The VLDB Journal (2020) 29:919–943
https://doi.org/10.1007/s00778-019-00595-4

REGULAR PAPER

Efficient query autocompletion with edit distance-based error tolerance

Jianbin Qin[1] · Chuan Xiao[2,3] · Sheng Hu[3,4] · Jie Zhang[5] · Wei Wang[6] · Yoshiharu Ishikawa[3] · Koji Tsuda[7] · Kunihiko Sadakane[7]

Received: 6 November 2018 / Revised: 1 April 2019 / Accepted: 12 August 2019 / Published online: 14 December 2019

Abstract
Query autocompletion is an important feature saving users many keystrokes from typing the entire query. In this paper, we study the problem of query autocompletion that tolerates errors in users' input using edit distance constraints. Previous approaches index data strings in a trie, and continuously maintain all the prefixes of data strings whose edit distances from the query string are within the given threshold. The major inherent drawback of these approaches is that the number of such prefixes is huge for the first few characters of the query string and is exponential in the alphabet size. This results in slow query response even if the entire query approximately matches only few prefixes. We propose a novel neighborhood generation-based method to process error-tolerant query autocompletion. Our proposed method only maintains a small set of active nodes, thus saving both space and time to process the query. We also study efficient duplicate removal, a core problem in fetching query answers, and extend our method to support top-*k* queries. Optimization techniques are proposed to reduce the index size. The efficiency of our method is demonstrated through extensive experiments on real datasets.

Keywords Query autocompletion · Similarity Search · Database · Neighbourhood generation tree

1 Introduction

Autocompletion guides users to type the query correctly and efficiently. Due to the convenience it brings to the users and the server, it has been adopted in many applications. For example, search engines like Google dynamically suggest keywords, and can optionally show top-ranked search results while users are typing a query. Other applications include desktop searches, command shells, text editors, and mobile applications. In some applications, especially for mobile devices, typing accurately is a tedious task and the user's input tends to contain typographical errors. Consequently, it is necessary for query autocompletion to tolerate errors when the user types in a query. Among the various approaches to deal with typographical errors, edit distance is a good

✉ Jianbin Qin
jqin@sics.ac.cn

Chuan Xiao
chuanx@nagoya-u.jp

Sheng Hu
hu.sheng.4s@kyoto-u.ac.jp

Jie Zhang
jiezhang1984@xaut.edu.cn

Wei Wang
weiw@cse.unsw.edu.au

Yoshiharu Ishikawa
ishikawa@i.nagoya-u.ac.jp

Koji Tsuda
tsuda@k.u-tokyo.ac.jp

Kunihiko Sadakane
sada@mist.i.u-tokyo.ac.jp

[1] Shenzhen Institute of Computing Sciences, Shenzhen University, Shenzhen, China

[2] Osaka University, Osaka, Japan

[3] Nagoya University, Nagoya, Japan

[4] Kyoto University, Kyoto, Japan

[5] Xi'an University of Technology, Xi'an, China

[6] The University of New South Wales, Sydney, Australia

[7] The University of Tokyo, Tokyo, Japan

measure for text documents and, therefore, has been widely adopted and studied [17,23,37,43,77].

The state-of-the-art solutions to the query autocompletion with edit distance constraints adopt the following paradigm: indexing data strings in a trie, and traversing the trie *incrementally* to compute edit distance between the paths and the current query string as each character of the query comes. Only the paths that satisfy the edit distance constraints are kept, and the end of these paths are called *active nodes*. Its performance has been shown to be superior to alternative paradigms such as q-grams [17]. Nevertheless, the efficiency of this approach critically depends on the number of active nodes, which is typically very large in practice (in the order of 10^5 for the methods in [17,37]), and linear in the database size or exponential in the alphabet size in the worst case. Some techniques have been proposed to alleviate the problem, e.g., maintaining only a chosen subset of active nodes [23,43,77], or using buffered strategy and precomputation [17]); however, the query processing time is still long as there are still a huge number of active nodes to maintain when the first few characters of the query string are typed in. The situation will be even worse for the applications in which strings have large-sized alphabets, e.g., Unicode or CJK characters.

In this paper, we explore in the following direction: *Can we drastically improve the runtime performance by preprocessing the data and build a large but affordable-sized index?* We devise a novel solution by indexing the *deletion-marked variants* of the data strings in a trie, and keeping a small set of active nodes during query processing.

Rather than index the original data strings, we index their deletion-marked variants, which are generated by deleting at most τ (the edit distance threshold) characters from the strings. When the user inputs a query, its deletion-marked variants are also (implicitly) generated and searched in the trie. This process can be performed incrementally and efficiently by maintaining a small set of *active states* whose size is small—typically in the order of 10^3—and insensitive to the alphabet size. To intuitively understand why this can be achieved by the deletion-marked variants, consider this example: Let the query string be `ab`. All the strings such as `aba`, `abb`, ..., `abz` can be represented as a single variant `ab#` ("#" means the corresponding character is deleted) which has an edit distance of 1 to the query string. Based on this idea, we develop the IncNGTrie algorithm for error-tolerant query completion. To further reduce the number of active states (to the order of 10), we remove redundant active states and propose the IncNGTrie+ algorithm.

In addition, we propose efficient duplicate removal techniques when fetching query results—a problem existing in previous approaches yet not fully investigated. We also study the case of processing top-k queries, in which results are ranked by a function monotonic in both edit distance and static score. When not optimized, our index size is large due to the inclusion of deletion-marked variants. Hence we introduce effective techniques to reduce the size of the index by eliminating different kinds of redundancy in the index. Finally, the superiority of our solution is demonstrated through extensive experimental evaluation.

We also note that we only focus on the single keyword case in this paper, but our method can be extended to support the multiple keyword case by taking intersection after result fetching [43,44]. For example, an inverted index is built to associate each keyword with a list of document (or term) IDs to indicate the documents (or terms) that contain the keyword. For each input keyword in the query, we identify the indexed keywords that contain the input as a prefix, using our method. Then an intersection is invoked to find the documents (or terms) having all the keywords. We may also use the threshold algorithm (TA) [26] on top of our method to efficiently retrieve top-k results for the case of multiple keywords. Moreover, the results tolerated by the edit distance constraints can serve as a set of candidates fed to language model-based postprocessing methods [24,25,29] for better error correction.

Our contributions can be summarized as:

- We solve the error-tolerant query autocompletion problem with edit distance constraints by utilizing deletion-marked variants. We develop indexing, searching, and result-fetching techniques for query processing, as well as optimization techniques to reduce index size to an affordable level.
- We design two neighborhood generation-based algorithms IncNGTrie and IncNGTrie+ that integrate the developed techniques. Unlike previous approaches, they achieve very small and alphabet-insensitive active node sizes to speed up query processing.
- We conduct extensive experiments on several real datasets. The proposed method has been shown to significantly outperform previous approaches in terms of query response time.

Compared to the conference version of this work [70], we make the following substantial improvements:

- An improved searching algorithm (IncNGTrie+) is developed to reduce the number of active states and searching time.
- The original duplicate removal technique for Case 1 duplicates is replaced with a faster and easier-to-understand one.
- In order to answer top-k queries, non-trivial extension techniques are developed.
- The new techniques and the performance of top-k query processing are empirically evaluated.

The rest of the paper is organized as follows: section 2 defines the problem and introduces preliminaries. Section 3 presents the index and the searching phase of our neighborhood generation-based algorithms. Section 4 elaborates the result-fetching phase of the algorithms. The processing of top-k queries is covered by Sect. 5. The techniques to reduce index size are presented in Sect. 6. Section 7 discusses miscellaneous issues, including deletion of characters in the query and updates in data strings. Experimental results and analyses are covered by Sect. 8. Section 9 reviews related work. Section 10 concludes the paper.

2 Preliminaries

2.1 Problem definition

Let Σ be a finite alphabet of symbols; each symbol is also called a character. A string s is an ordered array of symbols drawn from Σ. $|s|$ denotes the length of s. $s[i]$ is the ith character of s, starting from 1. $s[i\,..\,j]$ is the substring of s between positions i and j. Given two strings s and s', $s' \preceq s$ denotes that s' is a prefix of s; i.e., $s' = s[1\,..\,i]$, $1 \le i \le |s|$. The symbol $\circ$ denotes the concatenation of strings.

$ed(s, t)$ returns the edit (Levenstein) distance between two strings s and t, which measures the minimum number of edit operations, including insertion, deletion, and substitution of a character, to transform s to t, or vice versa. It can be computed in $O(|s||t|)$ time and $O(\min(|s|, |t|))$ space using the standard dynamic programming [63]. An efficient thresholded edit distance computation tests whether $ed(s, t) \le \tau$ in $O(\tau \cdot \min(|s|, |t|))$ time [62]. The error-tolerant query autocompletion with edit distance constraints is defined as follows.

Problem 1 Given a collection of data strings S, a query string q, and an edit distance threshold τ, an error-tolerant query autocompletion is to return all the strings $s \in S$, such that $\exists s' \preceq s, ed(s', q) \le \tau$. The results are computed incrementally as the user types in characters.

As the number of results can be large, we may return the top-k ones sorted by a ranking function. We first focus on the case of thresholded queries and then describe the extension to top-k queries.

2.2 Analysis of previous approaches

Chaudhuri and Kaushik [17] and Ji et al. [37] independently developed solutions to processing error-tolerant query autocompletion with edit distance constraints. The techniques proposed in the two studies are similar. In an offline *indexing phase*, data strings are organized in a trie. For online query processing, in the *searching phase*, they maintain the set of the prefixes of the data strings that are within edit distance τ from the query string. The end of these prefixes in the trie are called *active* nodes or *valid* nodes. Whenever a character is appended to the query string, the set of new active nodes is computed using current active ones. In the *result-fetching phase*, the data strings stored under the active nodes are returned as results. The time complexity of processing an input keystroke is $O(\tau \cdot (|A| + |A'|))$, where A and A' are the active node sets before and after inputting a keystroke, respectively. Li et al. [43] improved the method proposed in [37], presenting the notion of *pivotal* active nodes, which are composed of a subset of active nodes with 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 string and the prefix. By considering only pivotal active nodes, it improves the time complexity to $O(\tau \cdot (|P| + |P'|))$, where P and P' are the pivotal active node sets before and after inputting a keystroke. Zhou et al. proposed the BEVA algorithm [77] which achieves minimum active node size by eliminating ancestor–descendant relationship among active nodes. It has a time complexity of $O(\tau + |B| + |B'|)$, where B and B' are the active node sets before and after the keystroke, as defined by the algorithm. Deng et al. proposed the META algorithm [23] that finds active nodes using only matching characters and utilizes a compact tree index to speed up the process of searching active nodes.

We call the above algorithms *direct trie-based approaches* as their indexing methods are to construct a trie directly on data strings. The main drawback of the direct trie-based approaches is the large active node size. Even for BEVA, which minimizes active node size, the size of active nodes can be up to $O(|q|^\tau |\Sigma|^\tau)$ in the worst case. For example, considering a query string “abc” and $\tau = 1$, the active nodes include all the prefixes in the pattern of “?bc” or “a?c”. The problem is even serious for the first few keystrokes of the query, since the set of active nodes includes the prefixes from an enormous subset of data strings. We call this problem *early stage explosion*.

2.3 FastSS

A category of approaches to string similarity queries with edit distance constraints is the *neighborhood generation*-based approaches [9], which generate a set of strings within a certain edit distance from a query string. Among this category, the FastSS [7] algorithm utilizes deletion neighborhood [50] and achieves fast query processing speed for short strings under a small τ. We briefly summarize FastSS in order to best understand our proposed method.

We use $\Delta(s, p)$ to denote the transformation of string s by deleting the character at position p. For example, $\Delta(\texttt{brisbane}, 3) = \texttt{brsbane}$. The deletions can be applied recursively. For a number of deletions k, we use $\Delta(\Delta(\ldots\Delta(s, p_1), p_2), \ldots, p_k)$ to denote the resultant string after k deletions, and call $[\,p_1, p_2, \ldots, p_k\,]$ the *deletion list* of the resultant string. For example, $\Delta(\Delta(\texttt{brisbane}, 2), 2) = \texttt{bsbane}$, and the deletion list is $[\,2, 2\,]$. In order to avoid duplicate deletion lists, a deletion is restricted to occur only after all previous deletions; i.e., $p_{i+1} \geq p_i$.

For a given string s and a number of deletions k, we call x, a resultant string after deleting k characters from s at any possible positions, a *k-variant* of s. The pair $\langle x, D_x\rangle$ is called a *variant-list* pair, where D_x is the deletion list to transform s to x. The union of s' i-variant-list pair ($0 \leq i \leq k$) forms the *k-variant family* of s, denoted by $V(s, k)$.

The following lemma enables us to convert the edit distance constraint to an equivalent condition on variant families.

Lemma 1 (Variant Matching Principle [7]) *Given two strings s and t, $ed(s, t) \leq \tau$, if and only if there exist $\langle x, D_x\rangle \in V(s, \tau)$ and $\langle y, D_y\rangle \in V(t, \tau)$, such that $x = y$ and $|D_x \cup D_y| \leq \tau$.*

Note that multiplicities are considered when computing the union of two multisets: Let $mul(e, D_x)$ denote the multiplicity (number of occurrences) of an element e in a multiset D_x. To take the union of two multisets, the multiplicity of the element e in the result is the larger of $mul(e, D_x)$ and $mul(e, D_y)$. For example, $[\,1, 1, 2\,] \cup [\,1, 2, 2\,] = [\,1, 1, 2, 2\,]$. We call the size of the union of their deletion lists' the *incoordination* of two variants.

Example 1 Consider two strings $s = \texttt{brisbane}$ and $t = \texttt{brosbne}$. The edit distance is 2. They share a common variant "brsbne". The corresponding deletion lists are $[\,3, 5\,]$ and $[\,3\,]$. The incoordination is 2.

For error-tolerant query autocompletion, the variant matching principle can be adapted to handle the case when the edit distance does not exceed τ between a string's prefix and a whole string.

Lemma 2 (Variant Matching Principle for Prefix) *Given two strings s and t, $\exists s' \preceq s$ and $ed(s', t) \leq \tau$, if and only if there exist $\langle x, D_x\rangle \in V(s, \tau)$ and $\langle y, D_y\rangle \in V(t, \tau)$, such that $y \preceq x$ and $|D_x \cup D_y| \leq \tau$.*

Proof We first prove its necessity. Because $ed(s', t) \leq \tau$, by Lemma 1, there exists $\langle x', D'_x\rangle \in V(s', \tau)$ s.t. $x' = y$ and $|D'_x \cup D_y| \leq \tau$. Let $D_x = D'_x$. We delete from s the characters at position $p_i \in D_x$ and obtain the variant x. Because $s' \preceq s$, $x' \preceq x$. Therefore, $y \preceq x$ and $|D_x \cup D_y| \leq \tau$.

Then we prove its sufficiency. Consider a string s' and its variant-list pair $\langle x', D'_x\rangle$, where $x' = y$, $D'_x = \{\,p_i \mid p_i \in D_x, p_i \leq |y|\,\}$. The deleted characters are $s[p_i + i - 1]$ for any $p_i \in D'_x$. Because $D'_x \subseteq D_x$, $|D'_x \cup D_y| \leq \tau$. Hence by Lemma 1, $ed(s', t) \leq \tau$. Because D'_x contains all the deletions in the first $(|x'| + |D'_x|)$ characters of s, and $x' = y \preceq x$, we have $s' = s[1\,..\,|x'| + |D'_x|]$; i.e., $s' \preceq s$. □

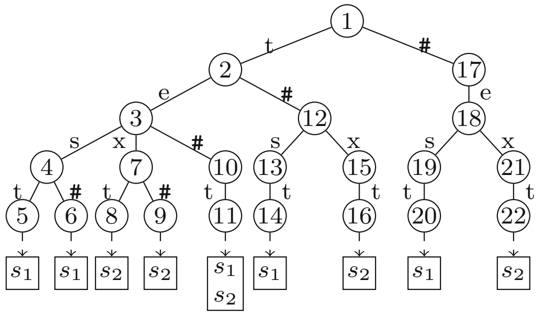


Fig. 1 Index of IncNGTrie ($s_1 = \texttt{test}, s_2 = \texttt{text}, \tau = 1$)

3 Neighborhood generation-based algorithms

3.1 The IncNGTrie algorithm

Based on neighborhood generation, we introduce a new algorithm for error-tolerant query autocompletion.

By Lemma 2, one can design an algorithm to process the query when characters are incrementally appended. The query string's new variants can be easily generated by appending these characters and searching for their matches in the prefixes of data strings' variants. However, incrementally computing incoordination is a challenging task. Variants and their corresponding deletion lists are separately processed in the FastSS algorithm, rendering it difficult to compare variants while computing incoordination at the same time. Seeing this challenge, we resort to *deletion-marked variants*, namely, to use a "#" to denote a character deleted from a data string. For example, "brsbne", a 2-variant of "brisbane", will be represented as "br#sb#ne". A deletion-marked variant combines the string content of a variant and its deletion list. As a result, by scanning the two deletion-marked variants from left to right, we are able to incrementally check whether the contents match and whether the incoordination exceeds the threshold at the same time. Now we design the offline indexing phase of our algorithm: The τ-variant families of all the strings in S are generated and deletion-marked. In order to efficiently process the prefix lookup, we index the deletion-marked variants in a trie. The algorithm is named IncNGTrie, standing for "**Inc**remental **N**eighborhood **G**eneration on a **Trie** index".

Example 2 Consider $S = \{\texttt{test}, \texttt{text}\}$, and $\tau = 1$. Figure 1 shows the trie constructed using the IncNGTrie algorithm. Each path from the root to a leaf node represents a deletion-marked variant of a data string.

By indexing deletion-marked variants, the positions of the deletions on the data strings can be retrieved through the traversal of the trie. They are compared with the deletions enumerated on the query string to get the incoordination. Before presenting the searching phase of the algorithm, we define an *active state*, as a triplet $\langle n, u, \delta \rangle$, where n is a node in the trie, u is called a *cursor*, indicating it is expecting the uth character of the query string, and δ denotes the incoordination that has been encountered. An active state specifies that a variant of the first $(u-1)$ characters of the query string, i.e., $q[1\,..\,u-1]$, matches a prefix of a data string's variant, represented by the path from the root to n with an incoordination of δ. When the user types the uth character of the query string, the active state reads in the character and propagates new active states: First, if n has a child n' through edge label $q[u]$, $\langle n', u+1, \delta \rangle$ becomes active. Second, we may impose a number of deletions at the end of the query string, and thus $\langle n', u+d_q, \delta+d_q \rangle$ becomes active, where $d_q \in [1\,..\,\tau-\delta]$. Third, we may also impose deletions on the data string, and thus $\langle n'', u+d_q, \delta+\max(d_q, d_s) \rangle$ becomes active, where n'' is a descendant of n' through d_s #'s, $d_q \in [1\,..\,\tau-\delta]$ and $d_s \in [1\,..\,\tau-\delta]$.

Following the above active state propagation strategy, we design the searching phase of the IncNGTrie algorithm. The pseudo-code is provided in Algorithm 1.

- It first initializes a set of active states (Line 1). As we are allowed to make at most τ deletions at the beginning of either the query or a data string, the initial active states involve the root of the trie and the nodes that can be reached through no more than τ #'s. Algorithm 2 shows how active states are generated on a node and its descendants through a number of #'s.
- When the user types a keystroke $q[v]$, the algorithm computes new active states with the current ones. For the current active states with $u > v$, they will still be active (Line 6) as the uth character has not come yet. For the other current active states, we include their children through edge label $q[v]$ into the new active states (Line 9). The children's descendants through edge label # are also included, like what we have done to the root and its descendants through # in the initialization step, but the number of available deletions is $\tau-\delta$ here.
- Finally, the string IDs under the nodes of active states whose cursors are $v+1$ will be returned as the results of the query. We dedicate the details of the result-fetching phase in Sect. 4.

Example 3 Consider a query q = `tas`. Table 1 shows the active states for each keystroke using the IncNGTrie algorithm.

Table 1 Active states for q = `tas` using IncNGTrie

Key	∅	`t`	`a`	`s`
Active states	$\langle 1, 1, 0 \rangle$ $\langle 1, 2, 1 \rangle$ $\langle 17, 1, 1 \rangle$ $\langle 17, 2, 1 \rangle$	$\langle 1, 2, 1 \rangle$ $\langle 17, 2, 1 \rangle$ $\langle 2, 2, 0 \rangle$ $\langle 2, 3, 1 \rangle$ $\langle 12, 2, 1 \rangle$ $\langle 12, 3, 1 \rangle$	$\langle 2, 3, 1 \rangle$ $\langle 12, 3, 1 \rangle$	$\langle 13, 4, 1 \rangle$

```
Algorithm 1: IncrementalSearch ($q, \tau, T$)
   Input  : $q$ is a query string input character by character; $\tau$ is an
            edit distance threshold; $T$ is a trie built on the
            deletion-marked $\tau$-variant family of the data strings in
            $S$.
   Output : $s \in S$, such that $\exists s' \preceq s, ed(s', q) \leq \tau$.
1  $A \leftarrow$ PropagateActiveStates ($r, 1, 0, \tau$) ;  /* r: the root
   of T */
2  foreach keystroke $q[v]$ do
3      $A' \leftarrow \emptyset$ ;                   /* new active states */
4      foreach $\langle n, u, \delta \rangle \in A$ do
5          if $u > v$ then
6              $A' \leftarrow A' \cup \{ \langle n, u, \delta \rangle \}$;
7          else
8              if $n$ has a child $n'$ through label $q[v]$ then
9                  $A' \leftarrow A' \cup$
                   PropagateActiveStates ($n', u+1, \delta, \tau$);
10     $A \leftarrow A'$;
11 $R \leftarrow \emptyset$;
12 foreach $\langle n, u, \delta \rangle \in A$ such that $u = v + 1$ do
13     $R \leftarrow R \cup$ string IDs stored on the nodes reachable from $n$;
14 return $R$
```

Now we analyze the worst-case time complexity per keystroke using the IncNGTrie algorithm.[1]

We divide active states in A into two groups: those with $u = |q|$ and those with $u \in [|q|+1\,..\,|q|+\tau]$. For the first group, the time complexity of propagating a state is $O(\max(1, (\tau-\delta)^2))$ in Algorithm 2. The number of states in this group with incoordinations equal to δ is $O(\tau(|q|+\delta)^{\delta})$. Because $\delta \in [0\,..\,\tau]$, the time complexity of processing the first group states is $O(\tau(|q|+\tau)^{\tau})$. For the second group, as they are inserted into new active state set without further actions, the time complexity of processing this group is the scale of the group size; i.e., $O(\tau(|q|+\tau-u+|q|)^{\tau-u+|q|})$ for each u. Combining the costs on the two groups, the time complexity is $O(\tau(|q|+\tau)^{\tau})$.

Compared with the so far most efficient direct trie-based methods, whose time complexity is $O(|q|^{\tau}|\Sigma|^{\tau})$, the runtime cost of IncNGTrie is independent of the alphabet size. Hence

[1] The worst case may happen for the first few keystrokes, when the prefixes of most data strings have their edit distances within τ from the query string.

```
Algorithm 2: PropagateActiveStates($n, u, \delta, \tau$)
Input  : $n$ is a node; $u$ is a cursor; $\delta$ is an incoordination; $\tau$ is
         an edit distance threshold.
Output : A new active state set $A$ propagated from $\langle n, u, \delta \rangle$.
$A \leftarrow \emptyset$;
for $d_q = 0$ to $\tau - \delta$ do      /* $d_q$: deletions on query string */
    $A \leftarrow A \cup \{\langle n, u + d_q, \delta + d_q \rangle\}$;
$d_s \leftarrow 1$ ;      /* $d_s$: deletions on data string */
while $\delta + d_s \leq \tau$ and $n$ has a child $n'$ through label # do
    for $d_q = 0$ to $\tau - \delta$ do
        $\delta' \leftarrow \max(d_q, d_s)$;
        $A \leftarrow A \cup \{\langle n', u + d_q, \delta + \delta' \rangle\}$;
    $n \leftarrow n'$, $d_s \leftarrow d_s + 1$;
return $A$
```

it does not suffer from the early stage explosion. The rationale behind is that all the paths whose edit distance are within τ from the query string share the same variants, and we do not need to activate them respectively in consequence. For the example of a query string "abc" and $\tau = 1$, the paths in the pattern of "?bc" or "a?c", which are inevitably active for direct trie-based methods, will be found by IncNGTrie through only two paths "#bc" and "a#c".

3.2 The IncNGTrie+ algorithm

Although the IncNGTrie algorithm avoids the early stage explosion and achieves a time complexity regardless of the alphabet size, it still propagates redundant active states for the following reasons:

- When applying deletions on the query side, the active states whose cursor values are greater than $|q| + 1$ are also generated, but they bear nothing on the results of the current query. For example, in Table 1, $\langle 2, 3, 1 \rangle$ and $\langle 12, 3, 1 \rangle$ are generated when $q = \texttt{t}$, but we only need the active states whose cursor values are 2 to output results.
- When applying deletions on the data string side, the nodes in the active states are of an ancestor–descendant relationship. For example, in Table 1, when $q = \texttt{ta}$, $\langle 12, 3, 1 \rangle$ is propagated from $\langle 2, 3, 1 \rangle$, which is also active for the current query. Since node 12 is a descendant of node 2, the string IDs reachable from $\langle 12, 3, 1 \rangle$ are a subset of those from $\langle 2, 3, 1 \rangle$, meaning that $\langle 12, 3, 1 \rangle$ can be discarded when we report the results for the current query.
- Multiple active states may reside on the same node. This means that the prefix of a data string's variant may match multiple variants of the query string. It is interesting to see that some matches are better than others in terms of incoordination. For example, a#a matches both a#a and #aa. The former is better because its incoordination is 1 while the latter's is 2. Since incoordinations never decrease as the algorithm runs, some active states can be discarded while not affecting the correctness of the algorithm.

We propose an improved version of the IncNGTrie algorithm, called IncNGTrie+, by addressing the above three issues respectively:

- We only generate active states with cursor values equal to $|q| + 1$.
- We propagate active states in a *lazy* manner to avoid redundant deletions on the data string side. Now deletions are applied on the data string side only when the next keystroke comes.
- We compare active states and pick better ones. Given two active states x and y on the same node n, if for all q, τ, and subtrees rooted at n, the nodes in the active states propagated from x are always a subset of those propagated from y, we say x is *dominated by* y. In this case, x can be safely discarded from the active state set, because all the data strings reachable from x are subsumed by those from y.

With the first improvement, the cursor becomes trivial and can be removed from the active state triplet. However, this will result in difficulty in computing incoordinations when propagating states since the triplet does not keep the content of the query string's variant. Our remedy is to record the difference in the numbers of trailing deletions between the two matching variants. For example, suppose the query string's variant $x = \texttt{a\#\#b\#\#\#}$, and the prefix of a data string's variant $y = \texttt{a\#b\#}$. The difference is 2 because x has 3 trailing deletions, while y has 1. Let $td_{\texttt{query}}$ and $td_{\texttt{data}}$ be the numbers of trailing deletions of the query string and the data string, respectively. Let ω denote their difference; i.e., $\omega = td_{\texttt{query}} - td_{\texttt{data}}$. An active state is then represented by $\langle n, \delta, \omega \rangle$. When we propagate $\langle n, \delta, \omega \rangle$ to a new state $\langle n', \delta', \omega' \rangle$, where n' is a child of n, there are three cases to determine δ' and ω':

- A keystroke matches the edge label to n': $\delta' = \delta$. $\omega' = 0$.
- A deletion is applied on the query string: $\delta' = \delta + 1$, if $\omega \geq 0$; or δ, otherwise. $\omega' = \omega + 1$.
- A deletion is applied on the data string: $\delta' = \delta + 1$, if $\omega \leq 0$; or δ, otherwise. $\omega' = \omega - 1$.

In other words, whether incoordinations increase depends on whether the deletion is applied on the side with more trailing deletions.

Example 4 Consider an active state $\langle n, 5, 2 \rangle$ which represents that the query string's variant a##b### matches a data

Algorithm 3: IncrementalSearchPlus (q, τ, T)

```
   Input  : $q$ is a query string input character by character; $\tau$ is an
            edit distance threshold; $T$ is a trie built on the
            deletion-marked $\tau$-variant family of the data strings in
            $S$.
   Output : $s \in S$, such that $\exists s' \preceq s, ed(s', q) \leq \tau$.
1  $A \leftarrow \{\langle r, 0, 0 \rangle\}$;
2  foreach keystroke $q[v]$ do
3      $A' \leftarrow \emptyset$;
4      foreach $\langle n, \delta, \omega \rangle \in A$ do
5          $\delta' \leftarrow 1$ ;    /* $\omega \geq 0$ always holds for $A$ */
6          if $\delta + \delta' \leq \tau$ and $\langle n, \delta + \delta', \omega + 1 \rangle$ is not dominated by
           any state in $A'$ then
7              remove from $A'$ the states dominated by
               $\langle n, \delta + \delta', \omega + 1 \rangle$;
8              $A' \leftarrow A' \cup \{\langle n, \delta + \delta', \omega + 1 \rangle\}$ ;   /* deletion
               on query string */
9          do
10             if $\delta \leq \tau$ and $n$ has a child $n'$ through label $q[v]$ then
11                 if $\langle n', \delta, 0 \rangle$ is not dominated by any state in $A'$
                   then
12                     remove from $A'$ the states dominated by
                       $\langle n', \delta, 0 \rangle$;
13                     $A' \leftarrow A' \cup \{\langle n', \delta, 0 \rangle\}$ ;   /* match $q[v]$
                       */
14             if $\omega \leq 0$ then $\delta' \leftarrow 1$ else $\delta' \leftarrow 0$;
15             if $n$ has a child $n'$ through label # then
16                 $\langle n, \delta, \omega \rangle \leftarrow \langle n', \delta + \delta', \omega - 1 \rangle$ ;
                   /* deletion on data string */
17             else
18                 break;
19         while true;
20     $A \leftarrow A'$;
21 $R \leftarrow \emptyset$;
22 foreach $\langle n, \delta, \omega \rangle \in A$ do
23     $R \leftarrow R \cup$ string IDs stored on the nodes reachable from $n$;
24 return $R$
```

string's variant a#b#. If the next keystroke is a and n has an outgoing edge a, we have a state $\langle n \circ \texttt{a}, 5, 0 \rangle$, representing that a##b###a matches a#b#a. If a deletion is applied on the query string, we have a state $\langle n, 6, 3 \rangle$, representing that a##b#### matches a#b#. If a deletion is applied on the data string, we have a state $\langle n \circ \#, 5, 1 \rangle$, representing that a##b### matches a#b##.

With the aforementioned three improvements, we design the IncNGTrie+ algorithm, whose searching phase is shown in Algorithm 3. Compared to Algorithm 1, it makes the following modifications to reflect the three improvements.

- There is only one initial active state $\langle r, 0, 0 \rangle$ (Line 1), as opposed to $(\tau + 1)^2$ initial states in Algorithm 1. When processing a keystroke, at most one deletion is imposed on the query string (Lines 5 –8).
- Only when we try to match the current keystroke (Lines 10 –13), a number of 0 to $\tau - \delta$ deletions are imposed on the data string (Lines 14–18).
- Dominated active states are removed (Lines 6 –7 and 11 –12).

Due to the second improvement, the IncNGTrie+ algorithm guarantees that for all the active states in A or A', the number of trailing deletions on the query string is no less than that on the data string. This implies that only the active states with nonnegative ω values will be stored in A and A', though intermediate states with negative ω values may exist during the propagation process (Lines 14 –18). Moreover, when we apply a deletion on the query side, since ω is always nonnegative, we do not have to compare it with 0 to decide the value of δ' (Line 5).

Before inserting an active state x into A', we need to check whether x is dominated by any state already in A', and remove from A' the states dominated by x. The dominance relationship for two active states whose $\omega \geq 0$ is equivalent to the condition stated in the following lemma.

Lemma 3 *Given two active states $\langle n, \delta_1, \omega_1 \rangle$ and $\langle n, \delta_2, \omega_2 \rangle$ such that $\omega_1 \geq 0$ and $\omega_2 \geq 0$, $\langle n, \delta_1, \omega_2 \rangle$ is dominated by $\langle n, \delta_2, \omega_2 \rangle$, if and only if $\delta_1 \geq \delta_2$ and $\delta_1 - \delta_2 \geq \omega_1 - \omega_2$.*

Proof We abuse n to denote both the node and the string from the root to n. Let $\rightarrow$ denote that an active state is propagated to another. Let a character $c = q[v]$.

We first prove the necessity by contradiction. If $\delta_1 < \delta_2$, we apply $\tau - \delta_1$ #'s on data string and then match through keystroke c. $\langle n, \delta_1, \omega_1 \rangle \rightarrow \langle n \circ d \circ c, \tau, 0 \rangle$, where d consists of $\tau - \delta_1$ #'s. Because $\delta_1 < \delta_2$, $\delta_2 + \tau - \delta_1 > \tau$. This means that the node $n \circ d \circ c$ cannot be propagated from $\langle n, \delta_2, \omega_2 \rangle$, thus contradicting that $\langle n, \delta_1, \omega_1 \rangle$ is dominated by $\langle n, \delta_2, \omega_2 \rangle$. If $\delta_1 - \delta_2 < \omega_1 - \omega_2$, there are two cases: $\omega_1 \leq \omega_2$ and $\omega_1 > \omega_2$. If $\omega_1 \leq \omega_2$, $\delta_1 < \delta_2$. This case has been proved. So we only need to prove the case when $\omega_1 > \omega_2$. We apply $\omega_1 + \tau - \delta_1$ deletions on data string and then match through keystroke c. $\langle n, \delta_1, \omega_1 \rangle \rightarrow \langle n \circ d \circ c, \tau, 0 \rangle$, where d consists of $\omega_1 + \tau - \delta_1$ #'s. Because $\delta_1 - \delta_2 < \omega_1 - \omega_2$, $\omega_1 > \omega_2$, and $\tau \geq \delta_1$, we have $\delta_2 + \max(0, \omega_1 + \tau - \delta_1 - \omega_2) > \tau$. This means that the node $n \circ d \circ c$ cannot be propagated from $\langle n, \delta_2, \omega_2 \rangle$, thus contradicting that $\langle n, \delta_1, \omega_1 \rangle$ is dominated by $\langle n, \delta_2, \omega_2 \rangle$.

Then we prove the sufficiency by induction; i.e., given two active states $\langle n, \delta_1, \omega_1 \rangle$ and $\langle n, \delta_2, \omega_2 \rangle$ such that $\omega_1 \geq 0$, $\omega_2 \geq 0$, $\delta_1 \geq \delta_2$, and $\delta_1 - \delta_2 \geq \omega_1 - \omega_2$, the new active states propagated from them will reside on the same node and also satisfy the above condition. Because the two states are both on n, the base case holds. For the inductive step, there are three cases of node propagation in Algorithm 3:

- A match occurs through keystroke c. Because $\delta_1 \geq \delta_2$, $\langle n, \delta_1, \omega_1 \rangle \rightarrow \langle n \circ c, \delta'_1, \omega'_1 \rangle$, and $\langle n, \delta_2, \omega'_2 \rangle \rightarrow \langle n \circ c, \delta'_2, \omega'_2 \rangle$, where $\delta'_1 = \delta_1$, $\omega'_1 = 0$, $\delta'_2 = \delta_2$, and $\omega'_2 = 0$. The new states are both on $n \circ c$, and satisfy that $\omega'_1 \geq 0, \omega'_2 \geq 0, \delta'_1 \geq \delta'_2$, and $\delta'_1 - \delta'_2 \geq \omega'_1 - \omega'_2$.
- A deletion is applied on the query string. When $\delta_1 + 1 \leq \tau$, $\langle n, \delta_1, \omega_1 \rangle \rightarrow \langle n, \delta'_1, \omega'_1 \rangle$, where $\delta'_1 = \delta_1 + 1$, and $\omega'_1 = \omega_1 + 1$. Because $\delta_1 \geq \delta_2$, $\delta_2 + 1 \leq \tau$. Therefore, $\langle n, \delta_2, \omega_2 \rangle \rightarrow \langle n, \delta'_2, \omega'_2 \rangle$, where $\delta'_2 = \delta_2 + 1$, and $\omega'_2 = \omega_2 + 1$. The new states are both on n, and satisfy that $\omega'_1 \geq 0, \omega'_2 \geq 0, \delta'_1 \geq \delta'_2$, and $\delta'_1 - \delta'_2 \geq \omega'_1 - \omega'_2$.
- $p \in [1 \mathinner{..} \tau]$ deletions are applied on the data string and then a match occurs through keystroke c. When $\delta_1 + \max(0, p - \omega_1) \leq \tau$, $\langle n, \delta_1, \omega_1 \rangle \rightarrow \langle n \circ d \circ c, \delta'_1, \omega'_1 \rangle$, where d consists of p #'s, $\delta'_1 = \delta_1 + \max(0, p - \omega_1)$, and $\omega'_1 = 0$. Because $\omega_1 \geq 0$, $\omega_2 \geq 0$, $\delta_1 \geq \delta_2$, and $\delta_1 - \delta_2 \geq \omega_1 - \omega_2$, we have $\delta_1 + \max(0, p - \omega_1) - (\delta_2 + \max(0, p - \omega_2)) =$

$$\begin{cases} \delta_1 - \delta_2 \geq 0, & \text{when } p < \omega_1 \text{ and } p < \omega_2; \\ \delta_1 - \delta_2 + p - \omega_1 \geq 0, & \text{when } p \geq \omega_1 \text{ and } p < \omega_2; \\ \delta_1 - \delta_2 - p + \omega_2 \geq 0, & \text{when } p < \omega_1 \text{ and } p \geq \omega_2; \\ \delta_1 - \delta_2 - \omega_1 + \omega_2 \geq 0, & \text{when } p \geq \omega_1 \text{ and } p \geq \omega_2. \end{cases}$$

  Therefore, $\delta_1 + \max(0, p - \omega_1) \geq \delta_2 + \max(0, p - \omega_2)$. Because $\delta_1 + \max(0, p - \omega_1) \leq \tau$, $\delta_2 + \max(0, p - \omega_2) \leq \tau$. Therefore, $\langle n, \delta_2, \omega_2 \rangle \rightarrow \langle n \circ d \circ c, \delta'_2, \omega'_2 \rangle$, where $\delta'_2 = \delta_2 + \max(0, p - \omega_2)$, and $\omega'_2 = 0$. The new states are both on $n \circ d \circ c$, and satisfy that $\omega'_1 \geq 0$, $\omega'_2 \geq 0$, $\delta'_1 \geq \delta'_2$, and $\delta'_1 - \delta'_2 \geq \omega'_1 - \omega'_2$.

□

Since the lemma gives a necessary and sufficient condition of active state dominance for nonnegative ω values, we may easily check dominance relationship using the conditions in the lemma and guarantee that no active state is dominated by another when calling Line 20 in Algorithm 3. Hence we have a salient property for the IncNGTrie+ algorithm:

Corollary 1 *By the IncNGTrie+ algorithm, the set of active states resident on every node n is minimal when the propagation is finished for any keystroke.*

Proof Given an active state $\langle n, \delta, \omega \rangle$, we can convert it to a 2-dimensional point $(\delta, \delta - \omega)$. The dominance relationship of active states in Lemma 3 is then converted to the dominance relationship of points: $(\delta_1, \delta_1 - \omega_1)$ dominates $(\delta_2, \delta_2 - \omega_2)$, if and only if $\delta_1 \leq \delta_2$ and $\delta_1 - \omega_1 \leq \delta_2 - \omega_2$. For the group of points such that $\delta_1 = \delta_2$ and $\delta_1 - \omega_1 = \delta_2 - \omega_2$, the IncNGTrie+ algorithm keeps at most one of them. Therefore, the IncNGTrie+ algorithm finds the *skyline* [8] points among all the points converted from the active states formed by matching a query string's variant and the string from the root to n. Since the skyline is a minimal set of points such that all the other points are dominated, the set of active states is minimal for node n. □

Table 2 Active states for $q =$ `tas` using IncNGTrie+

Key	∅	`t`	`a`	`s`
Active states	$\langle 1, 0, 0 \rangle$	$\langle 1, 1, 1 \rangle$ $\langle 2, 0, 0 \rangle$	$\langle 2, 1, 1 \rangle$	$\langle 13, 1, 0 \rangle$

Example 5 Recall Example 3. Table 2 shows the active states for each keystroke using the IncNGTrie+ algorithm. Compared to the IncNGTrie algorithm, the total number of active states is reduced from 13 to 5.

Since dominance check is invoked multiple times in the IncNGTrie+ algorithm, we briefly discuss how to implement it efficiently. Because δ is in the range of $[0 \mathinner{..} \tau]$ and ω never exceeds δ (otherwise the trailing deletions will result in greater δ), there are at most $(\tau + 1)(\tau + 2)/2$ possible (including dominated) active states for a node n. Since τ is a small number in most applications, we may offline compute the skyline for every possible combination of active states out of the $2^{(\tau+1)(\tau+2)/2}$ possibilities, and store them for online query processing. In doing so, we do not physically remove any state in A' but record all the propagated states for each node in a bit array of $(\tau + 1)(\tau + 2)/2$ bits, and then use the stored information to find the skyline states.

Another optimization is to collapse the nodes whose incoming edges are #, because there is plenty of index access for patterns like `#a`, `##a`, etc. (Lines 10 –18, Algorithm 3) in the IncNGTrie+ algorithm. We may encode `#a` (and `##a`, etc.) using a single character. Then for each node n whose incoming edge label is #, we may directly connect its parents and its children through an edge and remove n from the trie. For example, in Fig. 1, nodes 1 and 18 are connected through an edge `#e`, which is encoded by a single character, and then node 17 is removed. In doing so, the index access for descendants through edge label # is reduced. For ease of exposition, we still show the nodes whose incoming edge labels are # in the trie in the rest of the paper. They can be regarded as *virtual* nodes.

We analyze the worst-case time complexity of the IncNGTrie+ algorithm per keystroke. The time complexity of propagating an active state is $O(\max(1, \tau - \delta))$, $\delta \in [0 \mathinner{..} \tau]$. The number of active states with incoordination equal to δ is $O(\tau |q|^{\delta})$. The time complexity of the IncNGTrie+ algorithm is thus $O(\tau |q|^{\tau})$.

4 Fetching query results

We return as results the strings stored on the nodes reachable from the active states (with cursor equal to $|q|+1$ for IncNGTrie). One may notice that due to neighborhood generation, duplicates may exist in these string IDs. It is noteworthy to mention that duplicate results also exist for direct trie-based methods; For example, an active node is an ancestor of another and subsumes the string IDs under the latter, but most of the previous works did not discuss the removal of them. In this section, we investigate how to efficiently eliminate duplicates in the results of error-tolerant query autocompletion for neighborhood generation-based algorithms, and introduce the detailed result fetching phase.

The duplicates come from three sources. We use the example in Fig. 1 to illustrate them:

- **Case 1** The string IDs reachable from a node contain duplicates. For example, in Fig. 1, node 3 has five leaf descendants, yet they report only two results s_1 and s_2. These duplicates are caused by neighborhood generation on the data strings. The path from a node to its leaf descendants may differ due to the existence of deletions, but reach the same strings eventually.
- **Case 2** The nodes of two active states are of ancestor–descendant relationship. For example, node 3 is an ancestor of node 7, thereby subsuming the results from the latter. They are caused by neighborhood generation on the query string. Two variants match a path and its prefix in the trie, respectively.
- **Case 3** The nodes of two active states are not of ancestor–descendant relationship but still share common string IDs. For example, nodes 4 and 19 reach the same result s_1. They are caused by neighborhood generation on both the query and the data strings, which makes them share more than one variants.

Next we present our method to respectively deal with the three cases of duplicates.

4.1 Eliminating case 1 duplicates

To report all the distinct string IDs under a node, one feasible solution is to store in an array the string IDs linked to the end of paths in the trie, and equip each node with two pointers marking the range in the array containing the underlying string IDs (we call it *result-fetching range*). For example, consider the trie in Fig. 1 and the corresponding string ID array (denoted by B) in Fig. 2. The result fetching range of node 7 is $[3, 4]$. Hence the results from node 7 consist of the string IDs in $B[3, 4]$. The problem is then equivalent to a colored range listing problem [51] which returns distinct elements in an array. It can be solved in $O(1)$ time per distinct string, consuming $O(|B| \log |B|)$ bits, besides the string ID array and the result-fetching ranges. Next we propose a solution specific to our problem which needs no additional space but still reports each distinct string in $O(1)$ time.

Position:	1	2	3	4	5	6	7	8	9	10
String ID:	s_1	s_1	s_2	s_2	s_1	s_2	s_1	s_2	s_1	s_2

Fig. 2 String ID array for the trie in Fig. 1

Since Case 1 duplicates are caused by the existence of deletions in the paths from a node to its descendants, we observe the following facts:

- A node n and its child through # report the same results, except for the string IDs directly attached to n [2], e.g., both node 3 and node 10 in Fig. 1 reaching s_1 and s_2.
- If a node has no # in the paths to its descendants, there is no duplicate in its result-fetching range, e.g., node 17 in Fig. 1.

Based on the above observations, the results for a node can be found as follows: We repeat the process of reporting the string IDs directly attached to a node and going through # to its descendants until no # can be found, and then report the string IDs in the result-fetching range of the current descendant. The pseudo-code of the algorithm is shown in Algorithm 4, which guarantees no duplicates fetched for a given node n.

Proof *(Correctness of Algorithm 4)* Suppose the input node of Algorithm 4 is n_1, and there is a path $n_1, \ldots, n_k$ through label # in the trie, where there is no outgoing # from n_k. Algorithm 4 follows this path and outputs the following string IDs: (1) those directly attached to $n_1, \ldots, n_{k-1}$ and (2) those in the reporting fetching range of n_k. Let R_i denote the multiset of string IDs fetched from node n_i by Algorithm 4. To prove the correctness, we show that (1) there is no duplicates in any R_i, $1 \le i \le k$; and (2) $R_i \cap R_j = \emptyset$, if $i \neq j$.

We first prove (1). For R_i, $i < k$, because there is only one path from the root to any n_i in the trie, two deletion-marked variants of a string cannot both match (# must match # in the trie) the path. Therefore, there is no duplicates in the string IDs directly attached to n_i. For R_k, because there is no outgoing # from n_k, by the definition of deletion neighborhood, there is no # in the subtree rooted at n_k (i.e., we have encountered τ #'s from the root to n_k). Therefore, any two paths from n_k reach different strings, and thus there is no duplicates in n_k's result-fetching range.

We then prove (2). By the definition of deletion neighborhood, the strings directly attached to n_i are the strings ending at n_i ($i < k$), and the strings in the report fetching range of

[2] This exception happens when n is the end of a data string.

Algorithm 4: GetStrings (n)

```
Input  : $n$ is a node.
Output : The string IDs under $n$ without duplicates.
$R \leftarrow \emptyset$;
do
    $R \leftarrow R \cup$ string IDs directly attached to $n$;
    if $n$ has a child $n'$ through label # then
        $n \leftarrow n'$;
    else
        break;
while true;
$[i, j] \leftarrow n$'s result-fetching range;
$R \leftarrow R \cup B[i, j]$;
return $R$
```

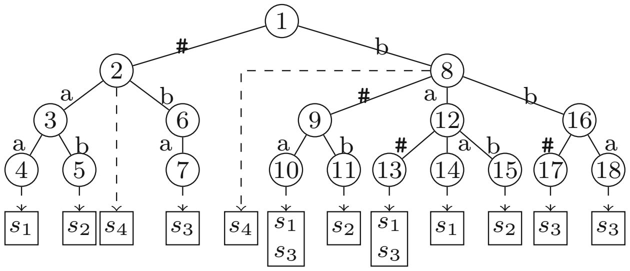


Fig. 3 Index of IncNGTrie (s_1 = baa, s_2 = bab, s_3 = bba, s_4 = b, $\tau = 1$)

n_k are at least as long as the path from the root to n_k. If $i \neq j$, R_i and R_j are different in lengths, and thus $R_i \cap R_j = \emptyset$. □

Example 6 Consider node 8 in Fig. 3. To fetch its underlying results, we first report its attached string s_4. Then we go through # to node 9, which has no outgoing #. We report the strings in node 9's result-fetching range [6, 8]: s_1, s_3, and s_2. No duplicate occurs.

Algorithm 4 still needs to go through a number of #'s. It can be done in a more efficient way. Consider a node n and its descendants $n_1, \ldots, n_k, n'$ via #, such that n' has no outgoing #. We may update the result-fetching range of n by copying from n' and then including the ranges for the strings directly attached to $n, n_1, \ldots, n_k$. Then the string IDs can be accessed directly from n's result-fetching range without producing any duplicates.

Example 7 Consider Example 6. We update the result fetching range of node 8 by copying from node 9, which is [6, 8], and then including the range of its attached strings, which is [5, 5]. The new result-fetching range of node 8 is [5, 8].

4.2 Eliminating case 2 duplicates

For ease of exposition, we call a node in the trie a *reporting* node, if it is in an active state (with cursor equal to $|q| + 1$ for IncNGTrie), and none of its ancestors is in an active state (with cursor equal to $|q| + 1$ for IncNGTrie). In other words, reporting nodes are those whose underlying string IDs are not subsumed by others among the nodes that we use to fetch results. Case 2 duplicates can be avoided by processing only reporting nodes. To check whether a node n is a reporting node, one solution is to assign additional codes (e.g., the region codes widely used in XML query processing [74]) to trie nodes and test the ancestor–descendant relationship between n and every other node in the active state set A. This method needs at most $|A| - 1$ ancestor–descendant relationship tests per node. Another solution is to maintain the nodes in the active states in a hash table, and test if none of n's ancestors is in the table. Due to the edit distance constraint, only its ancestors on level $|q| - \tau$ or below need to be tested, thus taking at most 2τ hash table lookup per node. Rather than choose these methods, we propose a method that runs in $O(\log |N_r|)$ per node check by exploiting the order in which active states are arranged. N_r denotes the set of reporting nodes and its size is usually very small.

When propagating active states, we organize them in the order of node numbers in pre-order traversal (see Fig. 1 for an example). This asserts that a node n's ancestor must appear before n when they are accessed. Accordingly, a reporting node verification algorithm can be devised: The nodes in the active states are processed one by one. A binary search tree (BST) is utilized to keep reporting nodes based on their numbers in pre-order traversal. To verify a reporting node n, we search the BST for the node whose number is smaller and closest to n. If the returned node is an ancestor of n, n is not a reporting node; otherwise n is a reporting node and will be added to the BST.

Algorithm 5 presents the pseudo-code testing if a node n_i is a reporting node. We abuse n_i to denote its node number in pre-order traversal. The algorithm begins with a search in the BST for the node whose number is just no more than n_i. If it returns n_i, n_i has been processed and thus is skipped. Otherwise, the returned node n_j is checked for ancestor–descendant relationship with n_i. The node numbers of n_j's descendants are in the range from $n_j + 1$ to $n_k - 1$, where n_k is n_j's next sibling [3]. If n_i is within this range, n_j is an ancestor of n_i and the algorithm returns false. Otherwise, n_i is a reporting node and is inserted into the BST. The search operation in the BST runs in $O(\log |N_r|)$ time. The ancestor–descendant check runs in $O(1)$ time. The overall time complexity of the algorithm is $O(\log |N_r|)$.

Proof *(Correctness of Algorithm 5)* We prove by contradiction. We assume n_j, the node returned from the BST, is not an ancestor of n_i, but there exists another reporting node n'_j which is n_i's ancestor. According to the order in which we

[3] In case n_j is the last child, we recursively go up the tree until reaching an ancestor such that it has a next sibling, and then use the next sibling as n_k.

Algorithm 5: CheckReportingNode (n_i, T_r)

```
Input  : $n_i$ is a node in an active state. $T_r$ is a binary search tree
         maintaining the reporting nodes seen so far.
Output : true, if $n_i$ is a reporting node and appears for the first
         time; or false, otherwise.
$n_j \leftarrow T_r.search(n_i)$ ;                    /* $n_j \le n_i$ */
if $n_i = n_j$ then
    return false
$n_k \leftarrow n_j$'s next sibling;
if $n_j + 1 \le n_i$ and $n_k - 1 \ge n_i$ then
    return false
else
    $T_r.insert(n_i)$;
    return true
```

arrange active states, n'_j must be accessed before n_i and thus $n'_j < n_j < n_i$. Because n'_j is n_i's ancestor, its next sibling n'_k satisfies $n'_k - 1 \ge n_i > n_j$. Therefore, $n'_j + 1 \le n_j < n'_k - 1$, meaning that n'_j is an ancestor of n_j. It contradicts that n_j is a reporting node. Hence the correctness of the algorithm is proved. □

Example 8 Consider the trie in Fig. 1 and a query string `te`. Suppose we use the IncNGTrie algorithm. The active states with cursors equal to $|q| + 1$ are $\langle 2, 3, 1 \rangle$, $\langle 3, 3, 0 \rangle$, $\langle 10, 3, 1 \rangle$, $\langle 12, 3, 1 \rangle$, and $\langle 18, 3, 1 \rangle$ in the node order. First, node 2 is a reporting node and inserted to the BST. For nodes 3, 10, and 12, the search in the BST returns 2 and it is an ancestor of them. So they are not reporting nodes. For node 18, since node 2 in the BST is not an ancestor of 18, 18 is a reporting node and inserted to the BST. Finally, nodes 2 and 18 are returned as reporting nodes.

4.3 Eliminating case 3 duplicates

To handle Case 3 duplicates, we use a hash table to record the string IDs to be returned as results. Since a result string can appear under at most $|N_r|$ reporting nodes, the worst-case time complexity of reporting all the results is $O(|R||N_r|)$, where R denotes the set of result strings.

5 Processing top-*k* queries

In this section, we introduce the extension of our algorithms to support top-k queries of error-tolerant query autocompletion.

To define the ranking function, we choose the same method as in [17] by combining the static score and string similarity in a monotonic fashion. We assume that each data string s is assigned with an application specified *static score*, denoted by $sscore(s)$. For example, in a dictionary of product names, a static score may reflect the popularity of a product. We focus on the following ranking function that gives an overall score of a data string s with respect to the query q, but our method can be extended to support other monotonic functions such as a linear combination of the two components.

$$F(s, q) = sscore(s) \cdot sim(s, q). \tag{1}$$

$sim(s, q)$ denotes the similarity between s and q. It is defined as $\max\{\, 1 - \frac{ed(s',q)}{|q|} \mid s' \preceq s \,\}$; i.e., we pick the prefix of s which has the smallest edit distance to q, normalize the distance by the length of q, and then convert the distance measure to a similarity measure. The data strings are ranked by $F(s, q)$, and the highest k results are returned. Threshold semantics is optional to enforce that the results are similar to the query, as adopted in [16,44].

5.1 Indexing and searching

We assume that the threshold semantics is not involved. In the absence of the threshold, we may build the index by enumerating the γ-variant family of data strings, where γ is used to control the number of deletions. In the searching phase, we use either IncNGTrie or IncNGTrie+ to compute active states, and switch to edit distance computation (see Sect. 6.1 for the correctness of this switch), as do direct trie-based methods, when the γth # in a path is seen. Because incoordinations may vary from 0 to $|q|$ when the threshold is not given, the number of active states may be huge when the query becomes long. To remedy this, we may generate only a small set of promising active states while the others are guaranteed not to produce top-k results. For ease of exposition, we introduce the result-fetching algorithm first and then elaborate on this optimization.

5.2 Result fetching

The basic framework of result fetching is to obtain the strings under the node in each active state as well as their static scores. The similarity can be computed using the incoordination of the active state. Hence the $F(s, q)$ score is obtained and we update the top-k results encountered so far (called temporary results). However, it is inefficient to fetch all the strings for all the active states and then sort them by the ranking function due to the potentially huge number of underlying strings. A direct trie-based method was proposed in [17] by precomputing the top-k data strings by static score for each node of the trie. Although the method can be extended to IncNGTrie or IncNGTrie+, we do not choose to return results in this way because it imposes considerable space overhead and only supports a static k. Our solution is based on two early termination techniques and one initialization technique. Next we choose the IncNGTrie+ algorithm to describe the techniques, and they can be applied to the IncNGTrie algorithm as well.

5.2.1 Early termination

The first early termination technique is to reduce the number of active states for result fetching. To this end, for each node n we store the maximum static score of its underlying strings, denoted by $max\_sscore(n)$. Given an active state $\langle n, \delta, \omega \rangle$, the maximum $F(s, q)$ of its underlying strings is

$$F_{\max}(n, q) = max\_sscore(n) \cdot \left(1 - \frac{\delta}{|q|}\right).$$

With this upper bound, we can organize the active states in the order of $F_{\max}(n, q)$, and break tie by the order of node number in pre-order traversal for the sake of duplicate removal (will be discussed later). We scan active states in this order, fetch string IDs as well as static scores, and update the temporary top-k results. When the $F_{\max}(n, q)$ value of the next active state is no higher than the $F(s, q)$ value of the kth temporary result, the process can be stopped, and we return the current top-k as the final results.

Duplicate removal can be applied to improve efficiency. We can use the techniques proposed in Sect. 4, but a modification is necessary to remove Case 2 duplicates. Given two active states $\langle n, \delta, \omega \rangle$ and $\langle n', \delta', \omega \rangle$ where n is an ancestor of n', unlike the non-top-k case in Sect. 4, we cannot simply skip the result fetching of n'. Although the underlying string IDs of n' are subsumed by those of n, they may achieve higher overall score when $\delta' < \delta$. Hence we modify the definition of a reporting node for top-k queries: A node is a reporting node, if it is in an active state, and none of its ancestors is in an active state with a smaller or equal incoordination. Because $max\_sscore(n') \leq max\_sscore(n)$, and the active states are sorted by decreasing $F_{\max}(n, q)$ and increasing node number, $\langle n', \delta', \omega \rangle$ is processed prior to $\langle n, \delta, \omega \rangle$ only if $\delta' < \delta$. With this property, we can verify a reporting node n by comparing n and δ only with the reporting nodes that have been processed. This can be done with an interval tree. For each reporting node n' that has been seen, we take as key the node number range of its descendants, i.e., $[n' + 1, n'' - 1]$ (n'' is the next sibling of n'), and as value the corresponding incoordination. It takes $O(\log |N_r| + occ)$ time to check whether n is in any range with a no larger incoordination.

Example 9 Consider the example in Fig. 3. Suppose the static scores of s_1 to s_4 are 0.5, 0.8, 0.4, and 0.2, respectively. Suppose $k = 2$, the query length is 4, and we have five active states: $\langle 12, 2, 2 \rangle$, $\langle 13, 2, 1 \rangle$, $\langle 14, 1, 0 \rangle$, $\langle 15, 1, 1 \rangle$, and $\langle 18, 1, 0 \rangle$. The $max\_sscore(n)$ values of their nodes are 0.8, 0.8, 0.5, 0.8, and 0.4, respectively. The $F_{\max}(n, q)$ values are 0.4, 0.4, 0.375, 0.6, and 0.3, respectively. So we process them in this order: $\langle 15, 1, 1 \rangle$, $\langle 12, 2, 2 \rangle$, $\langle 13, 2, 1 \rangle$, $\langle 14, 1, 1 \rangle$, and $\langle 18, 1, 0 \rangle$.

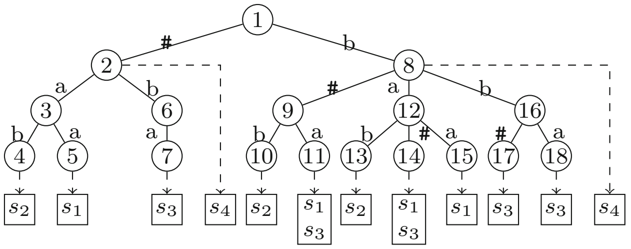
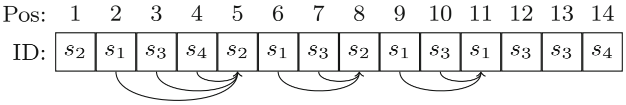


Fig. 4 Sorted trie

- $\langle 15, 1, 1 \rangle$: Node 15 is a reporting node and we get the first temporary result s_2 whose $F(s, q) = 0.6$. Since node 15 has no descendants, no operation is done to the interval tree.
- $\langle 12, 2, 1 \rangle$: Node 12 is a reporting node and we get the second temporary result s_1 whose $F(s, q) = 0.25$. Then the key-value pair $\langle[13, 15], 2\rangle$ is inserted into the interval tree.
- $\langle 13, 2, 1 \rangle$: Node 13 is not a reporting node because it is in the indexed range [13, 15] and the incoordination is equal to the indexed value.
- $\langle 14, 1, 1 \rangle$: Although node 14 is in the indexed range [13, 15], it has a smaller incoordination, and thus it is a reporting node. We update the second temporary result s_1 whose $F(s, q)$ becomes 0.375. No operation is done to the interval tree due to the absence of descendants.
- $\langle 18, 1, 0 \rangle$: Its $F_{\max}(n, q) = 0.3$, less than the lowest temporary result 0.375. We stop the process and return s_2 and s_1 as top-2 results.

The second early termination technique is to reduce the number of string IDs accessed for each node in an active state. In Sect. 4, each node is equipped with a result fetching range, and we access the string IDs in the range one by one. If the entries in the string ID array are sorted, we will be able to early terminate the access in the range. Based on this idea, we sort siblings in the trie by decreasing $max\_sscore(n)$, and the entries in the string ID array are sorted accordingly. If an internal node has directly attached string IDs, they are regarded as a special child and also involved in the sorting. As a result, the first entry in the result-fetching range has the maximum static score and hence maximum $F(s, q)$ among all the underlying strings of a node (because the similarity is specified by the incoordination of an active state and fixed throughout the result-fetching range). Consider Example 9. Figure 4 shows the trie after sorting.

The next step is for each entry $B[i]$ in the string ID array, we use a pointer linking it to $B[j]$, such that j is the smallest value $j > i$ and $sscore(B[j]) > sscore(B[i])$ (i.e., it is



Pos:	1	2	3	4	5	6	7	8	9	10	11	12	13	14
ID:	s_2	s_1	s_3	s_4	s_2	s_1	s_3	s_2	s_1	s_3	s_1	s_3	s_3	s_4

Fig. 5 String ID array with links. An arrow represents the link to the next entry with a higher static score. Entries without outgoing links shown are linked to null

linked to the next entry which has a higher static score); or null if there is no such $B[j]$. In consequence, we can fetch results for a node n as follows: We scan the entries in the range and update the temporary top-k results. Whenever the kth temporary result is updated or a string with a lower static score than the kth temporary result is seen, we use the link to quickly find the next entry with a higher static score. If the link points to null or goes beyond the range, the process can be early terminated, because it is guaranteed that no subsequent strings in the range are better than the kth temporary result. Assuming the maximum string length in S is $|s|$, there are at most $(k + \frac{k(k-1)\max(|s|-1,\gamma)}{2})$ entries to be accessed in a result fetching range.

Example 10 Figure 5 shows the string ID array with links for the example in Fig. 4. Suppose the temporary results are empty and we are going to fetch results for active state $\langle 2, 2, 0 \rangle$. The result-fetching range of node 2 is $[1, 4]$. s_2 is fetched and become the first temporary result with an $F(s, q)$ of 0.4. Then s_1 is fetched and become the second temporary result with an $F(s, q)$ of 0.25. Since the second temporary result is updated, we follow the link and go to $B[5]$. It is beyond the result-fetching range and hence we stop.

There are two optimizations on sorting: Due to the duplicate removal for Case 1 duplicates, if a node n has an outgoing #, we only need to order at most two of its children: the one with the label # and the special child for directly attached string IDs, because the underlying strings of the other children are not in n's result-fetching range, hence bearing no effect when we are going to fetch the strings under n. In addition, it is not necessary to physically sort the trie. Instead, we can sort the string ID array only and update the result-fetching ranges of the trie nodes.

5.2.2 Initialization

The early termination can be sped up if we find a good set of initial temporary results. Since active states are sorted by $F_{\max}(n, q)$ and siblings are (virtually) sorted by $max\_sscore(n)$ in the trie, the two orders can be exploited for this purpose. We pick top-k active states with distinct $max\_sscore(n)$, and take the first entries in their result-fetching ranges as top-k initial results. Despite the existence of duplicates in the string ID array, the distinctness of $max\_sscore(n)$ guarantees that top-k initial results can be found.

Algorithm 6: FetchTopK (q, A, B, k)

```
Input  : $q$ is a query string; $A$ is an ordered set of active states;
         $B$ is a string ID array; $k$ is the number of results to be
         reported.
Output : A set of strings ranked by $F(s, q)$.
1  $R \leftarrow$ InitializeTopK $(q, A, B, k)$;
2  $T_i \leftarrow \emptyset$ ;                 /* $T_i$ is an interval tree */
3  foreach $\langle n, \delta, \omega \rangle \in A$ do
4      if $F_{\max}(n, q) \leq F(R[k], q)$ then
5          break ;                          /* early terminate */
6      $f \leftarrow$ true;
7      foreach $\langle [i, j], \delta' \rangle \in T_i$ such that $n \in [i, j]$ do
8          if $\delta \geq \delta'$ then
9              $f \leftarrow$ false ;      /* not reporting node */
10             break;
11     if $f$ = true then
12         $T_i.insert(\langle [n+1, n\text{'s next sibling} - 1], \delta \rangle)$;
13         $[i, j] \leftarrow n$'s result-fetching range;
14         $R \leftarrow$ GetStringsTopK $(q, B, k, [i, j])$;
15 return $R$
```

Algorithm 7: GetStringsTopK *(q, B, k, [i, j])*

```
Input  : $q$ is a query string; $B$ is a string ID array; $k$ is the
         number of results to be reported; $[i, j]$ is a
         result-fetching range.
Output : A set of temporary results in $B[i, j]$.
$p \leftarrow i$;
while $p \leq j$ and $p \neq$ null do
    if $F(B[p], q) > F(R[k], q)$ then
        $R.add(B[p])$ ;  /* update temporary results
        */
    if $F(B[p], q) \leq F(R[k], q)$ then
        $p \leftarrow B[p].next$ ;        /* jump with link */
    else
        $p \leftarrow p + 1$;
return $R$
```

By integrating the two early termination techniques and the initialization technique, a result-fetching algorithm for top-k queries is devised (pseudo-code shown in Algorithm 6). Given a set of active states, a set of temporary results is initialized (Line 1). Then for each active state, it checks if its node is a reporting node (Lines 6 –10). If so, its result fetching range and incoordination are inserted into the interval tree (Line 12), and its underlying strings are fetched to update the temporary results (Line 14, details shown in Algorithm 7). The above process terminates once the next active state cannot yield a better one than the kth temporary result (Line 4), and then the top-k results are returned.

5.3 Searching revisited

Finally, we introduce the aforementioned optimization technique in the searching phase to reduce active states. The basic idea is to "hibernate" the active states that cannot produce top-k results for the current query, and "awake" them only when the next few keystrokes come, and these active states become necessary to compute top-k. The initialization of temporary results is leveraged for this purpose. If the $F_{\max}(n, q)$ value of an active state is lower than the $F(s, q)$ value of the kth initial temporary result, we can make sure that the active state cannot produce a top-k result of the current query, and thus it is turned into hibernation. When the next keystroke comes, we check whether to awake it by exploiting this observation: For all the active states $\langle n', \delta', \omega' \rangle$ propagated from $\langle n, \delta, \omega \rangle$, $max\_sscore(n') \leq max\_sscore(n)$ and $\delta' \geq \delta$. This means that $F_{\max}(n, q)$ monotonically decreases with the active state propagation. On the basis of this property, we compute the $F_{\max}(n, q)$ value of a hibernating active state with respect to the current query length. The state is awaken if its $F_{\max}(n, q)$ is higher than the $F(s, q)$ value of the current kth initial temporary result, and then propagated using the keystrokes that have been input since its hibernation. Otherwise, it is guaranteed that none of the active states propagated from the state can produce a top-k result for the query, and thus the hibernation remains.

6 Index size optimizations

Algorithms based on neighborhood generation, including our algorithms, often deliver large index size due to the enumeration of variants. In this section, we introduce three new techniques specific to our algorithms to remove redundancy in the index. We note that there are other physical compression methods such as double-array trie [2] that can be applied to our method only, because direct trie-based methods require traversing all child nodes in active node propagation, which is an expensive operation for a double-array trie.

6.1 Data string merge

The first technique to reduce index size can be regarded as a hybrid of neighborhood generation and edit distance computation. We show the basic idea with an illustrative example.

Example 11 Consider node 4 in Fig. 1. All of its descendant paths reach the same string s_1, meaning that these paths are variants of s_1 only. If node 4 is in an active state, we may directly compute the remaining part of the query string and s_1 for edit distance, and accumulate it to the current incoordination. Since the neighborhood generation is not needed here, only the path from node 4 that contains no deletions needs to be kept for edit distance computation.

The example suggests for any node n, we may disable neighborhood generation in the subtree rooted at n to save the space. When n appears in an active state, the searching algorithm switches to the edit distance computation mode for the following keystrokes of the query (invoked in Line 8 of Algorithm 1 or Line 10 of Algorithm 3). It computes the edit distance between the subsequent inputs and the remaining paths, as do the direct trie-based methods. The incoordination encountered before reaching n is added to the result of edit distance computation to obtain an overall incoordination. The correctness of the algorithm is proved as follows:

Proof Suppose a path x in the trie is divided into two parts x_1, on which active states are generated by IncNGTrie or IncNGTrie+, and x_2, on which edit distance is computed. To prove the correctness of the algorithm, we need to show that given a query q, there exists $\langle y, D_y \rangle \in V(q, \tau)$ such that $y = x$ and $|D_x \cup D_y| \leq \tau$, if and only if there exist substrings q_1 and q_2 such that (1) $q = q_1 \circ q_2$, and (2) there exists $\langle y_1, D_{y_1} \rangle \in V(q_1, \tau)$ such that $x_1 = y_1$ and $|D_{x_1} \cup D_{y_1}| + ed(x_2, q_2) \leq \tau$.

We first prove the sufficiency. Assume there exist substrings q_1 and q_2 that satisfy the above two conditions. Because $ed(x_2, q_2) \leq \tau - |D_{x_1} \cup D_{y_1}|$, by Lemma 1, $\exists \langle y_2, D_{y_2} \rangle \in V(q_2, \tau - |D_{x_1} \cup D_{y_1}|)$, such that $x_2 = y_2$ and $|D_{x_2} \cup D_{y_2}| \leq \tau - |D_{x_1} \cup D_{y_1}|$. By concatenating y_1 and y_2 as y, we have a $\langle y, D_y \rangle \in V(q, \tau)$ such that $x = y$ and $|D_x \cup D_y| \leq |D_{x_1} \cup D_{y_1}| + |D_{x_2} \cup D_{y_2}| \leq \tau$.

Then we prove the necessity. Assume there exists $\langle y, D_y \rangle \in V(q, \tau)$ such that $y = x$ and $|D_x \cup D_y| \leq \tau$. We divide D_x into two subsets D_{x_1}, which consists of the deletions occur before x_2, and D_{x_2}, which consists of the other deletions. Then we divide y into two substrings y_1, which is the same as x_1, and y_2, which is the remaining part. Because $y = x$ and $y_1 = x_1$, $y_x = x_2$. To divide D_y into corresponding deletion lists D_{y_1} and D_{y_2}, we do in the same way as we divide D_x. If there exist multiple $\langle y, D_y \rangle$'s, we pick the one which yields the smallest $|D_{x_1} \cup D_{y_1}|$. In doing so, we have q_1 and q_2 such that $q = q_1 \circ q_2$, $\langle y_1, D_{y_1} \rangle \in V(q_1, \tau)$ and $\langle y_2, D_{y_2} \rangle \in V(q_2, \tau)$. In addition, $D_{x_1} \cap D_{y_2} = \emptyset$ and $D_{x_2} \cap D_{y_1} = \emptyset$. Hence we have $|D_{x_1} \cup D_{y_1}| + |D_{x_2} \cup D_{y_2}| = |D_x \cup D_y|$. Because $|D_{x_2} \cup D_{y_2}| = |D_x \cup D_y| - |D_{x_1} \cup D_{y_1}| \leq \tau - |D_{x_1} \cup D_{y_1}|$, by Lemma 1, $ed(x_2, q_2) \leq \tau - |D_{x_1} \cup D_{y_1}|$. Therefore, we have q_1 and q_2 such that $q = q_1 \circ q_2$, $\langle y_1, D_{y_1} \rangle \in V(q_1, \tau)$, $\langle y_2, D_{y_2} \rangle \in V(q_2, \tau)$, $x_1 = y_1$, and $|D_{x_1} \cup D_{y_1}| + ed(x_2, q_2) \leq \tau$. □

We use this index reduction technique, called data string merge, in two ways. First, we merge the subtrees whose nodes reach the same single string ID. This is to guarantee that for each of these subtrees, there is only one path to compute edit distance after the merge. So our algorithms are still insensitive to the alphabet size. Second, since the number of variants of a data string is $|s|^{\tau}$, long strings may bring about considerable performance issues in the indexing construction. We truncate data strings at a length of l_p, and only use the truncated prefix to generate variants. Hence the number of variants of a data string is at most $|l_p|^{\tau}$. Edit distance computation is invoked for the remaining length when the query's and the data string's lengths exceed l_p.

6.2 Common subtree merge

The second technique is based on the observation that if the two subtrees rooted at two nodes are isomorphic to each other (we treat string IDs as labels), we can merge these two subtrees into one. An example is the subtrees rooted at node 12 and node 18 in Fig. 1, respectively. This is reminiscent of the minimization of automata [1], and the most efficient solution [19] is to traverse the trie while converting the subtree under each node into a hash code. Common subtrees are identified through hash table lookup and merged. The total time complexity is $O(|T|)$.

There is a subtle instance in merging common subtrees. Two common subtrees are literally identical single paths but (1) one is produced by common data string merge (e.g., the one formed by merging the paths under node 4 in Fig. 1), while (2) the other is not (e.g., the path under node 13 in Fig. 1). Our solution is not to distinguish the two types of subtrees but treat both of them as the first type. Then the two paths can be merged, and we switch to edit distance computation no matter what types they are. It can be shown that a second type subtree, being a single path, either contains only one node or there are τ #'s on the path from the root to the subtree. Thus, the efficiency of the algorithm will not be impaired in this case even though the active state propagation (Algorithm 2) is replaced by an edit distance computation.

Example 12 Figure 6 shows the trie in Fig. 1 after common data string merge and common subtree merge. For example, since all the paths from node 4 reach s_1, we merge them into a single path and mark node 4 as where we start edit distance computation. Node 7 is processed similarly. The subtrees under nodes 12 and 18 are identical, with incoming edges # and e, respectively. We pick either subtree to remove, say the one rooted at node 12, and then divert the incoming edge to node 18. The subtrees rooted at nodes 19 and 21 are removed likewise because they are identical to those rooted at nodes 4 and 7, respectively.

Due to the redundancy caused by neighborhood generation, which produces a number of similar strings, merging common subtrees may achieve remarkable reduction rate on index size. Apart from the index size reduction that can be applied to any tries, what is specific to our algorithms is that merging common subtrees facilitates the query processing performance because the number of active states can be reduced as well. We formally state the property that leads to the optimization on query processing:

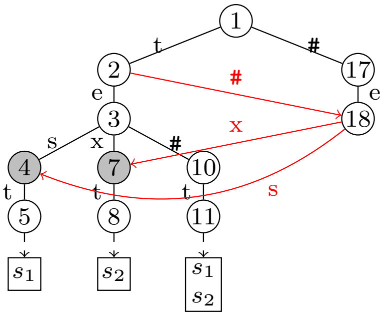


Fig. 6 Index with common data string merge and common subtree merge. Gray nodes indicate edit distance computation is invoked for descending paths. New edges formed by common subtree merge are colored in red

Lemma 4 *Consider two nodes n_1 and n_2 which share common subtrees rooted at them. For the* IncNGTrie *algorithm, given two active states in the same cursor $\langle n_1, u, \delta \rangle$ and $\langle n_2, u, \delta' \rangle$, if $\delta \leq \delta'$, $\langle n_2, u, \delta' \rangle$ can be discarded from the active state set. For the* IncNGTrie+ *algorithm, give two active states $\langle n_1, \delta, \omega \rangle$ and $\langle n_2, \delta', \omega' \rangle$, if $\langle n_1, \delta', \omega' \rangle$ is dominated by $\langle n_1, \delta, \omega \rangle$, $\langle n_2, \delta', \omega' \rangle$ can be discarded from the active state set.*

Proof As the subtrees rooted at n_1 and n_2 are the same, the two active states share the same string IDs as results. For the IncNGTrie algorithm, the results reachable from any future active states propagated from $\langle n_2, u, \delta' \rangle$ will be subsumed by those propagated from $\langle n_1, u, \delta \rangle$, because they read in the same keystroke and $\delta \leq \delta'$, which implies a more margin of incoordination. The correctness of the algorithm holds in spite of $\langle n_2, u, \delta' \rangle$ being discarded from the active state set. For the IncNGTrie+ algorithm, the results reachable from any future active states propagated from $\langle n_2, \delta', \omega' \rangle$ will be subsumed by those propagated from $\langle n_1, \delta, \omega \rangle$. Therefore, $\langle n_2, \delta', \omega' \rangle$ can be safely discarded. □

7 Discussions

7.1 Deletion of characters in query

As the number of active states is small (< 100 for IncNGTrie+, which will be reported in Sect. 8.2.2), we can cache the set of active states for every keystroke. When the user deletes a character, we roll back to the previous active state set. This is straightforward and only incurs tiny amount of memory consumption. For direct trie-based methods, caching active states for every keystroke consumes much more memory due to the large number of active nodes.

7.2 Updates in data strings

Updates may occur in data strings by inserting, deleting, or modifying a string. We discuss how to update the index when a data string is inserted or deleted. The case of modifying a string can be handled by first deleting it and then inserting a new one.

Insertion We use an auxiliary index to keep a trie built on the variants of new strings, with no index reduction technique applied on it. Whenever a data string is inserted, its deletion-marked τ-variant family is generated and inserted into the auxiliary index, and its ID is inserted into the string ID array of the auxiliary index. The auxiliary index is merged with the main index through an offline logarithmic merging [48], which is also adopted by many information retrieval solutions. We can also periodically reconstruct the index from scratch. Similar strategies have been adopted by most search engines to handle updates in their indexes.

Deletion If a data string in the main index is deleted, we do not modify the index but record its string ID in a table so that it will not be returned for future queries. If a data string in the auxiliary index is deleted, the variants of the string are removed from the trie, and its entries in the string ID array are removed as well. In case multiple strings can produce the same variants, we fetch the string IDs under the variant and see if there is only one result. If so, the variant can be safely deleted from the auxiliary index.

8 Experiments

We report experiment results and our analyses.

8.1 Experiment setup

Datasets and Queries We select the following publicly available datasets.

- **DBLP** is a dataset of bibliography records in computer science.[4]
- **UMBC** is a collection of English paragraphs with words processed from Stanford WebBase project.[5]
- **MEDLINE** is a set of journal citations and abstracts of biomedical literature.[6]
- **AOL** is a set of query logs from AOL.[7]

We tokenize the datasets into terms with white spaces and punctuation. Then each term is regarded as a data string. Statistics about the preprocessed datasets are provided in Table 3. The alphabet sizes are all 26. AOL is only used for the effectiveness of top-k queries. To study the effect of alphabet size, we also generate a synthetic dataset of 1 million strings with $|\Sigma| = 8$, 16, 32 and 64. 1000 strings are randomly sampled from each dataset as queries, and then 0 to 3 edit operations are applied to each query.

Evaluations and algorithms We summarize the experiments conducted and the methods compared:

- Overall (range) query processing performance. We compare the following algorithms: (1) **BEVA** [77] is a direct trie-based method that achieves minimum active node size by eliminating ancestor–descendant relationships among active nodes. It utilizes an automaton to speed up active node propagation. Since this is our previous work, we used the source code directly. (2) **META** [23] is a direct trie-based method which generates active nodes using only matching characters. We received the source code from the authors of this work. (3) **IncNGTrie** is our proposed algorithm that indexes in a trie the deletion-marked variants of data strings and incrementally computes active states during query processing. (4) **IncNGTrie+** is our proposed algorithm that improves IncNGTrie by reducing the number of active states.
- Searching phase performance. The aforementioned four algorithms are compared.
- Result-fetching phase performance. We compare the proposed duplicate removal technique with the deduplication method using only hash tables. BEVA and META are also compared.
- Scalability. We vary alphabet size and dataset size, and compare IncNGTrie(+) with BEVA and META.
- Top-k query processing performance. We first evaluate the effectiveness of the proposed ranking function. It is compared to static score ranking and edit distance ranking. Then we evaluate the optimization techniques proposed in Sect. 5. Finally, we compare the efficiency of our top-k query processing algorithm with META.
- Indexing performance. We evaluate the index size reduction techniques proposed in Sect. 6 and compare the index size and the construction time with BEVA and META. In addition, we study the effect of index size reduction on query processing speed.

For IncNGTrie and IncNGTrie+, we set $l_p = 12$ to generate deletion variants only for the length within this value. The exceeding part is processed by edit distance computation, as described in Sect. 6.1. Other existing methods for error-tolerant query autocompletion, such as [17] and [37], are not

[4] http://www.informatik.uni-trier.de/~ley/db/.

[5] http://ebiquity.umbc.edu/resource/html/id/351.

[6] http://mbr.nlm.nih.gov/Download/index.shtml.

[7] https://jeffhuang.com/search_query_logs.html.

Table 3 Dataset statistics

Dataset	$\|S\|$	avg. $\|s\|$
DBLP	319,690	8.6
UMBC	2,000,000	9.9
MEDLINE	1,782,517	10.0
AOL	365,274	9.4

compared because they have been shown to be outperformed by previous studies [23,43,70,77].

The keystroke saving of error-tolerant query autocompletion by edit distance has been demonstrated by [17]. We do not repeat the experiments here.

Measures We mainly measure the active node/state size and the query processing time, which consists of the searching time to maintain active states/nodes and the result-fetching time to get query results. All the measures are averaged over 1000 queries. For the evaluation of ranking methods, we measure mean reciprocal rank and success rate, which are common measures for query autocompletion [4,57,58,69].

Environments All the experiments were carried out on a on a PC with an AMD Opteron 2.4GHz Processor and 96GB RAM, running Ubuntu 16.04. All the algorithms were implemented in C++ in a main memory fashion.

8.2 Query processing performance

8.2.1 Overall query response time

We plot the average query response times with varying τ in Fig. 7a–c for a query length of 4 and in Fig. 7d–f for a query length of 7. The reason for choosing 4 and 7 is that they are representative lengths of short and long queries, respectively. Since the maximum edit distance threshold is 3 in our experiments, for any query length no greater than 3, all the strings in the dataset are results. Hence we choose 4, a length at which the results become meaningful, for short queries. A query length of 7 is where the active state numbers of direct trie-based methods drop to the same level as the proposed neighborhood generation methods (as we will see later). Hence we choose 7 for long queries.

The response time is not accumulated for previous characters but measured only when the 4-th or the 7-th character is typed. It is decomposed into searching time (top) and result-fetching time (bottom). BE, ME, NG, and N+ denote BEVA, META, IncNGTrie, and IncNGTrie+, respectively. We observe that IncNGTrie+ has the best response time among the four in most settings, followed by IncNGTrie, except BEVA is the fastest (due to its fast result fetching) in a few settings when the query length is 4. The advantage of IncNGTrie+ is more substantial when τ or the query length is large. IncNGTrie+ achieves up to 8 times speedup over IncNGTrie, and more than 100 times over BEVA and META. The main reason for the much longer response time of direct trie-based methods is the lengthy searching time. In the following, we analyze searching and result-fetching times of the algorithms separately. In the interest of space, we only show the results when $\tau = 3$. The results of other τ settings are similar.

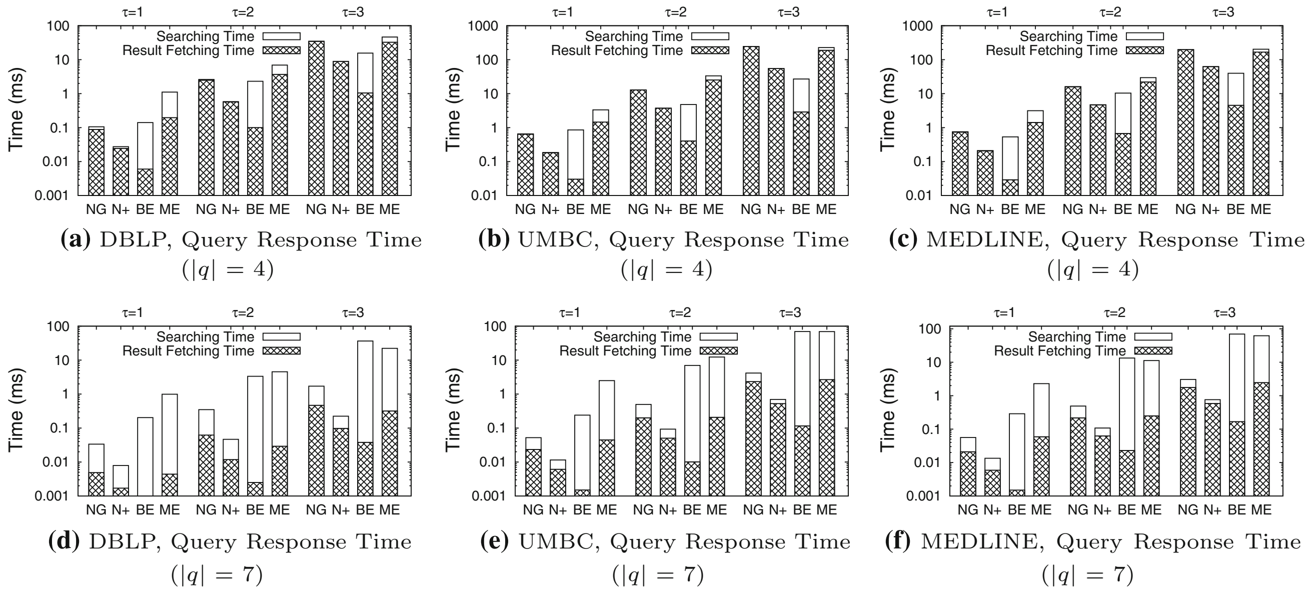
8.2.2 Searching time

We measure the active state numbers of the four algorithms on the three datasets when the ith character of the query string is input to the system, in line with [43]), and show the results in Fig. 8a–c. We observe that:

- The active state numbers of direct trie-based algorithms first increase with the query length, peak when the query length is 4 or 5, and then decrease.
- The active state numbers of neighborhood generation-based algorithms increase with the query length, but in a much more gentle manner.
- Although direct trie-based algorithm have smaller active state numbers when the query length exceeds 10 (because the query becomes very selective), their maximum active state numbers en route are still larger than those of neighborhood generation-based algorithms. For example, the numbers on MEDLINE are close to 10k for BEVA and META, but 3.6k for IncNGTrie and only 31 for IncNGTrie+.

We also observe the early stage explosion of direct trie-based methods. BEVA does not compute active states when $|q| \leq \tau$. However, when $|q|$ exceeds τ, its active state number exhibits a sudden explosion. Although META keeps only active nodes ending with matching characters, the number of active nodes is still huge, especially for queries with repeated characters.

Figure 8d–f shows the searching times of the three algorithms, accumulated from the first input character to the given query length. IncNGTrie+ is the fastest, followed by IncNGTrie. The performances of BEVA and META are close. In [23], IncNGTrie was shown to be slower than META on a small dataset (146k data strings). We also observe this result on DBLP, where META turns out to be faster than IncNGTrie for long queries. Nonetheless, IncNGTrie performs better than META on the two larger datasets used in our experiments, and IncNGTrie+ is even faster. Comparing META nd IncNGTrie+, the fastest direct trie-based and neighborhood generation-based methods, respectively, the latter is up to an order of magnitude faster on DBLP, and two orders of magnitude faster on the other two datasets. The speedup is more remarkable for short queries. This is more important for the applications featuring autocompletion.



(a) DBLP, Query Response Time ($|q| = 4$)

(b) UMBC, Query Response Time ($|q| = 4$)

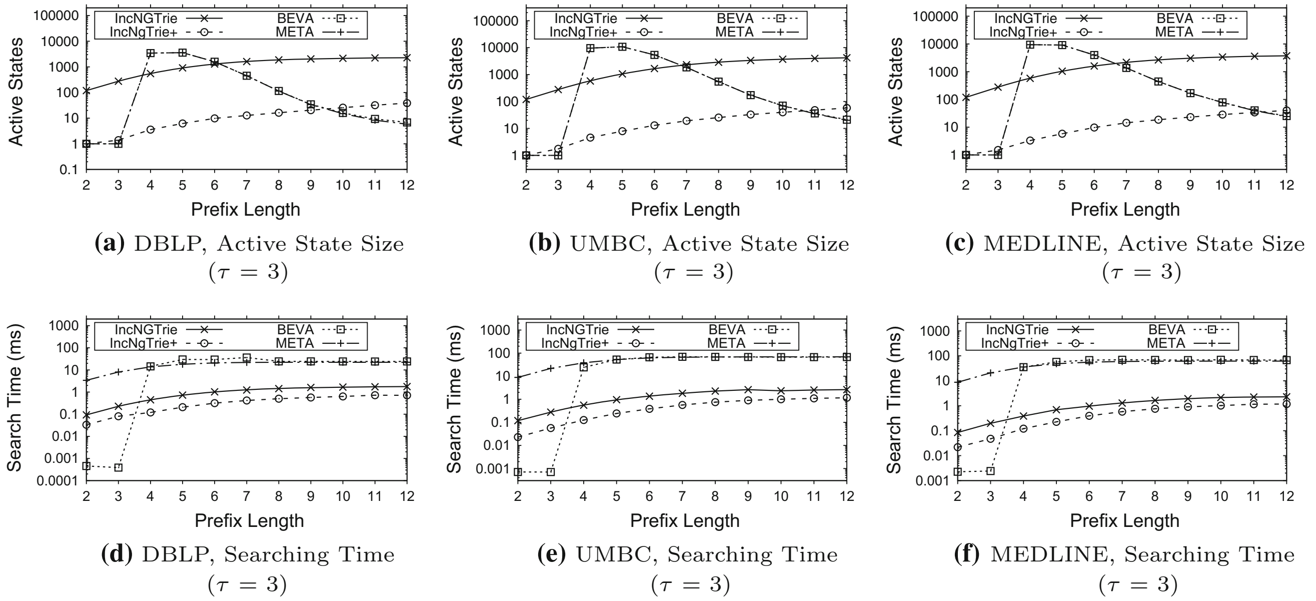
(c) MEDLINE, Query Response Time ($|q| = 4$)

(d) DBLP, Query Response Time ($|q| = 7$)

(e) UMBC, Query Response Time ($|q| = 7$)

(f) MEDLINE, Query Response Time ($|q| = 7$)

Fig. 7 Overall query processing performance



(a) DBLP, Active State Size ($\tau = 3$)

(b) UMBC, Active State Size ($\tau = 3$)

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

(d) DBLP, Searching Time ($\tau = 3$)

(e) UMBC, Searching Time ($\tau = 3$)

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

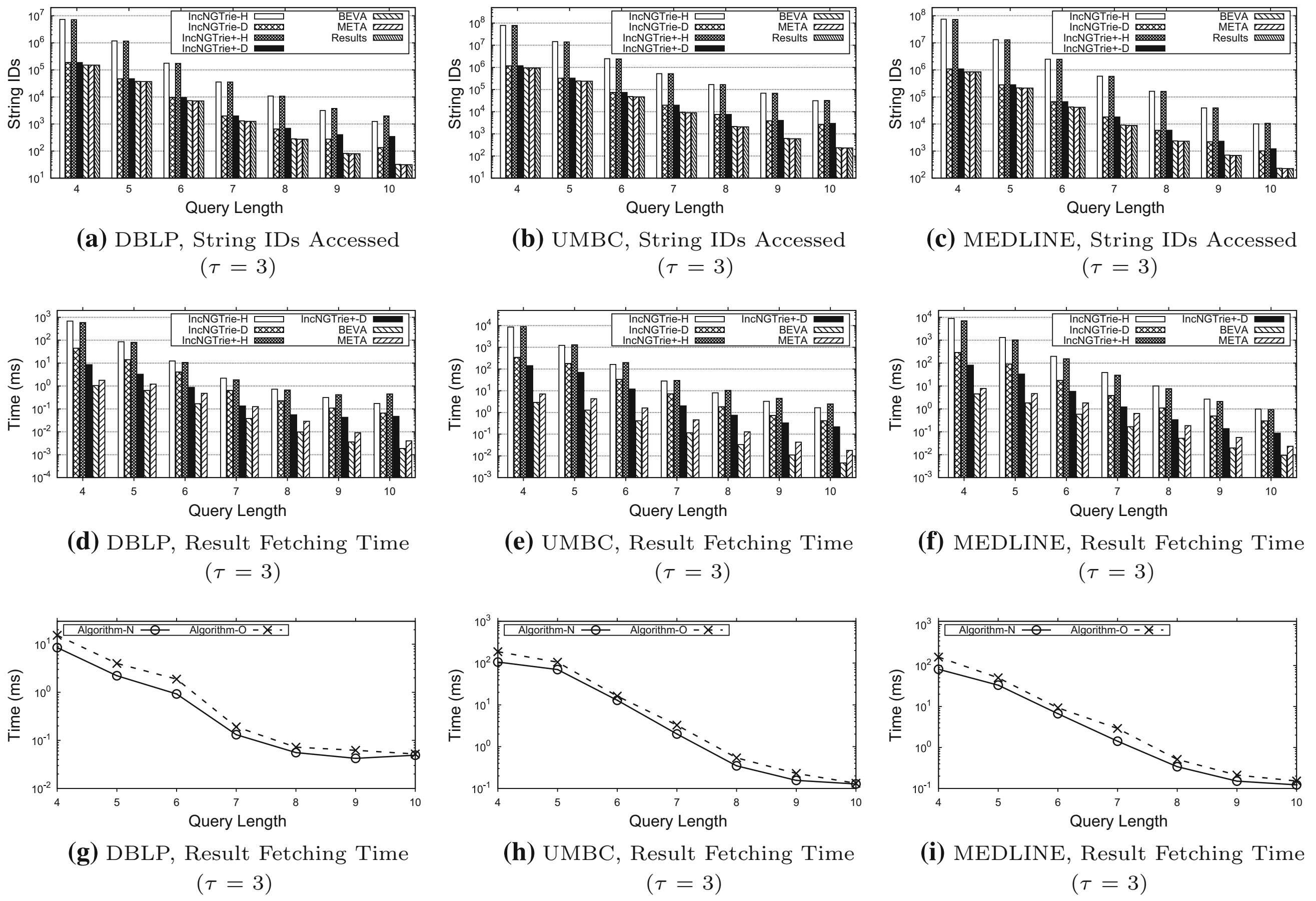
Fig. 8 Searching performance

8.2.3 Result-fetching time

We then evaluate the result-fetching performance. We consider the following two strategies of IncNGTrie and IncNGTrie+ to remove duplicates when fetching results:

- HashTable (denoted by Algorithm-H in figures). The algorithm uses a hash table to remove duplicates and return distinct strings reachable from active states.
- Deduplication (denoted by Algorithm-D in figures). The algorithm employs the duplicate removal techniques proposed in Sect. 4.

Figure 9a–c shows comparison of the string IDs accessed for respective query length when $\tau = 3$. We also show the results of BEVA and META as well as the number of distinct result strings (denoted by Results). As can be seen, HashTable strategy accesses a number of string IDs up to tens of million. When we use Deduplication strategy, the number of fetched string IDs can be reduced by almost two orders of magnitude, and is very close to direct trie-based methods for short queries. Figure 9d–f shows the corresponding result fetching times. By applying Deduplication, the result



(a) DBLP, String IDs Accessed ($\tau = 3$)

(b) UMBC, String IDs Accessed ($\tau = 3$)

(c) MEDLINE, String IDs Accessed ($\tau = 3$)

(d) DBLP, Result Fetching Time ($\tau = 3$)

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

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

(g) DBLP, Result Fetching Time ($\tau = 3$)

(h) UMBC, Result Fetching Time ($\tau = 3$)

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

Fig. 9 Result-fetching performance

fetching times of IncNGTrie and IncNGTrie+ is reduced from HashTable by up to almost two orders of magnitude.

Compared to the conference version of this work [70], we propose a new duplicate removal technique for Case 1 duplicates. To show their difference, we plot the result-fetching times of the IncNGTrie+ algorithm in Figure 9g–i. Algorithm-N denotes the algorithm equipped with the new technique. Algorithm-O denotes the algorithm equipped with the technique in [70]. The result-fetching times are close for the two algorithms, but the new technique is always slightly faster. The reason is that Algorithm-O accesses the entries in the string ID array using links, while Algorithm-N retrieves string IDs stored in continuous space.

8.3 Scalability

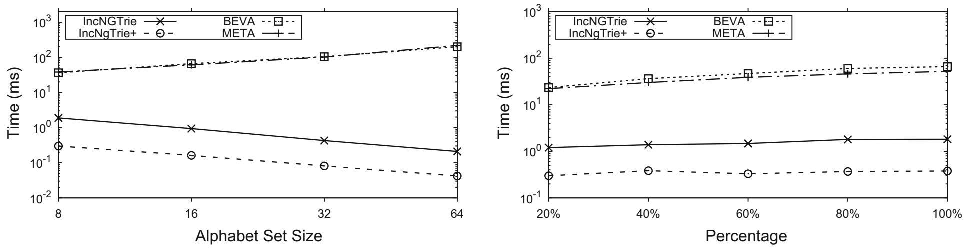
8.3.1 Varying alphabet size

We study the searching times of the algorithms using the synthetic dataset with alphabet size varying from 8 to 64. Figure 10a shows the accumulated searching times when the query length reaches 8 at a τ of 3. The searching times of direct trie-based methods rapidly grow with the alphabet size. On the contrary, the searching times of IncNGTrie and IncNGTrie+ decrease when we move alphabet size toward larger values, because their active state numbers are insensitive to the alphabet size, and the query becomes more selective for larger alphabets. When the alphabet has 8 characters, IncNGTrie+ is 89 times faster than BEVA and META. When there are 64 characters, the gap enlarges to 989 times. Note that in real applications, some alphabets can be very large, e.g., Unicode.

8.3.2 Varying dataset size

We study the scalability of the algorithms by varying dataset size. 20% to 100% data strings were randomly sampled from MEDLINE. Figure 10b shows the ratio of the searching times on the sample to that on the 20% dataset. We set query length to 8 and τ to 3. The general trend is that the searching times of the algorithms all grow with larger dataset size. IncNGTrie and IncNGTrie+ exhibit slower growth rates than direct trie-based methods. When the data set size jumps from 20% to

Fig. 10 Scalability



(a) Synthetic, Varying Alphabet Size

(b) MEDLINE, Varying Dataset Size

Table 4 Mean reciprocal rank (in percentage), AOL

k	$\|q\| = 4$			$\|q\| = 5$			$\|q\| = 6$			$\|q\| = 7$			$\|q\| = 8$		
	SSxED	SS	ED	SSxED	SS	ED	SSxED	SS	ED	SSxED	SS	ED	SSxED	SS	ED
10	**7.6**	0.6	5.4	**14.4**	2.8	11.5	**24.3**	9.5	20.8	**28.9**	16.5	23.1	**30.8**	19.2	25.9
50	**8.8**	0.8	5.8	**16.4**	2.9	12.5	**26.6**	10.3	22.8	**30.9**	17.3	25.9	**32.1**	20.0	26.9

Table 5 Success rate (in percentage), AOL

k	$\|q\| = 4$			$\|q\| = 5$			$\|q\| = 6$			$\|q\| = 7$			$\|q\| = 8$		
	SSxED	SS	ED	SSxED	SS	ED	SSxED	SS	ED	SSxED	SS	ED	SSxED	SS	ED
10	**16.7**	3.7	12.3	**25.4**	11.2	23.1	**42.2**	30.0	39.1	**55.8**	47.4	48.6	**63.2**	55.1	56.8
50	**19.4**	4.6	13.8	**27.3**	13.2	24.7	**45.2**	34.8	42.3	**59.4**	49.9	50.9	**70.1**	58.4	58.2

100%, the searching time increases by 4.1 times for BEVA, 4.2 times for META, but only 1.5 times for IncNGTrie and 1.2 times for IncNGTrie+. This showcases the less sensitiveness of our algorithms to dataset size than direct trie-based algorithms.

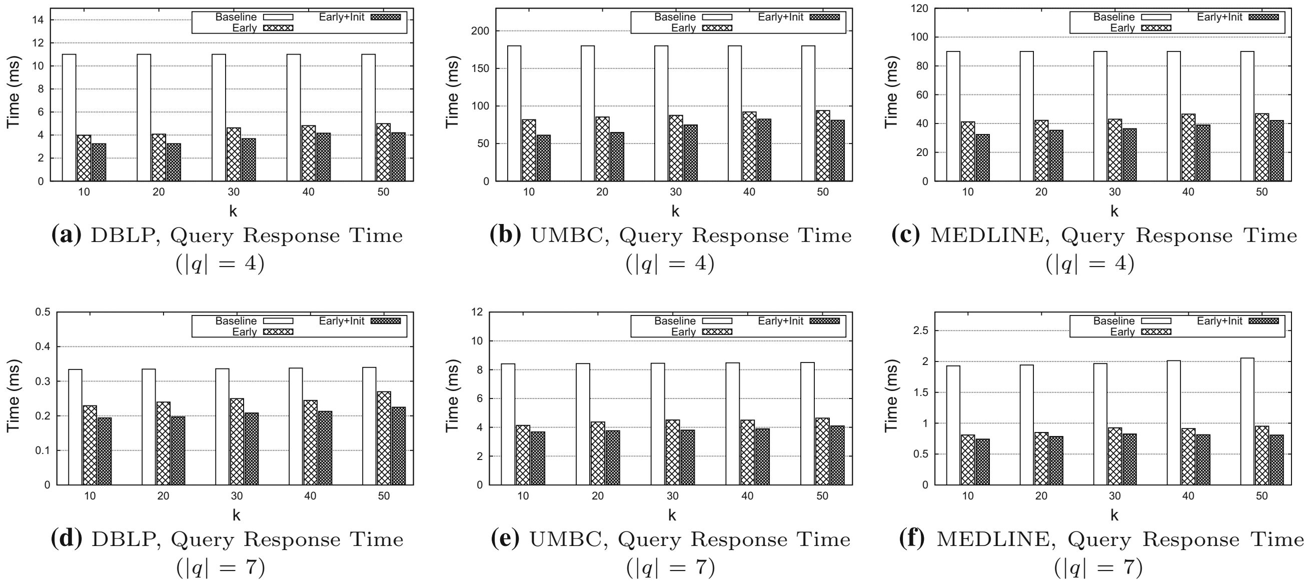
8.4 Top-*k* query processing

8.4.1 Ranking function

We first evaluate the result quality using the proposed ranking function (Eq. 1, denoted by SSxED). Two alternative ranking functions are compared: (1) descending order of static score and allowing a threshold of 3 edit operations (denoted by SS), and (2) ascending order of edit distance (denoted by ED). We use the AOL dataset because it has a query log. To balance the frequency and the edit distance, the static score is given by the logarithm to base 10 of frequency in the query log. Tables 4 and 5 show the mean reciprocal ranks (MRR) and the success rates (SR) of the three ranking functions when $k = 10$ and 50, respectively. MRR is defined as the average reciprocal of the intended string's ranking in the top-k suggestions (counted as 0 if not appearing). SR is defined as the percentage of queries such that the intended string appears in the top-k suggestions. We show the results for query lengths between 4 and 8, which we believe are suitable for query autocompletion applications. The results for queries shorter than 4 characters are less meaningful as the qualities are low for all the ranking methods compared. The best values are marked in boldface. The performance of SS is the worst among the three ranking methods, because it only considers the static score and ignores to what extent data strings are close to the query, though a threshold of 3 edit operations is allowed. ED ranks the results by the closeness to the query and delivers good performance, but it does not consider the popularity of terms in the query. By balancing between popularity and closeness, SSxED is the best in terms of both MRR and SR. Its advantage over ED is around 2% to 5% of MRR and 2% to 12% of SR. Note that even a small absolute difference in MRR could lead to considerable performance gain [57,69]. Another observation is that the margin increases when more characters are input to the query, showcasing the more importance of popularity when the query becomes more predictable.

8.4.2 Effect of optimizations

We evaluate the optimization techniques proposed in Sect. 5. We use the proposed ranking function SSxED and the first three datasets. The static scores are given by the frequency in the dataset. Figure 11a–f shows the average query response times by varying k under query lengths of 4 and 7 (same as Fig. 7, not accumulated time but measured only when the fourth or seventh character is typed). The baseline (denoted



(a) DBLP, Query Response Time ($|q| = 4$)

(b) UMBC, Query Response Time ($|q| = 4$)

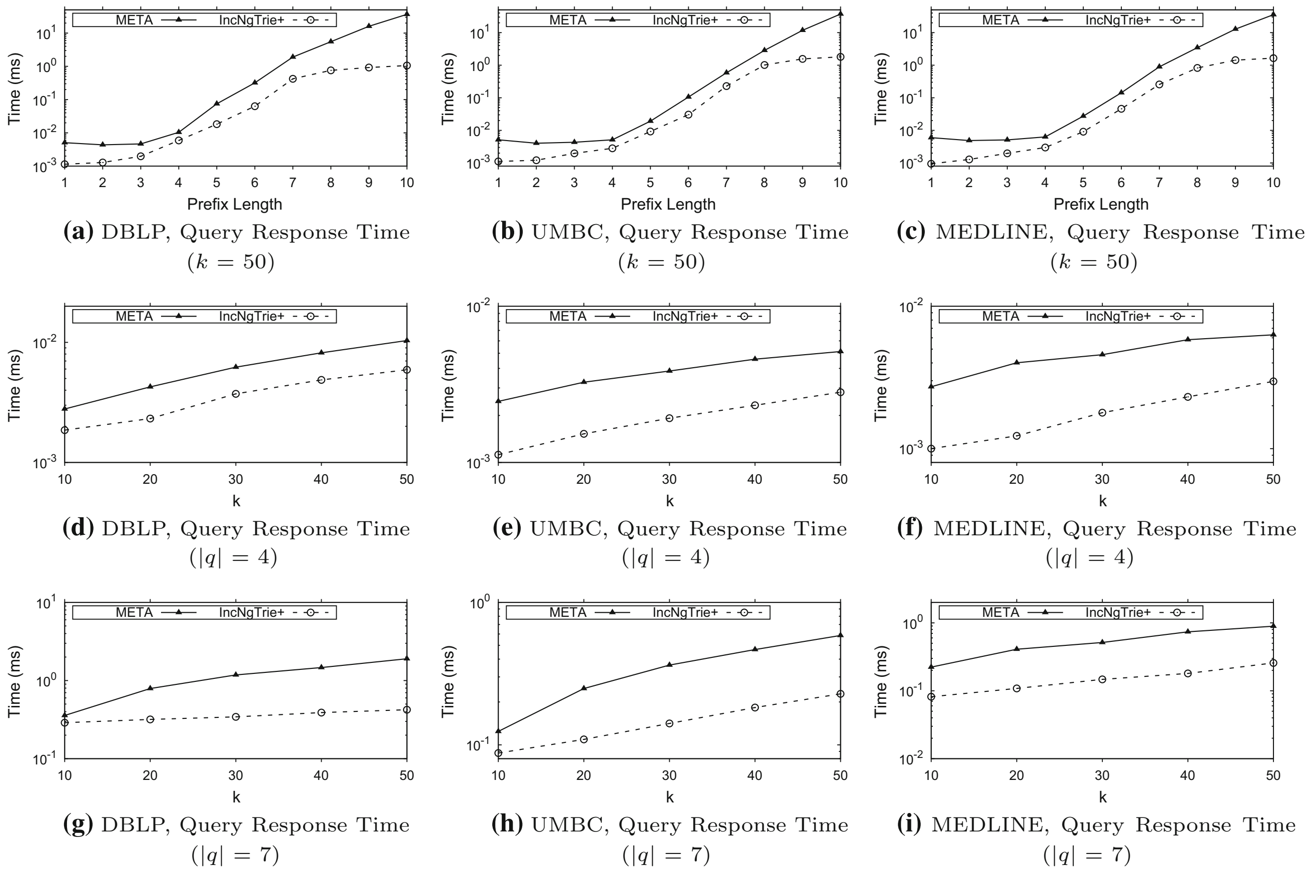
(c) MEDLINE, Query Response Time ($|q| = 4$)

(d) DBLP, Query Response Time ($|q| = 7$)

(e) UMBC, Query Response Time ($|q| = 7$)

(f) MEDLINE, Query Response Time ($|q| = 7$)

Fig. 11 Effect of optimizations for top-*k* query processing



(a) DBLP, Query Response Time ($k = 50$)

(b) UMBC, Query Response Time ($k = 50$)

(c) MEDLINE, Query Response Time ($k = 50$)

(d) DBLP, Query Response Time ($|q| = 4$)

(e) UMBC, Query Response Time ($|q| = 4$)

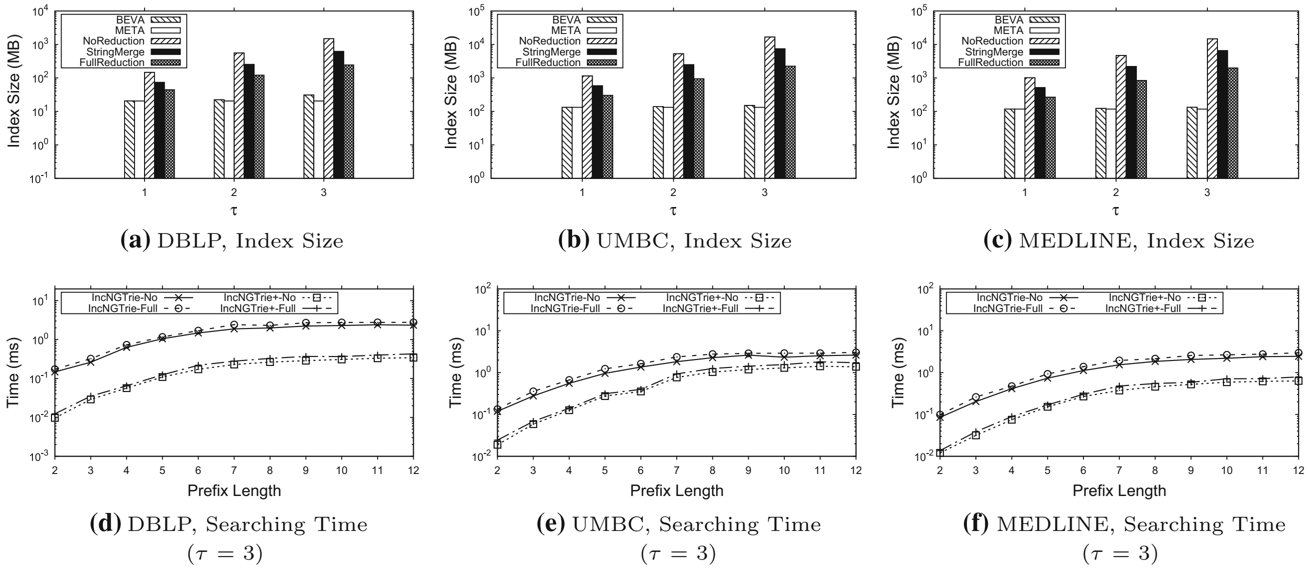
(f) MEDLINE, Query Response Time ($|q| = 4$)

(g) DBLP, Query Response Time ($|q| = 7$)

(h) UMBC, Query Response Time ($|q| = 7$)

(i) MEDLINE, Query Response Time ($|q| = 7$)

Fig. 12 Top-*k* query processing speed (ranking by edit distance only)



(a) DBLP, Index Size

(b) UMBC, Index Size

(c) MEDLINE, Index Size

(d) DBLP, Searching Time ($\tau = 3$)

(e) UMBC, Searching Time ($\tau = 3$)

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

Fig. 13 Indexing performance

by Baseline) is the IncNGTrie+ algorithm without any optimization. The algorithm that utilizes early termination is denoted by Early. We apply the proposed initialization technique on top of Early and denote the resultant algorithm by Early + Init. Since Baseline has to retrieve all the underlying strings of the active nodes, its running time is insensitive to k. For Early and Early + Init, as we have more temporal results to keep for a larger k, a moderate increase in query response time is witnessed. Comparing the two query lengths (4 and 7), we spend more query response time when the query length is 4. This is because a query of length 4 is on average less selective than a query of length 7, and thus we have more underlying strings to rank for the former case (cf. Fig. 9a–c). Early termination and good initialization are both useful in improving query processing speed. They are more effective when the query length is small, and early termination is more effective. With the two optimizations, the query processing speed is improved by up to 3.2 times when the query length is 4 and 2.5 times when the query length is 7.

8.4.3 Comparison with alternative method

We compare the top-k query processing performance of IncNGTrie+ with META under the ranking function of ascending order of edit distance (ED), because it is the only ranking function supported by META when additional reranking methods are not applied. Figure 12a–c shows the average query response time when $k = 50$, varying query length. An increasing trend is observed for both META and IncNGTrie+, since the query processing time is accumulated when more characters are input into the query. Thanks to neighborhood generation, IncNGTrie+ is always faster than META. The speedup varies from 2 to 35 times on the three datasets. For the comparison by varying k, we plot the response time when the query length is 4 in Fig. 12d–f, and the response time when the query length is 7 in Fig. 12g–i. The response times of the two algorithms are both moderately increasing with k. This is expected, as both need to retrieve more results. The gap between the two algorithms is consistent. No obvious difference is observed between the performances for the two query lengths. IncNGTrie+ is always the faster algorithm.

8.5 Indexing

The techniques proposed in Sect. 6 for index size reduction are evaluated. We use the term NoReduction to denote the trie that indexes deletion neighborhood without any index size reduction. StringMerge denotes that the data string merge is applied. FullReduction denotes that the common subtree merge is further applied. Figure 13a–c shows the index sizes of the algorithms on the three datasets with varying τ. Both optimizations are effective at reducing index size, and the reduction is more substantial with an increasing threshold. For example, on MEDLINE and $\tau = 3$, StringMerge reduces index size by 2.2 times, and FullReduction further reduces it by 6.4 times. Our algorithms have larger index sizes than direct trie-based methods (BEVA and META), and the gap increases for larger τ. This is expected as the number of variants is exponential in τ. Nonetheless, the index size is reduced to several GB (for 1–2 millions of data strings) by the two optimizations. Note that the index size can be further reduced using common trie compression techniques such as radix tree and double-array trie. BEVA's index size is slightly

Table 6 Index construction time (s)

Dataset	τ	BEVA	META	NoReduct.	FullReduct.
DBLP	1	0.63	0.14	5.4	20.1
	2	0.63	0.14	30.2	89.1
	3	0.63	0.14	101.1	236.9
UMBC	1	3.97	0.92	24.7	102.1
	2	3.97	0.92	130.2	508.7
	3	3.97	0.92	337.5	1029.1
MEDLINE	1	3.48	0.86	29.1	112.1
	2	3.48	0.86	141.6	698.8
	3	3.48	0.86	399.1	1222.7

larger than META's, since BEVA maintains an automaton for fast active node propagation.

We study the effect of index reduction on query processing speed. Figure 13d–f shows the searching times of IncNGTrie and IncNGTrie+ before and after applying the index reduction techniques when $\tau = 3$. -Full and -No denote the algorithms with and without index reduction, respectively. With index reduction, the searching time exhibits only a slight increase (up to 1.2 times). The increase results from the data string merge that replaces active state propagation with edit distance computation, which imposes more overhead.

We compare the index construction time and show the result in Table 6. Direct trie-based methods spend the least time on index construction as they simply build a trie from data strings. The index construction times of neighborhood generation-based method grow with τ due to more variants generated. With index reduction, the time spent on index construction increases, but considering this process is offline, the time is still affordable under the largest threshold setting.

9 Related work

Query autocompletion has been adopted in many applications such as Web search engines, command shells in operating systems, and text editors.

There has also been considerable interest in query autocompletion in research community. We refer readers to a recent survey for various kinds of query autocompletion [39]. The reactive keyboard [20] is a device that accelerates typewritten communication by predicting what the user is going to type. Grabski and Scheffer [30] studied the query prediction using index-based information retrieval techniques to complete a sentence given an initial fragment. Bast and Weber [5] proposed to use a succinct index to provide answers to word-level autocompletions. Nandi and Jagadish [53] studied the problem of autocompletion at the phrase level of multiple words. Hsu and Ottaviano [34] investigated in the direction of trie compression techniques to seek space efficiency. Fan et al. [27] and Bhatia et al. [6] proposed to handle query suggestion without query logs. Some recent studies [10,13,58,66,69] focused on find time-sensitive results by taking account of query popularity varying over time. There are also a few studies based on user behavior, e.g., returning context-aware [4,33,38,45–47,49,73] or personalized [11,12,57] results. Recently, with the rapid growth of the popularity of mobile devices, location-aware query autocompletion [35,36,55,75,76] received much attention.

Query autocompletion with edit distance constraints to tolerate errors were first studied in [37] and [17]. Li et al. [43] improved the method proposed in [37] to reduce memory consumption and query response time, both included in the TASTIER project [42] targeting type-ahead search. Zhou et al. [77] proposed the notion of edit vector to achieve the minimum active node size under reasonable constraints. Deng et al. [23] also proposed to reduce active node size by considering only matching characters and using a compact index. All of them are direct trie-based methods. It is noteworthy to mention the difference between this paper and our previous work [77]: (1) In [77], we directly index data strings in a trie, while data strings' deletion neighborhoods are indexed in this paper. (2) The method in [77] achieves minimum active node size by eliminating ancestor–descendant relationship among active nodes, while we reduce active state size using neighborhood generation on both query and data strings. (3) There is no need for duplicate removal for [77] because it guarantees ancestor–descendant relationship is eliminated for active nodes, while we develop corresponding techniques to remove duplicates that result from neighborhood generation in this paper. Algorithms were also developed to support top-k queries for fuzzy type-ahead search [16,23,44]. Cetindil et al. [16] proposed a ranking method for error-tolerant autocompletion using proximity information. Apart from edit distance, Markov n-gram transformation model [24] is adopted for error tolerance in the autocompletion task.

Another line of work aims at query recommendations, making reformulations to queries to assist users. The proposed solutions are mainly based on query clustering [3,56,68], session analysis [15,32], search behavior models [61], and Markov models [14,32,59].

Edit distance is a common distance function for approximate string matching. We refer readers to [9] for related work. Neighborhood generation is a category amid the various approaches. It computes a set of strings obtainable from the query or data strings by at most τ edit operations. The size of the neighborhood is $O(|s|^\tau \Sigma^\tau)$ for the full neighborhood method [52]. To reduce its size, deletion neighborhood was proposed for $\tau = 1$ [50] and extended to the general case [7]. k-errata trie [18] was proposed to process dictionary matching with edit distance constraints and was later improved [60]. Unlike our algorithm, the method in [60] generates full neighborhood, not only deletion but also insertion and substitution, though a character inserted or substituted is represented in a wildcard like our #. Subtrees are also merged in [60]. However, as its main purpose of merging subtrees is to speed up query processing, searching the merged tree is equivalent to searching multiple individual trees. In our work, only identical subtrees are merged as our main purpose is to reduce space usage. Edit distance is also adopted in similarity search and join [21,28,31,40,41,54,64,67,71,72] and approximate membership checking [22,65].

10 Conclusion

We investigate new solutions to error-tolerant query autocompletion using edit distance as constraints. Unlike existing approaches that directly index data strings in a trie, we devise a method to organize the trie index on the basis of deletion neighborhood of data strings. The new method achieves a very small and alphabet-insensitive active state size to speed up both thresholded and top-*k* query processing. Additional optimization techniques are developed to remove duplicates in query results and reduce index size. Extensive experimental evaluation over large-scale real datasets demonstrates that the proposed method significantly outperforms existing solutions in query response time.

Acknowledgements Chuan Xiao was supported by JSPS Kakenhi 16H01722, 17H06099, 18H04093, and NSFC 61702409. Sheng Hu and Yoshiharu Ishikawa were supported by JSPS Kakenhi 16H01722. Jie Zhang was supported by NSFC 61702409. Wei Wang was supported by ARC DPs 170103710 and 180103411, and D2DCRC DC25002 and DC25003. We thank the authors of [23] for kindly providing their source codes.

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley, Boston (1974)
2. Aoe, J.-I.: An efficient digital search algorithm by using a double-array structure. IEEE Trans. Softw. Eng. **15**(9), 1066–1077 (1989)
3. Baeza-Yates, R.A., Hurtado, C.A., Mendoza, M.: Improving search engines by query clustering. JASIST **58**(12), 1793–1804 (2007)
4. Bar-Yossef, Z., Kraus, N.: Context-sensitive query autocompletion. In: WWW, pp. 107–116 (2011)
5. Bast, H., Weber, I.: Type less, find more: fast autocompletion search with a succinct index. In: SIGIR, pp. 364–371 (2006)
6. Bhatia, S., Majumdar, D., Mitra, P.: Query suggestions in the absence of query logs. In: SIGIR, pp. 795–804 (2011)
7. Bocek, T., Hunt, E., Stiller, B.: Fast similarity search in large dictionaries. Technical Report ifi-2007.02. Department of Informatics, University of Zurich (2007)
8. Börzsönyi, S., Kossmann, D., Stocker, K.: The skyline operator. In: ICDE, pp. 421–430 (2001)
9. Boytsov, L.: Indexing methods for approximate dictionary searching: comparative analysis. ACM J. Exp. Algorithm. **16**(1), 1 (2011)
10. Cai, F., Chen, H.: Term-level semantic similarity helps time-aware term popularity based query completion. J. Intell. Fuzzy Syst. **32**(6), 3999–4008 (2017)
11. Cai, F., Chen, W., Ou, X.: Learning search popularity for personalized query completion in information retrieval. J. Intell. Fuzzy Syst. **33**(4), 2427–2435 (2017)
12. Cai, F., de Rijke, M.: Selectively personalizing query autocompletion. In: SIGIR, pp. 993–996 (2016)
13. Cai, F., Liang, S., de Rijke, M.: Prefix-adaptive and time-sensitive personalized query auto completion. IEEE Trans. Knowl. Data Eng. **28**(9), 2452–2466 (2016)
14. Cao, H., Jiang, D., Pei, J., Chen, E., Li, H.: Towards context-aware search by learning a very large variable length hidden Markov model from search logs. In: WWW, pp. 191–200 (2009)
15. Cao, H., Jiang, D., Pei, J., He, Q., Liao, Z., Chen, E., Li, H.: Context-aware query suggestion by mining click-through and session data. In: KDD, pp. 875–883 (2008)
16. Cetindil, I., Esmaelnezhad, J., Kim, T., Li, C.: Efficient instant-fuzzy search with proximity ranking. In: ICDE, pp. 328–339 (2014)
17. Chaudhuri, S., Kaushik, R.: Extending autocompletion to tolerate errors. In: SIGMOD, pp. 707–718 (2009)
18. Cole, R., Gottlieb, L.-A., Lewenstein, M.: Dictionary matching and indexing with errors and don’t cares. In: STOC, pp. 91–100 (2004)
19. Daciuk, J.: Comparison of construction algorithms for minimal, acyclic, deterministic, finite-state automata from sets of strings. In: CIAA, pp. 255–261 (2002)
20. Darragh, J.J., Witten, I.H., James, M.L.: The reactive keyboard: a predicive typing aid. IEEE Comput. **23**(11), 41–49 (1990)
21. Deng, D., Li, G., Feng. J.: A pivotal prefix based filtering algorithm for string similarity search. In: SIGMOD, pp. 673–684 (2014)
22. Deng, D., Li, G., Feng, J., Duan, Y., Gong, Z.: A unified framework for approximate dictionary-based entity extraction. VLDB J. **24**(1), 143–167 (2015)
23. Deng, D., Li, G., Wen, H., Jagadish, H.V., Feng, J.: META: an efficient matching-based method for error-tolerant autocompletion. PVLDB **9**(10), 828–839 (2016)
24. Duan, H., Hsu, B.-J.P.: Online spelling correction for query completion. In: WWW, pp. 117–126 (2011)
25. Duan, H., Li, Y., Zhai, C., Roth, D.: A discriminative model for query spelling correction with latent structural SVM. In: EMNLP-CoNLL, pp. 1511–1521 (2012)
26. Fagin, R., Lotem, A., Naor, M.: Optimal aggregation algorithms for middleware. In: PODS (2001)
27. Fan, J., Wu, H., Li, G., Zhou, L.: Suggesting topic-based query terms as you type. In: APWeb, pp. 61–67 (2010)
28. Feng, J., Wang, J., Li, G.: Trie-join: a trie-based method for efficient string similarity joins. VLDB J. **21**(4), 437–461 (2012)
29. Gao, J., Li, X., Micol, D., Quirk, C., Sun, X.: A large scale ranker-based system for search query spelling correction. In: COLING, pp. 358–366 (2010)
30. Grabski, K., Scheffer, T.: Sentence completion. In: SIGIR, pp. 433–439 (2004)
31. Gravano, L., Ipeirotis, P.G., Jagadish, H.V., Koudas, N., Muthukrishnan, S., Srivastava, D.: Approximate string joins in a database (almost) for free. In: VLDB, pp. 491–500 (2001)
32. He, Q., Jiang, D., Liao, Z., Hoi, S.C.H., Chang, K., Lim, E.-P., Li, H.: Web query recommendation via sequential query prediction. In: ICDE, pp. 1443–1454 (2009)
33. Hofmann, K., Mitra, B., Radlinski, F., Shokouhi, M.: An eye-tracking study of user interactions with query auto completion. In: CIKM, pp. 549–558 (2014)
34. Hsu, B.P., Ottaviano, G.: Space-efficient data structures for top-*k* completion. In: WWW, pp. 583–594 (2013)
35. Hu, S., Xiao, C., Ishikawa, Y.: An efficient algorithm for location-aware query autocompletion. IEICE Trans. **101–D**(1), 181–192 (2018)

36. Ji, S., Li, C.: Location-based instant search. In: SSDBM, pp. 17–36 (2011)
37. Ji, S., Li, G., Li, C., Feng, J.: Efficient interactive fuzzy keyword search. In: WWW, pp. 371–380 (2009)
38. Jiang, J., Ke, Y., Chien, P., Cheng, P.: Learning user reformulation behavior for query auto-completion. In: SIGIR, pp. 445–454 (2014)
39. Krishnan, U., Moffat, A., Zobel, J.: A taxonomy of query auto completion modes. In: ADCS, pp. 6:1–6:8 (2017)
40. Li, C., Wang, B., Yang, X.: VGRAM: improving performance of approximate queries on string collections using variable-length grams. In: VLDB, pp. 303–314 (2007)
41. Li, G., Deng, D., Feng, J.: A partition-based method for string similarity joins with edit-distance constraints. ACM Trans. Database Syst. **38**(2), 9:1–9:33 (2013)
42. Li, G., Ji, S., Li, C., Feng, J.: Efficient type-ahead search on relational data: a tastier approach. In: SIGMOD, pp. 695–706 (2009)
43. Li, G., Ji, S., Li, C., Feng, J.: Efficient fuzzy full-text type-ahead search. VLDB J. **20**(4), 617–640 (2011)
44. Li, G., Wang, J., Li, C., Feng, J.: Supporting efficient top-k queries in type-ahead search. In: SIGIR, pp. 355–364 (2012)
45. Li, L., Deng, H., Dong, A., Chang, Y., Baeza-Yates, R.A., Zha, H.: Exploring query auto-completion and click logs for contextual-aware web search and query suggestion. In: WWW, pp. 539–548 (2017)
46. Li, L., Deng, H., Dong, A., Chang, Y., Zha, H., Baeza-Yates, R.A.: Analyzing user's sequential behavior in query auto-completion via Markov processes. In: SIGIR, pp. 123–132 (2015)
47. Li, Y., Dong, A., Wang, H., Deng, H., Chang, Y., Zhai, C.: A two-dimensional click model for query auto-completion. In: SIGIR, pp. 455–464 (2014)
48. Manning, C.D., Raghavan, P., Schütze, H.: Introduction to Information Retrieval. Cambridge University Press, Cambridge (2008)
49. Mitra, B., Shokouhi, M., Radlinski, F., Hofmann, K.: On user interactions with query auto-completion. In: SIGIR, pp. 1055–1058 (2014)
50. Mor, M., Fraenkel, A.S.: A hash code method for detecting and correcting spelling errors. Commun. ACM **25**(12), 935–938 (1982)
51. Muthukrishnan, S.: Efficient algorithms for document retrieval problems. In: SODA, pp. 657–666 (2002)
52. Myers, E.W.: A sublinear algorithm for approximate keyword searching. Algorithmica **12**(4/5), 345–374 (1994)
53. Nandi, A., Jagadish, H.V.: Effective phrase prediction. In: VLDB, pp. 219–230 (2007)
54. Qin, J., Wang, W., Xiao, C., Lu, Y., Lin, X., Wang, H.: Asymmetric signature schemes for efficient exact edit similarity query processing. ACM Trans. Database Syst. **38**(3), 16 (2013)
55. Roy, S.B., Chakrabarti, K.: Location-aware type ahead search on spatial databases: semantics and efficiency. In: SIGMOD, pp. 361–372 (2011)
56. Sadikov, E., Madhavan, J., Wang, L., Halevy, A.Y.: Clustering query refinements by user intent. In: WWW, pp. 841–850 (2010)
57. Shokouhi, M.: Learning to personalize query auto-completion. In: SIGIR, pp. 103–112 (2013)
58. Shokouhi, M., Radinsky, K.: Time-sensitive query auto-completion. In: SIGIR, pp. 601–610 (2012)
59. Sordoni, A., Bengio, Y., Vahabi, H., Lioma, C., Simonsen, J.G., Nie, J.: A hierarchical recurrent encoder–decoder for generative context-aware query suggestion. In: CIKM, pp. 553–562 (2015)
60. Tsur, D.: Fast index for approximate string matching. J. Discrete Algorithms **8**(4), 339–345 (2010)
61. Tyler, S.K., Teevan, J.: Large scale query log analysis of re-finding. In: WSDM, pp. 191–200 (2010)
62. Ukkonen, E.: Algorithms for approximate string matching. Inf. Control **64**(1–3), 100–118 (1985)
63. Wagner, R.A., Fischer, M.J.: The string-to-string correction problem. J. ACM **21**(1), 168–173 (1974)
64. Wang, W., Qin, J., Xiao, C., Lin, X., Shen, H.T.: Vchunkjoin: an efficient algorithm for edit similarity joins. IEEE Trans. Knowl. Data Eng. **25**(8), 1916–1929 (2013)
65. Wang, W., Xiao, C., Lin, X., Zhang, C.: Efficient approximate entity extraction with edit constraints. In: SIMGOD, pp. 759–770 (2009)
66. Wang, Y., Ouyang, H., Deng, H., Chang, Y.: Learning online trends for interactive query auto-completion. IEEE Trans. Knowl. Data Eng. **29**(11), 2442–2454 (2017)
67. Wei, H., Yu, J.X., Lu, C.: String similarity search: a hash-based approach. IEEE Trans. Knowl. Data Eng. **30**(1), 170–184 (2018)
68. Wen, J., Zhang, H., Nie, J.: Query clustering using content words and user feedback. In: SIGIR, pp. 442–443 (2001)
69. Whiting, S., Jose, J.M.: Recent and robust query auto-completion. In: WWW, pp. 971–982 (2014)
70. Xiao, C., Qin, J., Wang, W., Ishikawa, Y., Tsuda, K., Sadakane, K.: Efficient error-tolerant query autocompletion. PVLDB **6**(6), 373–384 (2013)
71. Xiao, C., Wang, W., Lin, X.: Ed-Join: an efficient algorithm for similarity joins with edit distance constraints. PVLDB **1**(1), 933–944 (2008)
72. Yu, M., Wang, J., Li, G., Zhang, Y., Deng, D., Feng, J.: A unified framework for string similarity search with edit-distance constraint. VLDB J. **26**(2), 249–274 (2017)
73. Zhang, A., Goyal, A., Kong, W., Deng, H., Dong, A., Chang, Y., Gunter, C.A., Han, J.: adaqac: adaptive query auto-completion via implicit negative feedback. In: SIGIR, pp. 143–152 (2015)
74. Zhang, C., Naughton, J.F., DeWitt, D.J., Luo, Q., Lohman, G.M.: On supporting containment queries in relational database management systems. In: SIGMOD, pp. 425–436 (2001)
75. Zheng, Y., Bao, Z., Shou, L., Tung, A.K.H.: INSPIRE: a framework for incremental spatial prefix query relaxation. IEEE Trans. Knowl. Data Eng. **27**(7), 1949–1963 (2015)
76. Zhong, R., Fan, J., Li, G., Tan, K., Zhou, L.: Location-aware instant search. In: CIKM, pp. 385–394 (2012)
77. Zhou, X., Qin, J., Xiao, C., Wang, W., Lin, X., Ishikawa, Y.: BEVA: an efficient query processing algorithm for error-tolerant autocompletion. ACM Trans. Database Syst. **41**(1), 5:1–5:44 (2016)