Efficient Error-tolerant Query Autocompletion

Chuan Xiao
Nagoya University, Japan
chuanx@nagoya-u.jp

Jianbin Qin
UNSW, Australia
jqin@cse.unsw.edu.au

Wei Wang
UNSW, Australia
weiw@cse.unsw.edu.au

Yoshiharu Ishikawa
Nagoya University, Japan
y-ishikawa@nagoya-u.jp

Koji Tsuda
AIST and JST ERATO, Japan
koji.tsuda@aist.go.jp

Kunihiko Sadakane
NII, Japan
sada@nii.ac.jp

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 distance from the query are within the threshold. The major inherent problem is that the number of such prefixes is huge for the first few characters of the query and is exponential in the alphabet size. This results in slow query response even if the entire query approximately matches only few prefixes.

In this paper, we propose a novel neighborhood generation-based algorithm, IncNGTrie, which can achieve up to two orders of magnitude speedup over existing methods for the error-tolerant query autocompletion problem. Our proposed algorithm only maintains a small set of active nodes, thus saving both space and time to process the query. We also study efficient duplicate removal which is a core problem in fetching query answers. In addition, we propose optimization techniques to reduce our index size, as well as discussions on several extensions to our method. The efficiency of our method is demonstrated against existing methods through extensive experiments on real datasets.

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 user is typing a query. Other applications include command shells, desktop search, software development environments (IDE), 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, a recent trend of query autocompletion research is to tolerate errors when the user types in a query. Among the various approaches to deal with typographical errors, edit distance is a good measure for text documents, and therefore has been widely adopted and studied [8, 17, 20].

The existing 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 trie nodes and the current query, as each character of the query comes. Only trie nodes that satisfy the edit distance constraints are kept, and these nodes are called *active nodes*. Its performance has been shown to be superior to alternative paradigms (such as q-grams) [8]. Nevertheless, the efficiency of this approach critically depends on the number of active nodes. The number of active nodes is typically very large in practice (in the order of 10^5), 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 [20], or using buffered strategy and precomputation [8]), however, the query time is still long as there are still a huge number of active nodes to maintain when the first few characters of the query are typed in. For example, if we allow three edit errors, *all* the trie nodes on the highest four levels will be active nodes. The situation will be even worse for the applications where strings have large-sized alphabets; e.g., Unicode or CJK characters.

Previous approaches build a compact index (trie) and suffer from the inherent issue of having to maintain a huge set of active nodes in the query time. In this paper, we explore the other direction: *Can we drastically improve the query 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 τ-deletion marked variants (τ is the edit distance threshold), which are generated by deleting at most τ 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 10^2 – and insensitive to the alphabet size. To intuitive understand why this can

Proceedings of the VLDB Endowment, Vol. 6, No. 6

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#`[1] which has a distance of 1 to the query.

In addition, we propose efficient duplicate removal techniques when fetching query results – a problem existing in previous approaches yet not fully investigated. When not optimized, our index size is large due to the inclusion of deletion marked variants. Hence we introduce two 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 with previous methods.

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 conduct extensive experiments on several real datasets. The proposed method has been shown to outperform previous approaches by up to two orders of magnitude in terms of query response time.

The rest of the paper is organized as follows: Section 2 defines the problem definition and introduces preliminaries. Section 3 presents our neighborhood generation-based algorithm and its indexing and query processing technique. Section 4 elaborates how to fetch query results using our algorithm. The techniques to reduce index size are presented in Section 5. Section 6 discusses miscellaneous extensions such as processing data updates. Experimental results and analyses are covered by Section 7. Section 8 reviews related work. Section 9 concludes the paper.

2. PROBLEM DEFINITION AND 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 i-th character of s, starting from 1. $s[i\,..\,j]$ is the substring of s between position i and j. Given two strings s and s', "$s' \preceq s$" denotes that s' is a prefix of s; i.e., $s' = s[1\,..\,|s'|]$.

$ed(s,t)$ returns the edit 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 [33]. An efficient thresholded edit distance computation tests if $ed(s,t) \leq \tau$ in $O(\tau \cdot \min(|s|,|t|))$ time [32].

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

[1] "`#`" means the corresponding character is deleted.

2.2 Analysis of Previous Approaches

[8] and [17] independently developed solutions to processing error-tolerant query autocompletions with edit distance constraints. The techniques proposed in the two papers were similar. In the *indexing phase*, data strings are organized in a trie. In the *searching phase*, they maintain the set of all prefixes of the data strings that are within edit distance τ from the query string. The corresponding nodes in the trie are called *active* nodes or *valid* nodes. Whenever a character is appended to the query, the set of new active nodes is computed using current active ones. In the *result fetching* phase, the data strings stored under the leaf nodes that can be reached from active nodes are returned as the result. The time complexity of processing a query is $O(\tau \cdot (|A| + |A'|))$, where A and A' are the sizes of active nodes before and after inputting a key stroke, respectively. The space complexity is $O(|A| + |A'|)$. [20] further improved the method proposed in [17], 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 and the prefix. By considering only pivotal active nodes, it improves the time and the space complexity to $O(\tau \cdot (|P| + |P'|))$ and $O(|P| + |P'|)$, respectively, where P and P' are the sizes of pivotal active nodes before and after inputting a key stroke.

We call the above algorithms *direct trie-based approaches* as their indexing method is to construct a trie directly on data strings. The main drawback of the direct trie-based approaches is the large active node size. The problem is even serious for the first few key strokes of the query, since the set of active nodes includes the prefixes from an enormous subset of data strings. Even for pivotal active nodes, the size of P and P' can be up to $O((|Q|-1)^\tau |\Sigma|^\tau)$. For example, considering a query "`abc`" and $\tau = 1$, the pivotal active nodes include all the prefixes in the pattern of "`?bc`" or "`a?c`". We call this problem *early stage explosion*.

2.3 FastSS

A category of approaches to edit distance queries is the *neighborhood generation* based approaches [7], which generate a set of strings within a certain edit distance from a query. Among this category, the FastSS [6] algorithm utilizes deletion neighborhood [23] and achieves fast query performance 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. E.g., $\Delta(\texttt{brisbane}, 3) = \texttt{brsbane}$. 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 resulting string after k deletions, and call $[\,p_1,p_2,\ldots,p_k\,]$ the *deletion list* of the resulting string. E.g., $\Delta(\Delta(\texttt{brisbane}, 2), 2) = \texttt{bsbane}$, 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 resulting string after deleting s by k characters 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 $V(s,k)$.

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

LEMMA 1 (VARIANT MATCHING PRINCIPLE [6]). *Given two strings s and t, $ed(s,t) \le \tau$, 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 $|D_x \cup D_y| \le \tau$.*

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 the 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 $mul(e, D_y)$. E.g., $[1,1,2] \cup [1,2,2] = [1,1,2,2]$. We call the size of their deletion lists' union the *incoordination* of two variants.

EXAMPLE 1. *Consider two strings $s =$ `brisbane` and $t =$ `brosbne`, whose edit distance is 2. The two strings share a common variant "`brsbne`", and the corresponding deletion lists are $[3,5]$ and $[3]$, whose incoordination is 2.*

For error-tolerant query autocompletion tasks, the variant matching principle can be adapted to handle the case where 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 $ed(s',t) \le \tau$, 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 $|D_x \cup D_y| \le \tau$.*

PROOF. We first prove its necessity. Because $ed(s',t) \le \tau$, according to Lemma 1, there exists $\langle x', D'_x \rangle \in V(s',\tau)$ s.t. $x' = y$ and $|D'_x \cup D_y| \le \tau$. 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 $|D_x \cup D_y| \le \tau$.

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

3. NEIGHBORHOOD GENERATION-BASED ALGORITHMS

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

According to Lemma 2, one can design an algorithm to process the query when characters are incrementally appended. The query'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 problem, we resort to *deletion-marked variants*, namely, to use a "`#`" to denote a character deleted from a data string. E.g., "`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 if the contents match as well as if their incoordination exceeds the threshold at the same time. Now we design the 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 choose to index the deletion-marked variants in a trie. We name our algorithm IncNGTrie, standing for "Incremental algorithm based on Neighborhood Generation and a Trie index".

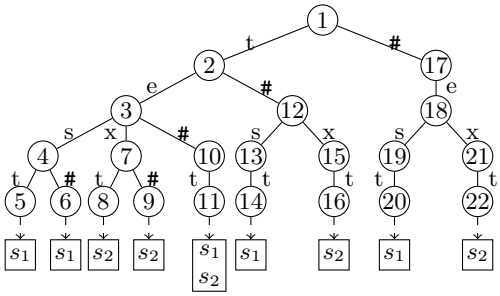


Figure 1: Example of IncNGTrie ($s_1 =$ `test`, $s_2 =$ `text`)

Table 1: Active States for $q =$ `tas`

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

EXAMPLE 2. *Consider $S = \{$ `test`, `text` $\}$, and $\tau = 1$. Figure 1 shows the trie constructed using the IncNGTrie algorithm. Each path in the trie 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 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 *cursor*, indicating it is expecting the u-th character of the query, and δ denotes the incoordination that has been encountered. An active states specifies that a variant of $q[1..u-1]$ matches a prefix of a data string's variant, represented by the path from the root to the node n, with an incoordination of δ. When the user types the u-th character of the query, 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, 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, and both d_q and d_s are in the range $[1, \tau - \delta]$.

Following the above active state propagating strategy, we design the searching phase of the IncNGTrie algorithm. The pseudocode 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 key stroke $q[v]$, the algorithm computes new active states with current ones. For the current active states with $u > v$, they will still be active (Line 6) as the u-th character has not come yet. For the other current active ones, we include their children through edge label $q[v]$ into the new active states. 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 stored on the leaf nodes reachable from the 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 Section 4.

EXAMPLE 3. *Consider a query $q =$ `tas`. Table 1 shows the active states for each key stroke using the IncNGTrie algorithm.*

3.1 Complexity Analysis

Now we analyze the worst-case time and space complexity of processing a query using the IncNGTrie algorithm. [2]

We divide active states into two groups: those with $u = |q| + 1$ and those with $u > |q| + 1$. For the first group, the search algorithm finds paths in the trie that match any variant of the query with an incoordination no more than τ. The runtime cost can be measured as the number of results joining the query's τ-variant family and the variant universe on the predicate that incoordination does not exceed τ. Hence the cost is $O(\tau^2 \cdot |q|^\tau)$. The space cost is the number of variants in the universe that match a query's variant with an incoordination no more than τ; i.e., $O(\tau \cdot (|q| + \tau)^\tau)$. For the second group, as they are inserted into new active state set without further actions, the runtime and the space costs equal the number of second group active states; i.e., $O((\tau - u + |q| + 1) \cdot (|q| + \tau - u + |q| + 1)^{\tau - u + |q| + 1})$ for each u. Combining the costs on the two groups, the time complexity is $O(\tau^2 \cdot |q|^\tau)$, and the space complexity is $O(\tau \cdot (|q| + \tau)^\tau)$.

Compared with direct trie-based methods, whose time and space complexity are $O(\tau \cdot |\Sigma|^\tau)$ and $O(|\Sigma|^\tau)$, respectively, the runtime and space costs of IncNGTrie are independent of the alphabet size, and therefore the algorithm 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 share the same variants, and we do not need to activate them respectively in consequence. For the example of a query "`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`".

4. FETCHING QUERY RESULTS

We return as results the strings stored on the leaf nodes reachable from the active states whose cursors are equal to $|q| + 1$. One may notice that duplicates may exist in the string ids on leaves due to neighborhood generation. It is noteworthy to mention that duplicate results also exist for direct trie-based methods; e.g., an active node is an ancestor of another and subsumes the string ids under the latter; but none of previous work discusses the removal of them. In this section, we investigate how to efficiently eliminate duplicates in the results of error-tolerant query autocompletion, and introduce the detailed result fetching phase of the IncNGTrie algorithm.

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

- **Case 1** The associated string ids reachable from a node contain duplicates. E.g., node 3 has five leaf descendants yet they report only two results s_1 and s_2. They are caused by neighborhood generation on the data strings. The path from a node to its descendants may differ due to the existence of deletions, but reach the same string eventually.
- **Case 2** The nodes of two active states are of ancestor-descendant relationship. E.g., node 3 is an ancestor of node 7, therefore subsuming the results from the latter. They are caused by neighborhood generation on the query. 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. E.g., node 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

Algorithm 1: IncrementalSearch (q, τ, T)

Input : q is a query string input character by character; τ is an edit distance threshold; T is a trie built on the deletion-marked τ-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 ← ExpandActive(r, 1, 0, τ) ;          /* r: the root of T */
2  for each key stroke q[v] do
3      A' ← ∅ ;                            /* new active states */
4      for each ⟨ n, u, δ ⟩ ∈ A do
5          if u > v then
6              A' ← A' ∪ { ⟨ n, u, δ ⟩ };
7          if n has a child n' through label q[v] then
8              A' ← A' ∪ ExpandActive(n', u + 1, δ, τ);
9      A ← A';
10 R ← ∅;
11 for each ⟨ n, u, δ ⟩ ∈ A such that u = v + 1 do
12     R ← R∪ strings resident on leaf nodes reachable from n;
13 return R
```

Algorithm 2: ExpandActive (n, u, δ, τ)

```
1  A ← ∅;
2  for d_q = 0 to τ − δ do          /* d_q: deletions in query */
3      A ← A ∪ { ⟨ n, u + d_q, δ + d_q ⟩ };
4  d_s ← 1 ;                        /* d_s: deletions in data string */
5  while δ + d_s ≤ τ and n has a child n' through label # do
6      for d_q = 0 to τ − δ do
7          δ' ← max(d_q, d_s);
8          A ← A ∪ { ⟨ n', u + d_q, δ + δ' ⟩ };
9      n ← n', d_s ← d_s + 1;
10 return A
```

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

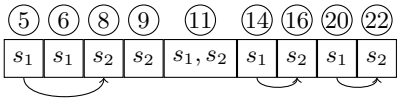


Figure 2: Example of String ID Array. Resident leaf nodes are listed above entries. An arrow represent the link to the next result. Entries without out-links shown are linked to null.

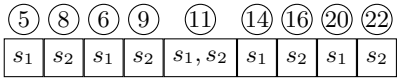


Figure 3: Example of Rearranged String ID Array. Resident leaf nodes are listed above entries.

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 resident on leaf nodes, and equip each node with two pointers marking the range in the array containing the string ids under its leaf descendants (we call *result fetching range* of a node). For example, consider the strings ids in Figure 1 as an array B. 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 [24] 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 that needs no additional space but still reports each distinct string in $O(1)$ time.

Case 1 duplicates are caused by the existence of deletions in the path from a node to its descendants. For a node and the strings under its leaf descendants, the problem becomes how to skip the string ids that follow paths containing a # from the reporting node. Now we ask in a reverse way: given a set of leaf nodes, for which node their associated string ids are the results? The following observation gives the answer.

Observation 1. *For a set of leaf nodes, if the path from their least common ancestor (LCA) to each of them contains no #, the associated string ids on these leaf nodes are distinct and include all the results for the LCA.*

Motivated by Observation 1, we link the entries in the string id array using their least common ancestor. For each entry $B[i]$ in the array, we use a pointer linking it to $B[j]$, such that j is the smallest value $j > i$ and the paths from $LCA(B[i], B[j])$ to $B[i]$ and $B[j]$ contain no #; or "null" if there is no such $B[j]$. Then the results for a node can be found as follows: From the node's result fetching range, we locate the first result, and then go to the next result through the link. The above process is repeated until we reach null or go beyond the result fetching range.

Example 4. *Figure 2 shows the example of string id array with links. Consider node 3, whose result fetching range in the string id array is [1, 5]. The first result is $B[1] = s_1$, the second result is $B[3] = s_2$, and then we stop as $B[3]$ links to null.*

Algorithm 3: CreateStringIDArray $(n, B, prev)$

```
1  if n is a leaf node then
2      B.push(n's associated string id );
3      L[prev] ← B.length ;          /* L[0] is a dummy link */
4      L[B.length] ← null ;           /* initialize a link */
5      return B.length
6  else
7      for each n's child n' do
8          if n' is through label # then
9              CreateStringIDArray(n', B, 0);
10         else
11             prev ← CreateStringIDArray(n', B, prev);
12     return prev
```

Algorithm 4: CreateRearrangedStringIDArray (T)

```
1  B ← ∅, B' ← ∅;
2  CreateStringIDArray (r, B, 0) ;      /* r: the root of T */
3  for i = 1 to |B| do
4      if B[i] has not been accessed then
5          j = i, B'.push(B[j]);
6          while L[j] ≠ null do    /* go through the chain */
7              j ← L[j], B'.push(B[j]);
8  return B'
```

The link creation algorithm is shown in Algorithm 3 (invoked by Line 2 in Algorithm 4). It builds a string id array B and creates the links L between entries in $O(|T|)$ time through a traversal of the trie. Each distinct string can be returned in $O(1)$ time for a given node. However, it still needs additional space for the links. The remedy is to rearrange the string id array in the order of the links. We start with the first entry and put all the entries reachable from it into a new array, and then continue with the next unprocessed entry, and so on. Lines 3 – 7 in Algorithm 4 show the rearrangement algorithm which passes all the chains in $O(|B|)$ time. After that we update the nodes in the trie for their result fetching ranges in the rearranged array.

Example 5. *Consider the string id array in Example 4. The resulting array after rearrangement is given in Figure 3. We update the nodes' result fetching ranges in the rearranged array. E.g., for node 3, within its current range [1, 5], $B[1]$ and $B[3]$ are the first and last results for node 3. Since $B[1]$ and $B[3]$ are moved to $B'[1]$ and $B'[2]$ in the new array, node 3's range in the rearranged array becomes [1, 2].*

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 $|q| + 1$, and none of its ancestors is in an active state with cursor equal to $|q| + 1$. In other words, reporting nodes are those whose leaf descendants 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 if 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 [35]) 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 with a hash table, and test if none of n's ancestors is in the table. Due to the edit distance constraints, only its ancestors on

level $|q| - \tau$ or below need to be tested, thus taking at most 2τ hash table lookup. 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 generated. N_r denotes the set of reporting nodes and its size is usually very small.

LEMMA 3. *If a node n is an ancestor of a node n', for a fixed cursor u, IncNGTrie always inserts $\langle n, u, \delta \rangle$ into active state set before $\langle n', u, \delta' \rangle$ for any incoordinations δ and δ'.* [3]

Assuming active states with a fixed cursor are accessed according to the order in which they are generated, Lemma 3 asserts that a node n's ancestor must appear before n in the order. 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. (See Figure 1 for an example). 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 pseudocode 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. [4] If n_i is within this range, n_j is an ancestor of n_i and hence 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, and 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 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 Lemma 3, 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 $n'_k - 1 \geq n_i > n_j$. Therefore $n'_j + 1 \leq n_j < n'_k - 1$, stating that n'_j is an ancestor of n_j. It contradicts that n_j is a reporting node and hence the correctness of the algorithm is proved. □

EXAMPLE 6. *Consider the trie in Figure 1 and a query* te. *The active states with cursors equal to $|q| + 1$ are $\langle 2, 3, 1 \rangle$, $\langle 12, 3, 1 \rangle$, $\langle 18, 3, 1 \rangle$, $\langle 3, 3, 0 \rangle$, and $\langle 10, 3, 1 \rangle$ in correct order. First, node 2 is a reporting node and inserted to the BST. For node 12, the search in BST returns 2 and it is an ancestor of 12. For node 18, since node 2 in the BST is not an ancestor of 18, 18 is a reporting node and inserted to BST. Both nodes 3 and 10 can find their ancestor 2 in the BST. Finally, node 2 and 18 are returned as reporting nodes.*

[3] In some subtle cases two active states may share the same node and cursor. The lemma holds for the first occurrence of the (node, cursor) pair.

[4] 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, and **false** otherwise.

```
1 n_j ← T_r.search(n_i) ;                          /* n_j ≤ n_i */
2 if n_i = n_j then
3     return false
4 n_k ← n_j's next sibling;
5 if n_j + 1 ≤ n_i and n_k − 1 ≥ n_i then
6     return false
7 else
8     T_r.insert(n_i);
9     return true
```

4.3 Eliminating Case 3 Duplicates

To handle Case 3 duplicates, we use a hash table to memorize 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. INDEX SIZE OPTIMIZATIONS

Algorithms based on neighborhood generation, including our IncNGTrie algorithm, often exhibit large index size due to the enumeration of variants. In this section, we introduce two new techniques specific to the IncNGTrie algorithm to remove redundancy in the index and hence resulting in substantial size reduction. 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 expansion, which is an expensive operation for a double-array trie. Such discussion is omitted here in the interest of space.

5.1 Common 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 7. *Consider node 4 in Figure 1. All its descendant paths reach the same string s_1, meaning that these paths are variants of the string s_1 only. If node 4 is in an active state, we may directly compute the remaining parts of the query and s_1 for edit distance, and add 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.*

For any node whose leaf descendants share the same single string id, we remove from the trie its descendant nodes on paths containing any #. When the node appears in an active state, the searching algorithm switches to edit distance computation mode for the following key strokes of the query. It computes the edit distance between the subsequent inputs and the single remaining path, as do the direct trie-based methods. But the difference is that we have only one path to compute edit distance, so our algorithm is still insensitive to the size of alphabet. The incoordination encountered before reaching this node is added to the result of edit distance computation to obtain an overall incoordination. Besides the trie, when a leaf node is removed, the corresponding entry in the string id array is also removed to save space. The

correctness of the algorithm holds. The reason is that there must be a # on the path reaching the removed leaf node, and thus the corresponding removed string id is a result only for the nodes after the #, which are also removed.

5.2 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 resident on leaf nodes as labels), we can merge these two subtrees into one. An example is the subtrees rooted at node 12 and node 18 in Figure 1. This is reminiscent of the minimization of automata [1], and the most efficient solution [10] 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 are two subtle instances in merging common subtrees. The first one is that 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 Figure 1), while (2) the other is not (e.g., the path under node 13 in Figure 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 a second type subtree in this scenario either contains only one node or has τ #'s before reaching the subtree, and thus the efficiency of the algorithm will not be impaired though its node expansion (Algorithm 2) is replaced by a more expensive edit distance computation.

The second subtlety lies in reducing the size of the string id array. Although nodes can be removed by common subtree merge, we cannot simply remove the underlying string ids from the string id array because they may be results for some nodes outside the subtree. Nevertheless, if the incoming edge to the root of the subtree is a #, the underlying entries in the string id array can be safely discarded, because we are sure they will not become the results of any nodes outside the subtree due to Observation 1. In order to achieve maximum size reduction on the string id array, for a series of identical subtrees, we reserve in the trie the first one in terms of pre-order traversal whose root's incoming edge is not a #. In case of all incoming edges being #, we keep the first one.

EXAMPLE 8. *Figure 4 shows the trie in Figure 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 node 12 and 18 are identical, with incoming edges # and e, respectively. Thus we remove the subtree rooted at node 12 and divert the incoming edge to node 18. The subtrees rooted at node 19 and 21 are removed likewise because they are identical to those rooted at node 4 and 7, respectively. In the rearranged string id array, $B'[3]$, $B'[4]$, $B'[6]$, and $B'[7]$ (See Figure 3) are removed.*

Due to the redundancy caused by variant enumeration, which generates 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 algorithm is that merging common subtrees facilitates the query processing

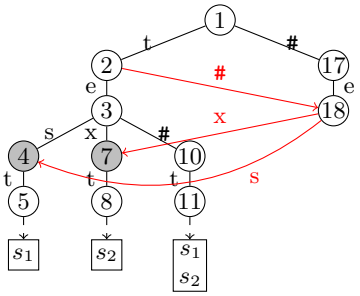


Figure 4: Index after size reduction. Gray nodes indicate edit distance computation is invoked for descending paths. New edges formed by common subtree merge are colored in red.

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:

LEMMA 4. *Consider two nodes n_1 and n_2 which share common subtrees rooted at them. 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.*

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. Moreover, any future active states expanded from $\langle n_2, u, \delta' \rangle$ will be subsumed by those expanded from $\langle n_1, u, \delta \rangle$, because they read in the same key stroke and $\delta \leq \delta'$, which implies a more margin of incoordination. In all, the correctness of the algorithm holds in spite of $\langle n_2, u, \delta' \rangle$ being discarded from the active state set. □

6. DISCUSSIONS

In this section we briefly comment on the techniques by which the IncNGTrie algorithm tackles long strings and updates in data strings.

6.1 Long Strings

Since the number of variants of a data string is $|s|^\tau$, long strings may bring about considerable performance issues in the indexing construction. Our remedy is to truncate data strings at a length l_p, and only use the truncated prefix to generate variants. Hence the number of variants of data string is at most $|l_p|^\tau$. As the truncation may introduce false positives to query processing, edit distance computation is invoked for the remaining length when the query's and the data string's lengths exceed l_p.

6.2 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 choose to 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. In order not to rebuild the rearranged string id array from scratch,

we choose to implement the rearranged string id array of the auxiliary index in a linked list so that the insertion of the string id is done in $O(1)$ time. The auxiliary index is merged with the main index through an offline logarithmic merging [22], which is also adopted by many IR 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 won't 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.

7. EXPERIMENTS

We report experiment results and our analyses.

7.1 Experiment Setup

The following algorithms are compared in the experiment.

- **ICAN** and **ICPAN** are two state-of-the-art, direct trie-based algorithms for error-tolerent autocompletion [20]. The ICAN algorithm is very similar to the method with the *full strategy* in [8], while ICPAN improves ICAN by further reducing the size of active node set by keeping only pivotal active nodes. [20] has experimentally showed that ICAN has similar performance as the algorithm in [8], and both are inferior to the ICPAN algorithm.
- **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.

All the experiments were carried out on a PC with an AMD Opteron 2.4GHz Processor and 96GB RAM, running Ubuntu 4.4.3. We implemented all the algorithms in C++ and in a main memory fashion.

We select three publicly available datasets. The first two were used in prior studies [8, 20].

- **DBLP** contains about 1.1 million bibliography records in Computer Science.
- **MEDLINE** is a set of about 4 million journal citations and abstracts of biomedical literature.
- **UNIREF** is the UniRef90 protein sequence data from the UniProt project.

For DBLP and MEDLINE datasets, we tokenize the dataset into terms with white spaces and punctuations. For UNIREF, we take the first 12 characters of each sequence as a term. Each term is then regarded as a data string. Statistics about the preprocessed datasets are provided in Table 2.

We follow [8] to randomly sampled 1,000 strings from each dataset as queries.

We measure (1) the **active state size**, which is the number of active nodes for ICAN, or the number of pivotal active nodes for ICPAN, or the number of active states for IncNGTrie; (2) the **query time**, which consists of the **searching time** and the **result fetching time**. The former is to maintain active states/nodes, and the latter is to fetch query results. (3) the **index size**, which consists of the trie and all the auxiliary data structures. All the measures are averaged over 1,000 queries.

Table 2: Dataset Statistics

Dataset	$\|S\|$	avg. $\|s\|$	$\|\Sigma\|$
DBLP	351,207	8	27
MEDLINE	1,782,517	10	27
UNIREF	356,585	12	26

7.2 Query Processing Performance

7.2.1 Query Response Time

We plot the average query response times of the three algorithms under query lengths of 4 and 7 and varying τ in Figures 5(a) –5(f). Note the the response times are not accumulated for previous characters but measured only when the 4-th or 7-th character is typed. Response times are decomposed into searching time (top) and result fetching time (bottom). NG, IC, and IP denote IncNGTrie, ICAN, and ICPAN, respectively.

We observe that our method has the best response time among the three. The advantage is more substantial when τ or query length is large. For example, we can achieve up to 308x speedup against ICAN and up to 150x speedup against ICPAN. The major reason for the much longer response time of direct trie-based method is the lengthy searching time, which will be analyzed in detail later. The response time for short queries (len = 4) are longer than long queries (len = 7), mainly because there are much more results under a fixed threshold when the query is short.

The runtime performances on short queries (query length ≤ 3) are also evaluated. We show the result on MEDLINE in Figures 5(g) –5(i). When the query length is no larger than τ, all the data strings become query results. Hence in this case, we replace the result fetching phase with a scan over the set of data strings. IncNGTrie is the fastest under most settings and the gap increases rapidly for larger τ, because it consumes less searching time. There are exceptional cases where IncNGTrie is slower than direct trie-based algorithms by a very small margin (e.g., when $|q| = 2, \tau = 1$), because result fetching time dominates under these settings and IncNGTrie spends a little more time in this phase.

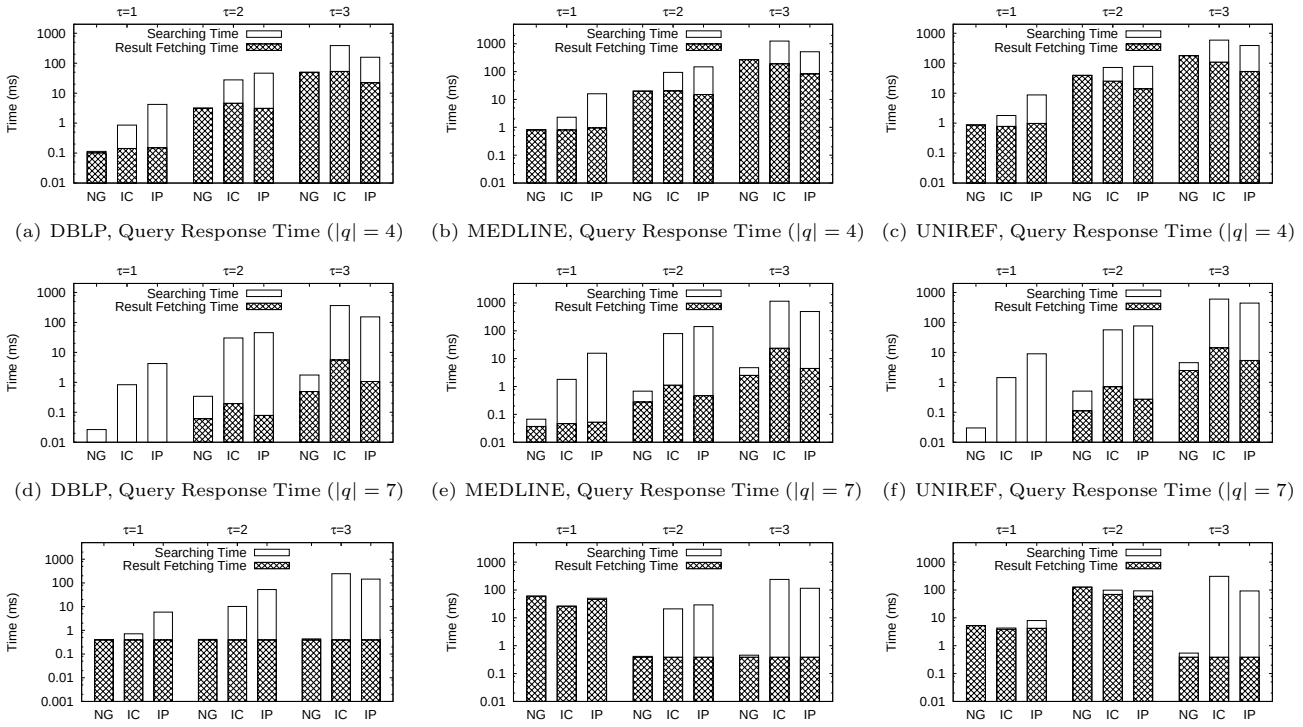
In the following, we analyze searching and result fetching times of the algorithms separately. In interest of space, we only show the results when $\tau = 3$.

7.2.2 Searching Time

We measure the active state numbers of the three algorithms on the three datasets when the i-th character of the query is input to the system (called *prefix length* [20]), and show the result in Figures 6(a) – 6(c). We observe that

- The active state numbers of the algorithms first increase with prefix length, and then decrease. For direct trie-based algorithms, the peak is reached when prefix length is 3 or 4 (note that $\tau = 3$), while the peak of IncNGTrie is reached when prefix length is 6 or 7.
- The maximum active state numbers of direct trie-based methods are much larger than that of IncNGTrie. The numbers on MEDLINE are 97k and 12k for ICAN and ICPAN respectively, but only 468 for IncNGTrie.

The active state size reduces under larger prefix lengths, because the query becomes more selective when characters are appended. In this case, IncNGTrie may have slightly larger active states than direct trie-based approaches due to neighborhood generation; e.g., when prefix length is 9.



(a) DBLP, Query Response Time ($|q| = 4$) (b) MEDLINE, Query Response Time ($|q| = 4$) (c) UNIREF, Query Response Time ($|q| = 4$)

(d) DBLP, Query Response Time ($|q| = 7$) (e) MEDLINE, Query Response Time ($|q| = 7$) (f) UNIREF, Query Response Time ($|q| = 7$)

(g) MEDLINE, Query Response Time ($|q| = 1$) (h) MEDLINE, Query Response Time ($|q| = 2$) (i) MEDLINE, Query Response Time ($|q| = 3$)

Figure 5: Overall Query Processing Performance

We also observe the early stage explosion of direct trie-based methods and the benefit of reducing active states by the use of deletion marked variants. On DBLP, IncNGTrie reduces the active state number by 78.2 times from ICPAN. On UNIREF and MEDLINE, the reduction rate over ICPAN can be up to 80.9 times and 196.8 times.

The huge difference in active state numbers is also reflected in the searching time, as it is proportional to the number of active states (See Section 2.2). Figures 6(d) – 6(f) show the times of the searching phase of the three algorithms with varying prefix length. IncNGTrie can be up to 1613.4 times faster than the runner up ICPAN on DBLP, 1338.1 times on UNIREF, and 3175.1 times on MEDLINE. All the maximum speed-ups are achieved when query length is 1. One may notice that IncNGTrie has more active states when query length is 9, but faster searching time. This is because among its active states, the cursors of some states are over current query length, and hence are directly included into new active state set without further expansions; while all the active nodes of direct trie-based approaches are used to expand new active nodes.

Finally, we note that no nodes in the first τ levels of the direct trie can be pruned, yet all of them have to be computed to enable the computation of the following input characters. This will inevitably lead to slow response time for the first few character entered by the user. This is the inherent limitation of direct trie-based method, and part of the technical reasons why buffered strategy and precomputing results via alphabet reduction are used in [8] to alleviate this problem.

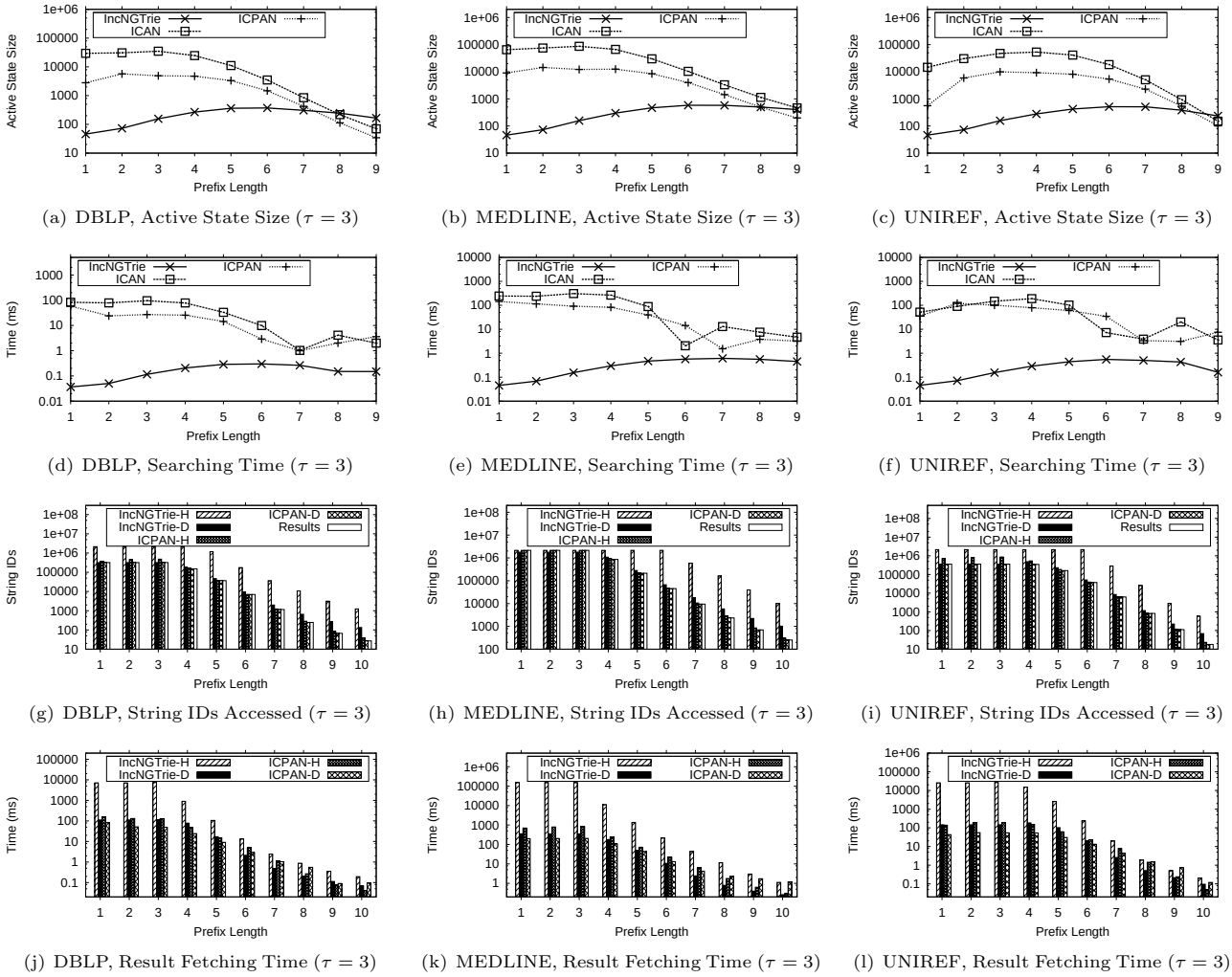
7.2.3 Result Fetching Time

We then evaluate the result fetching time with a straightforward and the optimized duplicate elimination techniques described in Section 4, and compare the result fetching time with ICPAN. Results on ICAN are not shown, as it has even more duplicates than ICPAN. We consider the following two strategies to remove duplicates when fetching query result:

- HashTable. The algorithm uses a hash table to return distinct strings from the set of string IDs associated with the leaf nodes reachable from active states.
- Dedup. The algorithm employs duplicate removal techniques. For IncNGTrie, we use the techniques proposed in Section 4 to deal with the three types of duplicates. For ICPAN, we use the following optimization. As the duplicates of ICPAN are caused by ancestor-descendant relationship among pivotal active nodes, we first index all pivotal active nodes in a hash table, and then test for each pivotal active node if one of their ancestors on level $|q| - \tau$ or below is in the hash table. The algorithm takes $O(\tau \cdot |P|)$ time, where $|P|$ is the number of pivotal active nodes.

These two strategies can be combined with either algorithm and hence we have four resulting algorithms. Figures 6(g)– 6(i) compare the string ids accessed by the different combinations for respective query length when $\tau = 3$. We also show the number of distinct result strings (denoted Results). As can be seen, HashTable strategy (denoted -H) accesses a number of string ids up to 2M for IncNGTrie. When we use Dedup strategy (denoted -D), it can be reduced by one order of magnitude, and is very close to the distinct results. For ICPAN, the number of accessed string ids using HashTable is slightly larger than the distinct results, and the duplicates are removed when Dedup strategy is applied.

Figures 6(j)– 6(l) show corresponding result fetching time. After applying Dedup on IncNGTrie, it reduces the result fetching time from HashTable by up to almost two orders of magnitude. Another interesting observation is that the



(a) DBLP, Active State Size ($\tau = 3$) (b) MEDLINE, Active State Size ($\tau = 3$) (c) UNIREF, Active State Size ($\tau = 3$)

(d) DBLP, Searching Time ($\tau = 3$) (e) MEDLINE, Searching Time ($\tau = 3$) (f) UNIREF, Searching Time ($\tau = 3$)

(g) DBLP, String IDs Accessed ($\tau = 3$) (h) MEDLINE, String IDs Accessed ($\tau = 3$) (i) UNIREF, String IDs Accessed ($\tau = 3$)

(j) DBLP, Result Fetching Time ($\tau = 3$) (k) MEDLINE, Result Fetching Time ($\tau = 3$) (l) UNIREF, Result Fetching Time ($\tau = 3$)

Figure 6: Searching and Result Fetching Performance

fastest combination is ICPAN + Dedup under small query length, but becomes either IncNGTrie + Dedup (on MEDLINE) or ICPAN + HashTable (on DBLP and UNIREF) when the length increases. This is because there are more duplicate results caused by ancestor-descendant relationship under small query length, where the equipped Dedup strategy is very effective at removing them. As query length increases, duplicates caused by ancestor-descendant relationship are reduced, and thus the ancestor lookup hardly returns any results but leads to more runtime overhead.

7.3 Scalability

7.3.1 *Varying Alphabet Size*

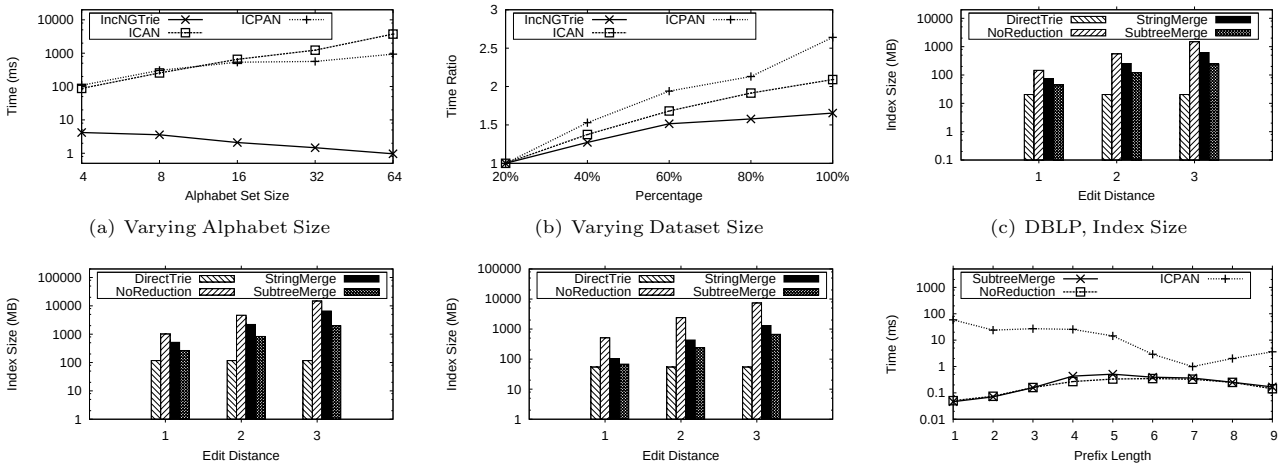
We study the searching time of the three algorithms with varying alphabet size. Synthetic datasets are generated with $|\Sigma| = 4$, 8, 16, 32 and 64, each containing 50k data strings. Figure 7(a) shows the accumulated searching times when query length reaches 8 at a τ of 3. The searching times of direct trie-base approaches rapidly grow with alphabet size, though ICPAN has a smaller growth rate. On the contrary, the searching time of IncNGTrie decreases when we move alphabet size towards larger values, because its active state size is insensitive to the alphabet size, and the query becomes more selective. When the alphabet contains 4 characters, IncNGTrie is 26.1 times faster than ICPAN. When there are 64 characters, the gap enlarges to 967.8 times.

7.3.2 *Varying Dataset Size*

We study the scalability of the algorithms by varying dataset size. We randomly sampled 20% to 100% data strings from MEDLINE so that the data and result distributions remain approximately the same as the whole corpus. Figure 7(b) shows the ratio of accumulated searching times against that on 20%, when query length reaches 8 and τ is 3. The general trend is that the searching times of the three algorithms all grows with larger dataset size. IncNGTrie exhibits the slowest growth rate, followed by ICPAN. When the dataset size jumps from 20% to 100%, the searching time increases by 2.6 times for ICAN, 2 times for ICPAN, and 1.6 times for IncNGTrie, showcasing the less sensitiveness of IncNGTrie to dataset size than direct trie-based approaches.

7.4 Index Size Reduction

The two techniques proposed in Section 5 for index size reduction are evaluated. We use the term "NoReduction" to



(a) Varying Alphabet Size (b) Varying Dataset Size (c) DBLP, Index Size

(d) MEDLINE, Index Size (e) UNIREF, Index Size (f) DBLP, Searching Time ($\tau = 3$)

Figure 7: Scalability & Index Size Reduction

Table 3: Index Construction Time (DBLP)

τ	ICPAN	NoReduction	SubtreeMerge
1	2.9s	7.4s	24.5s
2	2.9s	33.7s	95.2s
3	2.9s	116.0s	266.8s

denote the IncNGTrie algorithm without any index size reduction. "StringMerge" denotes that we apply the common data string merge. "SubtreeMerge" denotes that the common subtree merge is further applied.

7.4.1 Effect on Index Size

Figures 7(c) – 7(e) show the index sizes of the algorithms on the three datasets varying τ. Both optimizations are effective at reducing index size, and the reduction is more substantial with an increasing threshold. For example, on UNIREF and $\tau = 3$, StringMerge reduces index size by 5.6 times, and SubtreeMerge further reduces it by 2.0 times.

Our algorithm has larger index sizes than direct trie-based approaches (denoted DirectTrie in the figures), and the gap increases for larger τ. This is expected as the number of variants is exponential in τ. On UNIREF, our index sizes are 1.3 times that of direct trie-based approaches when $\tau = 1$, 4.4 times when $\tau = 2$, and 12.3 times when $\tau = 3$.

7.4.2 Effect on Runtime Performance

Figure 7(f) shows the searching time before and after applying the two index reduction techniques on DBLP when $\tau = 3$. With index reduction, the searching time exhibits almost no change except a slight increase (up to 1.6 times) at a prefix length of 4 or 5. The increase results from the common data string merge that replaces node expansion with edit distance computation and hence imposes more overhead.

We also compare the index construction time and show the result on DBLP in Table 3. Direct trie-based approach spends the least time on index construction as it simply builds a trie from the data strings. The index construction times of neighborhood generation-based approaches grow with τ due to more variants generated. With index reduction, the time spent on index construction increases by around 3 times, but is still in an affordable manner under the largest threshold setting.

8. RELATED WORK

Query autocompletion has been adopted in many applications such as Web search engines, command shells in operating systems, etc. They either rely on query logs or a predefined dictionary to process the autocompletion.

There has also been considerable interest in query autocompletion in research community. The Reactive Keyboard [11] is a device that accelerates typewritten communication by predicting what the user is going to type. Grabski and Scheffer [14] 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 built on underlying document corpus to provide answers to word-level autocompletions. Nandi and Jagadish [26] studied the problem of autocompletion at the level of a phrase containing multiple words.

The autocompletions with edit distance to tolerate errors were first studied in [17] and [8]. Li *et al.* [20] improved the method proposed in [17] to reduce memory consumption and query response time, both included in the TASTIER project [21], targeting type-ahead search in which the system returns answers on-the-fly as users type in the query. Apart from edit distance, cosine similarity [4] and Markov n-gram transformation model [13] are also adopted for error tolerance in the autocompletion task.

Another line of work aims at query recommendations, taking a full query and making arbitrary reformulations to assist users. The proposed solutions are mainly based on query clustering [3, 28], session analysis [16], or search behavior models [31], just to name a few recent studies.

Edit distance is a common distance function for approximate string matching. We refer readers to [7] 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 [25]. To reduce its size, deletion neighborhood was proposed for $\tau = 1$ [23] and extended to general case [6]. k-errata trie [9] was proposed to process dictionary matching with edit distance constraints and was later improved by [30]. Unlike our algorithm, [30] generates full neighborhood, not only deletion, but also insertion and substitution,

though a letter inserted or substituted is represented in a wildcard like our #. Subtree are also merged in [30]. However, as their main purpose is to speed up query processing, subtrees are merged so that 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 [15, 27, 19] and membership checking [34, 12].

Our index reduction techniques are similar to condensed data cube and its indexing based on prefix and suffix sharing [29] or equivalence classes on identical aggregate values [18]. Our deletion mark # can be regarded as "ALL" in data cube queries, the string ids are the measure attribute, and the aggregate function is a UNION. We briefly discuss the differences: (1) The fact table records are of the same dimensionality, whereas the data strings in our problem may vary in length. (2) It is unknown how to support our query using the index structure proposed in [29] or [18]. (3) We consider duplicate removal in our index, whereas it is unclear how to efficiently deal with it using the two techniques.

9. 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 an algorithm to organize the trie index on the basis of deletion neighborhood of data strings. The new algorithm achieves a very small and alphabet-insensitive active state size to speed up 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 algorithm outperforms existing solutions by up to two orders of magnitude in terms of query response time.

Acknowledgements. Chuan Xiao, Yoshiharu Ishikawa, Koji Tsuda, and Kunihiko Sadakane are supported by FIRST Program, Japan. Jianbin Qin and Wei Wang are supported by ARC Discovery Projects DP130103401 and DP130103405.

10. REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] J.-I. Aoe. An efficient digital search algorithm by using a double-array structure. *IEEE Trans. Software Eng.*, 15(9):1066–1077, 1989.

[3] R. A. Baeza-Yates, C. A. Hurtado, and M. Mendoza. Improving search engines by query clustering. *JASIST*, 58(12):1793–1804, 2007.

[4] Z. Bar-Yossef and N. Kraus. Context-sensitive query auto-completion. In *WWW*, pages 107–116, 2011.

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

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

[7] L. Boytsov. Indexing methods for approximate dictionary searching: Comparative analysis. *ACM Journal of Experimental Algorithmics*, 16(1), 2011.

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

[9] R. Cole, L.-A. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. In *STOC*, pages 91–100, 2004.

[10] J. Daciuk. Comparison of construction algorithms for minimal, acyclic, deterministic, finite-state automata from sets of strings. In *CIAA*, pages 255–261, 2002.

[11] J. J. Darragh, I. H. Witten, and M. L. James. The reactive keyboard: A predicive typing aid. *IEEE Computer*, 23(11):41–49, 1990.

[12] D. Deng, G. Li, and J. Feng. An efficient trie-based method for approximate entity extraction with edit-distance constraints. In *ICDE*, pages 762–773, 2012.

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

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

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

[16] Q. He, D. Jiang, Z. Liao, S. C. H. Hoi, K. Chang, E.-P. Lim, and H. Li. Web query recommendation via sequential query prediction. In *ICDE*, pages 1443–1454, 2009.

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

[18] L. V. S. Lakshmanan, J. Pei, and Y. Zhao. Socqet: Semantic olap with compressed cube and summarization. In *SIGMOD Conference*, page 658, 2003.

[19] G. Li, D. Deng, J. Wang, and J. Feng. Pass-Join: A partition-based method for similarity joins. *PVLDB*, 5(1):253–264, 2012.

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

[21] G. Li, S. Ji, C. Li, J. Wang, and J. Feng. Efficient fuzzy type-ahead search in tastier. In *ICDE*, 2010.

[22] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*. Cambridge University Press, 2008.

[23] M. Mor and A. S. Fraenkel. A hash code method for detecting and correcting spelling errors. *Commun. ACM*, 25(12):935–938, 1982.

[24] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *SODA*, pages 657–666, 2002.

[25] E. W. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4/5):345–374, 1994.

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

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

[28] E. Sadikov, J. Madhavan, L. Wang, and A. Y. Halevy. Clustering query refinements by user intent. In *WWW*, pages 841–850, 2010.

[29] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis. Dwarf: shrinking the petacube. In *SIGMOD Conference*, pages 464–475, 2002.

[30] D. Tsur. Fast index for approximate string matching. *J. Discrete Algorithms*, 8(4):339–345, 2010.

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

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

[33] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.

[34] W. Wang, C. Xiao, X. Lin, and C. Zhang. Efficient approximate entity extraction with edit constraints. In *SIMGOD*, pages 759–770, 2009.

[35] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD Conference*, pages 425–436, 2001.