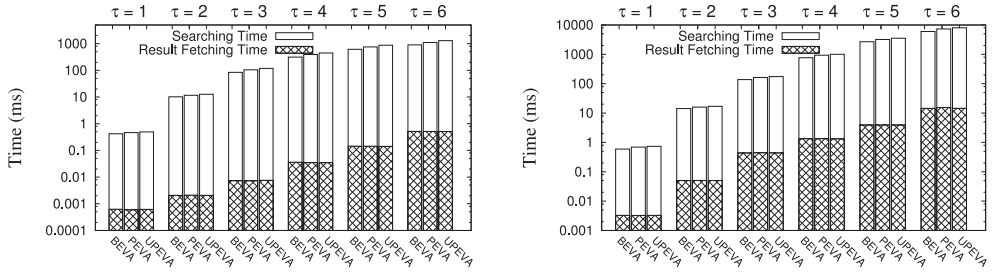
(c) MEDLINE, Searching Time, PEVA ($\tau = 3$) (d) USADDR, Searching Time, PEVA ($\tau = 3$)

(e) MEDLINE, Searching Time, UPEVA ($\tau = 3$) (f) USADDR, Searching Time, UPEVA ($\tau = 3$)

Fig. 15. Effects of optimizations.

8.8. Effects of Optimizations

We also perform experiments on the impact of our two optimizations (DFS and ND) and under all three types of automata settings (EVA, PEVA, and UPEVA). The results are shown in Figure 15. We call the basic BEVA algorithm without the two optimizations



(a) MEDLINE, Query Response Time ($|Q| = 12$) (b) USADDR, Query Response Time ($|Q| = 12$)

Fig. 16. Effects of varying automata.

BEVA0, and those with either optimization BEVA0+DFS and BEVA0+ND, respectively. For each algorithm, we plot the ratio of its searching time over that of BEVA. We only show the performance when $|Q| > \tau$, where the active state maintenance occurs.

First, both DFS and ND optimizations improve the search performance, although the effects of improvement depend on the dataset, and BEVA, with both optimizations, outperforms the other three by a large margin under all settings. In MEDLINE, the two optimization techniques improve the basic algorithm by around 2.9× in total (e.g., 243ms and 83ms for BEVA0 and BEVA, respectively, at query length 8 in Figure 15(a)). In USADDR, the total improvement is around 3.2 times (399ms for BEVA0 against 125ms for BEVA at query length 8 in Figure 15(b)).

Second, the improvement ratio of BEVA against BEVA0 on USADDR reaches the peak value of 3.3 at query length 6. The is due to the increasing number of active nodes from query length 1 to 6 as shown in Figure 12(d), which leads to the increasing improvement effects of BEVA0+DFS such that it reaches the peak ratio of 2.1 at $|Q| = 6$.

Finally, we observe that the automaton used has little impact on the algorithm performance. Detailed results on automata effects are discussed in Section 8.9.

8.9. Effects of Automata

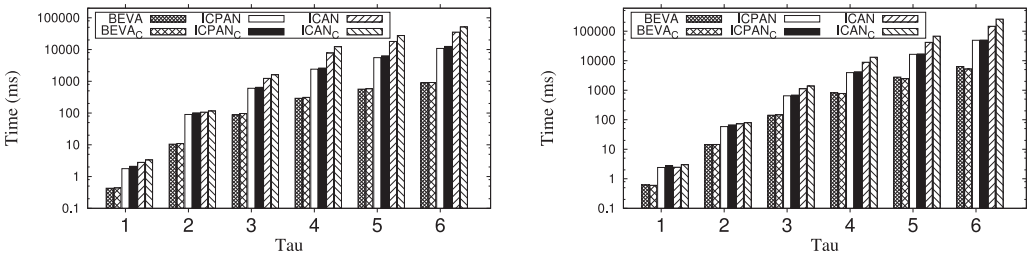
The experimental result regarding the effects of varying the type of automata (EVA, PEVA, and UPEVA) in the BEVA framework is presented in Figure 16. The test is performed by varying τ from 1 to 6. l_{max} in PEVA and UPEVA are both set to 7; therefore, there is no state partition when $\tau \in [1, 3]$.

We have the following observations: (i) It reveals a slight increase in the query time along EVA, PEVA, and UPEVA under the same parameter settings. The reason is that PEVA requires state partition when $l_{max} < 2\tau + 1$, and thus incurs a larger constant state transition cost than EVA, and UPEVA in turn consumes more time than PEVA due to an extra step of maintaining state delta values. (ii) Although the performance of EVA is better than that of PEVA and UPEVA, the difference ratio is no larger than 1.5 on both datasets. This is because the only difference among the three settings is the automata state transition cost, which is constant in all three settings. For example, when $|Q| = 12$ and $\tau = 6$, EVA is 1.22 and 1.43× faster than PEVA and UPEVA, respectively, on MEDLINE, and 1.21 and 1.33× faster than PEVA and UPEVA, respectively, on USADDR. Performance differences under other query lengths are similar.

We did not perform a comparison on even larger τ values, since the size of EVA increases exponentially along with the τ value. When $\tau = 7$, the estimated number of transitions of EVA using a lower bound given in Lemma 4.2 is around 1.2 billion, which is difficult to be precomputed.

Table VII. Index Statistics

Dataset	Num of Nodes, Original Trie	Num of Nodes, Compressed Trie
MEDLINE	4,933,395	1,962,946
UMBC	8,485,686	3,351,770
USADDR	302,284,735	14,132,375



(a) MEDLINE, Query Response Time ($|Q| = 12$) (b) USADDR, Query Response Time ($|Q| = 12$)

Fig. 17. Effects of compressed index.

8.10. Effects of Compressed Index

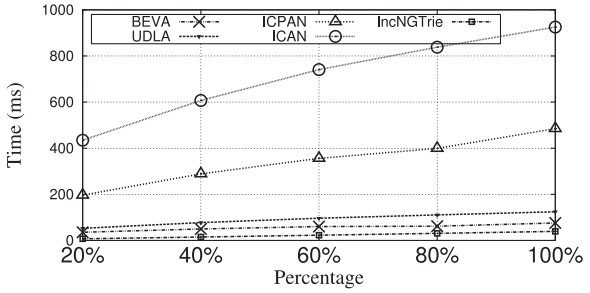
In this part, we evaluate the performance of BEVA, ICPAN, and ICAN on compressed tries as compressed tries are usually adopted to reduce the index space in practice. In this experiment, we use the widely employed Patricia trie as a compressed index structure. The main implementation differences with the original trie index are as follows:

—Each trie node has a key of char array type rather than a single char.
—Each active node is attached with (1) a pointer to the trie node as in an uncompressed index and (2) an extra integer that encodes the position of the current processing character in the char array of the pointed trie node.
—During the trie traversing process, each time we need to get the next data characters assuming the current processing node is n, instead of directly enumerating all the child nodes of n as in an uncompressed index, we need an extra checking process to see whether the stored position refers to the end of the char array of n. If not, we just increment the position by one and return the next character in the char array.

Table VII gives the number of trie nodes with respect to the original and compressed tries on the three datasets. It can be seen that we have saved more than half of the original space on MEDLINE and UMBC by adopting compressed tries, and saved 90% of the original space on USADDR.

The query performance result is shown in Figure 17. BEVA$_C$ refers to BEVA with a compressed index, similarly for ICPAN$_C$ and ICAN$_C$. Generally, there is a slight increase of query time for the three methods on the compressed index, due to the extra checking costs introduced at each node's key access. However, the time difference is not much, especially for BEVA and ICPAN. For example, the ratio of query time on the compressed index against time on the uncompressed index is no more than 1.1 on BEVA and 1.2 on ICPAN when τ varies from 1 to 6, while ICAN reveals a higher ratio of 1.5 than the other two methods since it involves a large amount of repeated trie node accesses, which leads to more extra checking costs.

Despite the adoption of a more complex index than the original trie, we notice a slight decreasing of query time for BEVA on the large dataset USADDR when the threshold value is large (e.g., $\tau > 4$). This is because BEVA traverses the index trie in DFS fashion as proposed in Section 6.1; when the compressed index is adopted, compressed



(a) Vary Dataset Size, MEDLINE ($\tau = 3$)

Fig. 18. Scalability test results.

Table VIII. Comparison of Automata Sizes

	# of States		# of Transitions		
τ	UDLA	EVA	UDLA	EVA	Reduced EVA
1	14	9	163	72	64
2	90	51	5,073	1,632	1,204
3	602	323	144,133	41,344	23,638
4	4,212	2,188	4,067,325	1,120,256	558,475

nodes help to reduce trie node enumeration during data string traversing and thus save memory access costs. However, the savings are not significant on small dataset or threshold values where the memory is already large enough regarding the algorithm requirements.

8.11. Other Experiments

We also evaluate the scalability with respect to the data size. We take the MEDLINE dataset and sample 20%, 40%, 60%, 80%, and 100% of the data. We run all the algorithms on these five sampled datasets and plot the query times in Figure 18(a). It can be seen that the query times of all algorithms grow approximately linearly with the size of the dataset, and BEVA and IncNGTrie have smaller growth rates than ICAN and ICPAN.

We show the sizes of our edit vector automata in Table VIII. The physical size is less than 5MB even when $\tau = 4$. The computational times for τ from 1 to 4 are 1, 13, 83, and 305ms.

9. RELATED WORK

Query autocompletion has become an important feature in most search systems. It is also helpful to construct complex queries [Nandi and Jagadish 2007a; Bast and Buchhold 2013] and perform complex tasks [Hawking and Griffiths 2013].

Traditional query autocompletion does not allow errors and performs only prefix matching. Bast and Weber [2006] proposed to use a succinct index built on an underlying document corpus to provide answers to word-level autocompletions and presented an efficient realization of the interactive search engine that integrates this feature [Bast et al. 2007]. Nandi and Jagadish [2007b] studied the problem of autocompletion at the level of a phrase containing multiple words. Li et al. [2009] proposed an approach to the query autocompletion on relational data. A user study comparing search interfaces with and without query autocompletions was conducted by White and Marchionini [2007].

Error-tolerant query autocompletion was first investigated in Ji et al. [2009] and Chaudhuri and Kaushik [2009], both methods based on the trie index and the framework of maintaining intermediate results for each query prefix to facilitate incremental search. Li et al. [2011] further improved the method in Ji et al. [2009] by reducing the size of intermediate results maintained while the user types in the query. Li et al. [2012] studied the problem of how to efficiently return the top-k relevant results of error-tolerant autocompletion and proposed top-k algorithms that support random access and sorted access on inverted lists. Xiao et al. [2013] proposed to index the deletion-marked variants of data strings to support very efficient error-tolerant autocompletion at the cost of a much-increased index size. Bast and Celikik [2013] extended Complete-Search [Bast and Weber 2006] by tolerating errors in the user input. Their approach does not follow the incremental fashion; instead, it proposes new indexes specializing in fuzzy prefix matching queries. As such, when the query length is short or the number of query results is large, this approach is less effective. A recent work [Zheng et al. 2014] presents a map application that supports error-tolerant type-ahead search over geo-textual data.

There have been many proposals on ranking the suggestions. Many factors that can be mined from query logs have been proposed to enable context-sensitive ranking [Bar-Yossef and Kraus 2011; Sengstock and Gertz 2011]. Returning personalized results based on a user's search history and location was also investigated [Shokouhi 2013]. However, these proposals do not tolerate errors in autocompletion.

If only edit distance is used for the ranking, the problem is similar to edit distance-based k-NN query, and several efficient algorithms based on trie [Deng et al. 2013] and approximate q-grams [Wang et al. 2013] have been proposed. A unified framework based on a hierarchical segment tree that supports both top-k and threshold-based string similarity search was proposed in Wang et al. [2015]. However, these methods require the query to be given a priori, and they cannot be easily adapted to our problem setting. Currently, most work assumes a monotonic ranking function that aggregates both the static scores of candidate strings and their edit distances to the query [Chaudhuri and Kaushik 2009; Li et al. 2011]. There are recent proposals considering more general and useful ranking functions, for example, proximity scores [Cetindil et al. 2014]. When the application domain includes spatial objects, ranking factors may include locations [Roy and Chakrabarti 2011; Zhong et al. 2012] and directions [Li et al. 2012b].

Apart from edit distance, other similarity criteria are also proposed for query autocompletion, for example, cosine similarity [Bar-Yossef and Kraus 2011], n-gram models [Nandi and Jagadish 2007b; Duan and Hsu 2011], and hidden Markov models with web-scale resources [Li et al. 2012a]. While these methods are all based on statistical language modeling and information retrieval techniques, another category of methods is based on extrinsic attributes such as popularity, review score [Chaudhuri and Kaushik 2009], and forecasted frequency [Shokouhi and Radinsky 2012].

Instead of query completions, another line of work aims at query recommendations, taking a full query and making arbitrary reformulations to assist users. The proposed solutions are mainly based on query clustering [Baeza-Yates et al. 2007; Sadikov et al. 2010], session analysis [He et al. 2009], search behavior models [Tyler and Teevan 2010], or probabilistic mechanisms [Bhatia et al. 2011].

If we perform error-tolerant matching for the entire string rather than one of its prefixes, this is the classic approximate string matching problem. Most existing methods can be classified into the following three categories. *Gram-based* methods, which are also the most common methods in the literature, converts the given similarity constraint to a gram overlap constraint and finds all possible candidates that share a certain number of grams with the query string. There are methods based on

fixed-length grams [Gravano et al. 2001; Sarawagi and Kirpal 2004; Chaudhuri et al. 2006; Qin et al. 2011, 2013] and variable-length grams [Li et al. 2007; Yang et al. 2008; Wang et al. 2013]. Due to the large number of false positives in potential candidates, various optimizations and filtering methods are proposed such as position filtering and length filtering [Gravano et al. 2001], prefix filtering [Chaudhuri et al. 2006; Bayardo et al. 2007], list skipping [Li et al. 2008], positional and suffix filtering [Xiao et al. 2008b], mismatch filtering [Xiao et al. 2008a], and alignment filtering [Deng et al. 2014]. However, as this line of work does not consume the query string character by character and it is hard to do incremental computation using inverted lists, they are not easy to be adapted to the approximate prefix matching problem. *Enumeration-based* methods [T. Bocek 2007; Arasu et al. 2006; Wang et al. 2009; Li et al. 2011, 2012; Zhang et al. 2013] generate all possible strings up to τ errors from the data strings and convert the approximate matching of the data string to the problem of exact matching of neighborhood strings. The obvious drawback of these algorithms is the exponentially increased index size along with the increase of similarity threshold, which renders them infeasible for a large threshold or dataset size in practice. The third class of work is *tree based* [Chaudhuri and Kaushik 2009; Feng et al. 2012; Zhang et al. 2010; Deng et al. 2013]. The prefix sharing intrinsic of a tree-based index saves a vast number of efforts on distance computation between the query string and data strings, and the traversal procedure is well fitted for incremental search. Therefore, this is one of the most widely adopted index structures for the approximate prefix matching problem [Chaudhuri and Kaushik 2009; Ji et al. 2009; Xiao et al. 2013]. When the error threshold grows relatively large, many of the aforementioned methods lose their filtering power rather rapidly. There are methods specially designed for this type of problem, including extended prefix filtering [Wang et al. 2012] and, most recently, local filtering [Yang et al. 2015].

We refer readers to the surveys of Navarro [2001a] and Boytsov [2011] for more extensive coverage on approximate string matching.

Edit Distance Computation. The classic and standard method to compute the edit distance between two strings d and Q (of length n and m, respectively) is the dynamic programming algorithm that fills in a matrix M of size $(n+1) \times (m+1)$ as mentioned in Section 2.2. It has been rediscovered many times in previous works (e.g., in Vintsyuk [1968], Needleman and Wunsch [1970], Sellers [1974], and Wagner and Fischer [1974]) from different areas back to the 1960s and 1970s. The time complexity is $O(n \cdot m)$ and the space complexity is $O(min(n, m))$. Based on the observation that the edit distance values on the upper-left to lower-right diagonals in the matrix are nondecreasing [Ukkonen 1985a], Ukkonen improved the standard $O(n \cdot m)$ method to $O(s \cdot min(n, m))$, where s is the actual edit distance of d and Q, by filling the matrix in diagonal-wise order in 1985. The space complexity is $O(s^2)$. Another line of improvement of the this standard method was proposed by Masek and Paterson [1980] in the early 1980s based on the Four-Russian techniques [Arlazarov et al. 1970]. The algorithm first precomputes solutions of all the possible subproblems of size $r \times r$ (r-blocks in the matrix) and stores them into a table, and then uses the table to achieve a constant lookup time for problems of size r in the original problem. If picking $r = \log_{3|\Sigma|} n$, the final time complexity is $O(nm/\log_{|\Sigma|})n)$, which improved the worst-case theoretical result (i.e., $O(nm)$). However, this algorithm is of only theoretical interest since it will not outperform the classical method for dataset size under 40GB as estimated by the same authors [Navarro 2001a]. In 1998, Myers [1999] proposed a bit-parallel algorithm for the computation of the dynamic programming matrix, which better used the bits of the computer word and improved the standard method to an algorithm with $O(\lceil m/w \rceil n)$ worst-case complexity.

In the following part, we focus on edit distance computations based on automata here, and refer readers to the survey of Navarro [2001a] for detailed information about other types of approaches.

The existing computation methods based on automata can be categorized into either NFA- or DFA-based methods. Navarro [2001b] builds an NFA of $(\tau + 1) \cdot (|Q| + 1)$ states for a query string Q with the edit distance threshold τ and then exploits bit-parallelism to simulate running the NFA. This idea has been pursued in many follow-up works, with some of the latest work [Hyyrö 2008] using an optimal number of bits and achieving further speedups. Early work constructs a distinct DFA for each distinct query string with a given threshold, and the DFA contains huge numbers of states. For example, Ukkonen [1985b] shows a bound of $O((|\Sigma| + |Q|) \cdot \min(3^{|Q|}, (2|\Sigma|)^{\tau} \cdot |Q|^{\tau+1}))$. Given its huge size, Navarro [1997] proposes to construct the DFA in a lazy fashion. One of the state-of-the-art DFA-based methods is the universal deterministic Levenshtein automata (abbreviated as UDLA) [Schulz and Mihov 2002; Mihov and Schulz 2004]. Its size does not depend on the string length or alphabet size, only on the edit distance threshold. Therefore, it is most similar to our EVA, but with several important differences:

—UDLA is not the most suitable for the error-tolerant prefix matching problem, as it is driven by input bitmaps (called *characteristic vectors*). Depending on whether additional characters will be appended to the query string or not, different bitmaps will be generated and may lead to different states. We replaced EVA with UDLA in our BEVA algorithm in our experiments, and the aforementioned limitations result in inferior performance.

—EVA is simpler and smaller than UDLA. While having almost the same runtime efficiency and usage, EVA is much smaller than UDLA, and Table VIII compares the number of states and transitions between UDLA and EVA when τ varies from 1 to 4. In addition, EVA is arguably easier to understand and implement than UDLA.

Finally, if comparing UDLA with our UPEVA, the obvious advantage of UPEVA is that it is independent of the similarity threshold, while UDLA requires building different automata for a different threshold.

10. CONCLUSIONS AND FUTURE WORK

Autocompletion has become a popular feature for search applications. By allowing a small amount of errors in the user input, the usability of the system can be significantly increased. This article addresses the query processing issue to efficiently support such error-tolerant autocompletions. We propose an efficient algorithm, BEVA, that maintains the minimum number of active nodes efficiently with the help of the novel edit vector automaton. Several optimizations were proposed on top of our methods. In addition, we devise the universal partitioned edit vector automaton that supports arbitrarily large thresholds. We demonstrated the superiority of our methods over existing solutions under a variety of settings.

There are several directions for future work.

First, in the current work, the EVA and UPEVA are precomputed by starting from the initial state and continue to feed **all** possible bitmaps of a given length to the newly created states. However, we observe that this naïve method, which was also employed by universal deterministic Levenshtein automata [Mitankin et al. 2011], creates *unnecessary* states and transitions, that is, the state or transition such that it can never be reached for two input strings.

Taking the transition $f(S2, B2)$ in Figure 6 as an example, this transition will never be reached by any pair of strings. The impossibility can be proved easily. As can be seen from Figure 6, the only transition (except self-loop) that can reach $S2$ is $f(S1, B3)$.

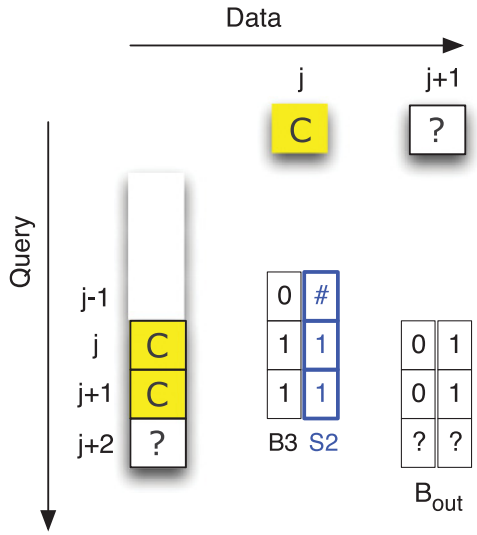


Fig. 19. Outgoing bitmap ($\tau = 1$).

Assume the transition between query Q and data string D reaches state $S2$ following $f(S1, B3)$ as shown in Figure 19. We know $B3 = 011$, which indicates the part of the query $Q[j-1, j+1]$ and the data character $D[j]$ that form the bitmap $B3$ have the following equality relationships: $Q[j] = Q[j+1] = D[j]$. Therefore, given any possible next data character $D[j+1]$, the bitmap that can be formed by $D[j+1]$ and $Q[j, j+2]$ must be in the form of "00?" or "11?". In other words, the valid outgoing bitmaps from state $S2$ contain only $B0 = 000$, $B1 = 001$, $B6 = 110$, and $B7 = 111$, thus, $f(S2, B2)$ is an invalid outgoing transition from $S2$.

Therefore, it is interesting to investigate how to precompute the automaton that ensures only necessary states and transitions are generated, hence generating the *minimum* edit vector automaton that satisfies both soundness and minimality. This can be achieved by only feeding *valid* bitmaps to new states by checking them against a history of previous bitmaps leading to the state. We have achieved some preliminary results by applying this idea to generate an EVA with reduced size. The number of transitions of the reduced EVAs are 64, 1,204, 23,638, and 558,475 for τ from 1 to 4, respectively, which compares favorably with the current EVA as shown in Table VIII.

Second, it is desirable to support more and flexible error-tolerant autocompletion. While most current works are based on edit distance and its variants, other similarity or distance functions are needed to model errors other than typographical errors. Furthermore, additional research is needed to support more flexible ranking of candidates, for example, in a context-sensitive manner [Brill and Moore 2000].

ACKNOWLEDGMENTS

We thank the anonymous reviewers who have provided valuable comments to help us improve the quality of the article.

REFERENCES

Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. 2006. Efficient exact set-similarity joins. In *VLDB*.

V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradžev. 1970. On economical construction of the transitive closure of a directed graph. *Soviet Math.—Doklady* 11, 5 (1970), 1209–1210.

Ricardo A. Baeza-Yates, Carlos A. Hurtado, and Marcelo Mendoza. 2007. Improving search engines by query clustering. *JASIST* 58, 12 (2007), 1793–1804.

Ziv Bar-Yossef and Naama Kraus. 2011. Context-sensitive query auto-completion. In *WWW*. 107–116.

Hannah Bast and Björn Buchhold. 2013. An index for efficient semantic full-text search. In *CIKM*. 369–378.

Hannah Bast and Marjan Celikik. 2013. Efficient fuzzy search in large text collections. *ACM Trans. Inf. Syst.* 31, 2 (2013), 10.

Holger Bast, Debapriyo Majumdar, and Ingmar Weber. 2007. Efficient interactive query expansion with complete search. In *CIKM*. 857–860.

Holger Bast and Ingmar Weber. 2006. Type less, find more: Fast autocompletion search with a succinct index. In *SIGIR*.

Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. 2007. Scaling up all pairs similarity search. In *WWW*.

Sumit Bhatia, Debapriyo Majumdar, and Prasenjit Mitra. 2011. Query suggestions in the absence of query logs. In *SIGIR*. ACM, 795–804.

Leonid Boytsov. 2011. Indexing methods for approximate dictionary searching: Comparative analysis. *ACM J. Exper. Algorithmics* 16, 1 (2011), 1.1–1.91.

Eric Brill and Robert C. Moore. 2000. An improved error model for noisy channel spelling correction. In *ACL*.

Andrei Z. Broder, Peter Ciccolo, Evgeniy Gabrilovich, Vanja Josifovski, Donald Metzler, Lance Riedel, and Jeffrey Yuan. 2009. Online expansion of rare queries for sponsored search. In *WWW*.

Inci Cetindil, Jamshid Esmaelnezhad, Taewoo Kim, and Chen Li. 2014. Efficient instant-fuzzy search with proximity ranking. In *ICDE*.

Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. 2006. A primitive operator for similarity joins in data cleaning. In *ICDE*.

Surajit Chaudhuri and Raghav Kaushik. 2009. Extending autocompletion to tolerate errors. In *SIGMOD*.

Silviu Cucerzan and Eric Brill. 2004. Spelling correction as an iterative process that exploits the collective knowledge of web users. In *EMNLP*. 293–300.

Dong Deng, Guoliang Li, and Jianhua Feng. 2014. A pivotal prefix based filtering algorithm for string similarity search. In *SIGMOD*. 673–684.

Dong Deng, Guoliang Li, Jianhua Feng, and Wen-Syan Li. 2013. Top-K string similarity search with edit-distance constraints. In *ICDE*.

Huizhong Duan and Bo-June (Paul) Hsu. 2011. Online spelling correction for query completion. In *WWW*. 117–126.

Jianhua Feng, Jiannan Wang, and Guoliang Li. 2012. Trie-join: A trie-based method for efficient string similarity joins. *VLDB J.* 21, 4 (2012), 437–461.

Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. 2001. Approximate string joins in a database (almost) for free. In *VLDB*.

David Hawking and Kathy Griffiths. 2013. An enterprise search paradigm based on extended query auto-completion. Do we still need search and navigation?. In *ADCS*.

Qi He, Daxin Jiang, Zhen Liao, Steven C. H. Hoi, Kuiyu Chang, Ee-Peng Lim, and Hang Li. 2009. Web query recommendation via sequential query prediction. In *ICDE*. 1443–1454.

Bo-June (Paul) Hsu and Giuseppe Ottaviano. 2013. Space-efficient data structures for top-k completion. In *WWW*. 583–594.

Heikki Hyyrö. 2008. Improving the bit-parallel NFA of Baeza-Yates and Navarro for approximate string matching. *Inf. Process. Lett.* 108, 5 (2008), 313–319.

Shengyue Ji, Guoliang Li, Chen Li, and Jianhua Feng. 2009. Efficient interactive fuzzy keyword search. In *WWW*. 371–380.

Chen Li, Jiaheng Lu, and Yiming Lu. 2008. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*.

Chen Li, Bin Wang, and Xiaochun Yang. 2007. VGRAM: Improving performance of approximate queries on string collections using variable-length grams. In *VLDB*.

Guoliang Li, Dong Deng, Jiannan Wang, and Jianhua Feng. 2011. PASS-JOIN: A partition-based method for similarity joins. *PVLDB* 5, 3 (2011), 253–264.

Guoliang Li, Jianhua Feng, and Jing Xu. 2012b. DESKS: Direction-aware spatial keyword search. In *ICDE*. 474–485.

Guoliang Li, Shengyue Ji, Chen Li, and Jianhua Feng. 2009. Efficient type-ahead search on relational data: A TASTIER approach. In *SIGMOD*. 695–706.

Guoliang Li, Shengyue Ji, Chen Li, and Jianhua Feng. 2011. Efficient fuzzy full-text type-ahead search. *VLDB J.* 20, 4 (2011), 617–640.

Guoliang Li, Jiannan Wang, Chen Li, and Jianhua Feng. 2012. Supporting efficient top-k queries in type-ahead search. In *SIGIR*.

Yanen Li, Huizhong Duan, and ChengXiang Zhai. 2012a. CloudSpeller: Query spelling correction by using a unified hidden markov model with web-scale resources. In *WWW (Companion Volume)*. 561–562.

Yinan Li, Jignesh M. Patel, and Allison Terrell. 2012. WHAM: A high-throughput sequence alignment method. *ACM Trans. Database Syst.* 37, 4 (2012), 28.

William J. Masek and Mike Paterson. 1980. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.* 20, 1 (1980), 18–31.

Stoyan Mihov and Klaus U. Schulz. 2004. Fast approximate search in large dictionaries. *Comput. Linguistics* 30, 4 (2004), 451–477.

Petar Mitankin, Stoyan Mihov, and Klaus U. Schulz. 2011. Deciding word neighborhood with universal neighborhood automata. *Theor. Comput. Sci.* 412, 22 (2011), 2340–2355.

Gene Myers. 1999. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. ACM* 46, 3 (1999), 395–415.

Arnab Nandi and H. V. Jagadish. 2007a. Assisted querying using instant-response interfaces. In *SIGMOD*.

Arnab Nandi and H. V. Jagadish. 2007b. Effective phrase prediction. In *VLDB*. 219–230.

Gonzalo Navarro. 1997. A partial deterministic automaton for approximate string matching. In *WSP'*. 112–124.

Gonzalo Navarro. 2001a. A guided tour to approximate string matching. *ACM Comput. Surv.* 33, 1 (2001), 31–88.

Gonzalo Navarro. 2001b. NR-grep: A fast and flexible pattern-matching tool. *Softw. Pract. Exper.* 31, 13 (2001), 1265–1312.

Saul B. Needleman and Christian D. Wunsch. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.* 48, 3 (1970), 443–453.

Greg Pass, Abdur Chowdhury, and Cayley Torgeson. 2006. A picture of search. In *Infoscale*. 1.

Jianbin Qin, Wei Wang, Yifei Lu, Chuan Xiao, and Xuemin Lin. 2011. Efficient exact edit similarity query processing with the asymmetric signature scheme. In *SIGMOD*. 1033–1044.

Jianbin Qin, Wei Wang, Chuan Xiao, Yifei Lu, Xuemin Lin, and Haixun Wang. 2013. Asymmetric signature schemes for efficient exact edit similarity query processing. *ACM Trans. Database Syst.* 38, 3 (2013), 16.

Senjuti Basu Roy and Kaushik Chakrabarti. 2011. Location-aware type ahead search on spatial databases: Semantics and efficiency. In *SIGMOD*. 361–372.

Eldar Sadikov, Jayant Madhavan, Lu Wang, and Alon Y. Halevy. 2010. Clustering query refinements by user intent. In *WWW*. 841–850.

Sunita Sarawagi and Alok Kirpal. 2004. Efficient set joins on similarity predicates. In *SIGMOD*.

Klaus U. Schulz and Stoyan Mihov. 2002. Fast string correction with Levenshtein automata. *IJDAR* 5, 1 (2002), 67–85.

Peter H. Sellers. 1974. On the theory and computation of evolutionary distances. *SIAM J. Appl. Math.* 26, 4 (1974), 787–793.

Christian Sengstock and Michael Gertz. 2011. CONQUER: A system for efficient context-aware query suggestions. In *WWW*.

Milad Shokouhi. 2013. Learning to personalize query auto-completion. In *SIGIR*. 103–112.

Milad Shokouhi and Kira Radinsky. 2012. Time-sensitive query auto-completion. In *SIGIR*. 601–610.

B. Stiller, T. Bocek, and E. Hunt. 2007. *Fast Similarity Search in Large Dictionaries*. Technical Report ifi-2007.02. Department of Informatics, University of Zurich.

Sarah K. Tyler and Jaime Teevan. 2010. Large scale query log analysis of re-finding. In *WSDM*. 191–200.

Esko Ukkonen. 1985a. Algorithms for approximate string matching. *Inf. Control* 64, 1–3 (1985), 100–118.

Esko Ukkonen. 1985b. Finding approximate patterns in strings. *J. Algorithms* 6, 1 (1985), 132–137.

T. K. Vintsyuk. 1968. Speech discrimination by dynamic programming. *Cybernetics* 4, 1 (1968), 52–57. Russian Kibernetika 4, 1, (1968), 81–88.

Robert A. Wagner and Michael J. Fischer. 1974. The string-to-string correction problem. *J. ACM* 21, 1 (Jan. 1974), 168–173.

Jin Wang, Guoliang Li, Dong Deng, Yong Zhang, and Jianhua Feng. 2015. Two birds with one stone: An efficient hierarchical framework for top-k and threshold-based string similarity search. In *ICDE*. 519–530.

Jiannan Wang, Guoliang Li, and Jianhua Feng. 2012. Can we beat the prefix filtering? An adaptive framework for similarity join and search. In *SIGMOD*. ACM, 85–96.

Wei Wang, Jianbin Qin, Chuan Xiao, Xuemin Lin, and Heng Tao Shen. 2013. VChunkJoin: An efficient algorithm for edit similarity joins. *IEEE Trans. Knowl. Data Eng.* 25, 8 (2013), 1916–1929.

Wei Wang, Chuan Xiao, Xuemin Lin, and Chengqi Zhang. 2009. Efficient approximate entity extraction with edit constraints. In *SIMGOD*. 759–770.

Xiaoli Wang, Xiaofeng Ding, Anthony K. H. Tung, and Zhenjie Zhang. 2013. Efficient and effective KNN sequence search with approximate n-grams. *PVLDB* 7, 1 (2013), 1–12.

Ryen W. White and Gary Marchionini. 2007. Examining the effectiveness of real-time query expansion. *Inf. Process. Manage.* 43, 3 (2007), 685–704.

Chuan Xiao, Jianbin Qin, Wei Wang, Yoshiharu Ishikawa, Koji Tsuda, and Kunihiko Sadakane. 2013. Efficient error-tolerant query autocompletion. *PVLDB* 6, 6 (2013), 373–384.

Chuan Xiao, Wei Wang, and Xuemin Lin. 2008a. Ed-Join: An efficient algorithm for similarity joins with edit distance constraints. *PVLDB* 1, 1 (2008), 933–944.

Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey Xu Yu. 2008b. Efficient similarity joins for near duplicate detection. In *WWW*. 131–140.

Xiaochun Yang, Bin Wang, and Chen Li. 2008. Cost-based variable-length-gram selection for string collections to support approximate queries efficiently. In *SIGMOD*. ACM, 353–364.

Xiaochun Yang, Yaoshu Wang, Bin Wang, and Wei Wang. 2015. Local filtering: Improving the performance of approximate queries on string collections. In *SIGMOD*. 377–392.

Xiaoyang Zhang, Jianbin Qin, Wei Wang, Yifang Sun, and Jiaheng Lu. 2013. HmSearch: An efficient hamming distance query processing algorithm. In *SSDBM*. 19:1–19:12.

Zhenjie Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, and Divesh Srivastava. 2010. Bed-tree: An all-purpose index structure for string similarity search based on edit distance. In *SIGMOD*. ACM, 915–926.

Yuxin Zheng, Zhifeng Bao, Lidan Shou, and Anthony K. H. Tung. 2014. MESA: A map service to support fuzzy type-ahead search over geo-textual data. *PVLDB* 7, 13 (2014), 1545–1548.

Ruicheng Zhong, Ju Fan, Guoliang Li, Kian-Lee Tan, and Lizhu Zhou. 2012. Location-aware instant search. In *CIKM*. 385–394.

Received October 2014; revised August 2015; accepted October 2015