Autocompletion for Prefix-Abbreviated Input

Sheng Hu
Nagoya University
& Kyoto University
hu@db.ss.is.nagoya-u.ac.jp

Chuan Xiao ✉
Nagoya University
& Osaka University
chuanx@nagoya-u.jp

Jianbin Qin
Shenzhen Institute of Computing Sciences, Shenzhen University
jqin@sics.ac.cn

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

Qiang Ma
Kyoto University
qiang@i.kyoto-u.ac.jp

ABSTRACT

Query autocompletion (QAC) is an important interactive feature that assists users in formulating queries and saving keystrokes. Due to the convenience it brings to users, QAC has been adopted in many applications, including Web search engines, integrated development environments (IDEs), and mobile devices. For existing QAC methods, users have to manually type delimiters to separate keywords in their inputs. In this paper, we propose a novel QAC paradigm through which users may abbreviate keywords by prefixes and do not have to explicitly separate them. Such paradigm is useful for applications where it is inconvenient to specify delimiters, such as desktop search, text editors, and input method editors. E.g., in an IDE, users may input `getnev` and we suggest `GetNextValue`. We show that the query processing method for traditional QAC, which utilizes a trie index, is inefficient under the new problem setting. A novel indexing and query processing scheme is hence proposed to efficiently complete queries. To suggest meaningful results, we devise a ranking method based on a Gaussian mixture model, taking into consideration the way in which users abbreviate keywords, as opposed to the traditional ranking method that merely considers popularity. Efficient top-k query processing techniques are developed on top of the new index structure. Experiments demonstrate the effectiveness of the new QAC paradigm and the efficiency of the proposed query processing method.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-5643-5/19/06...$15.00
https://doi.org/10.1145/3299869.3319858

CCS CONCEPTS

• **Information systems → Query suggestion**;

KEYWORDS

autocompletion, query suggestion, prefix-abbreviated input

ACM Reference Format:
Sheng Hu, Chuan Xiao✉, Jianbin Qin, Yoshiharu Ishikawa, and Qiang Ma. 2019. Autocompletion for Prefix-Abbreviated Input. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3299869.3319858

1 INTRODUCTION

Query autocompletion (QAC) is an important feature that guides users to type a query correctly while reducing the effort to submit the query. As a user types the query into the search box, QAC gives possible suggestions that contain the currently input characters as a prefix. In addition to its prevalence among many visible features of Web search engines, QAC has also been adopted in various applications where typing is laborious and error-prone, such as command shells, desktop search, and mobile devices. Due to its importance, QAC has received considerable attention from information retrieval [4, 41, 50] and database research communities [11, 12, 25, 32, 36].

For existing QAC methods [4, 12, 32, 41, 50] (including type-ahead search [25, 27] that directly identifies matching documents), users need to manually separate keywords in the input and then the system takes the input characters as the prefixes of keywords to match. Hence a limitation is that these methods are unable to handle the case when users prefer not to manually separate keywords in the input or it is inconvenient to do so. Such input is common in many applications, including Web services managing large datasets:

- In text editors and integrated development environments (IDEs), users may type a variable/function name using concatenation of keyword prefixes or first letters, e.g., typing `tbf` for `textbf` and `getnev` for `GetNextValue`. A similar

feature is provided by Eclipse, a prevalent Java IDE, but it only supports acronyms composed of first letters.
- In input method editors (IMEs), a common practice is to save keystrokes by omitting some vowels since typing is laborious and error-prone on mobile devices, e.g., typing `luoshj` for `luoshanji` (Los Angeles) in Chinese pinyin.
- In desktop search, users may search files using the first few characters of the words in file names, e.g., typing `nagoulh` to search `NagoyaULetterHead.pdf`.
- For search engines, when searching proper names comprising multiple morphemes, a common scenario for biological and medical terms, users may want to give only a few characters for each morpheme, e.g., typing `fuspch ginv` for `fusospirochetal gingivitis`, where morphemes are separated by underline.

In this paper, we propose a novel QAC feature by which users do not have to explicitly separate keywords. We focus on the input of *abbreviations using keywords' prefixes*. This is common in practice: by the statistics of ALLIE [46], a dataset of two million medical terms extracted from MEDLINE, the abbreviations of 82% terms belong to this category. Inputting user-defined keywords' prefixes to look for terms is recognized and utilized by the participants in a study on QAC for medical vocabulary [39]. For software engineering, a user study showed that using acronym-like abbreviated input of multiple keywords reduces 30% time and 41% keystrokes over conventional code completion [19]. Prefix-abbreviated input is also partially supported by IMEs like Sougou Pinyin, though such feature does not respond quickly on cloud dictionaries. We call the proposed feature **q**uery **a**uto**c**ompletion for **p**refix-**a**bbreviated input (QACPA). It provides a solution to the demand in the aforementioned applications. To save keystrokes, users may also input the prefixes of the first few keywords instead of all; e.g., we suggest `GetNextValue` for the query `getn`, where `Value` is saved. Despite focusing on prefix-abbreviated input, QACPA is designed to be *extensible* to the following cases: (1) keywords are manually separated (traditional QAC), (2) keywords are skipped, (3) keywords are abbreviated by non-prefixes, and (4) full-text search for terms.

Most traditional QAC methods rely on a trie to index data strings and process queries. For QACPA, one may also index data strings [1] in a trie and design a baseline algorithm to traverse the trie incrementally to find matching data strings. The nodes matched by the query are called *active nodes*. The efficiency critically depends on the number of active nodes per keystroke and the time complexity of finding an active node. Since users may not separate keywords in the input, the number and the time complexity (discussed in Section 2.2) are both $O(|T|)$, where $|T|$ denotes the number of nodes in the trie, in the worst case and typically large for online query processing, rendering this algorithm inefficient for QACPA. With the growing popularity of online text editors/IDEs (e.g., Overleaf and IBM Bluemix) and cloud IMEs, the demand on efficiency is increasing. E.g., for popular cloud IMEs like Sougou Pinyin, the number of active users is over 300 million per day [34].

Seeing the inefficiency of the trie-based method for QACPA, we design an index, called *nested trie*, composed of an outer trie and a number of inner tries nested on outer trie nodes. Based on this index, we are able to reduce the number of nodes matched by the query, exploiting the shared characters in the indexed keywords. The index also includes the data structure to quickly identify these matching nodes. Hence an efficient query processing algorithm is devised. We show that the number of active nodes by this algorithm is at most $2^{|q|-1}$ per keystroke and practically very small (4 nodes per keystroke on ALLIE). The time complexity of finding an active node n is $O(|I_n|)$, where $|I_n|$ is the number of intervals (formally defined in Section 4.2) to cover the underlying data strings of n. By several optimizations, this process is reduced to sublinear time and very fast in practice.

To rank results for suggestions, in contrast to many traditional QAC methods that consider only string popularity (static scores), we develop a ranking method for QACPA by combining string popularity and the way in which users abbreviating keywords. A Gaussian mixture model is utilized to predict the probability that a user abbreviates keywords into a given set of prefixes observed in the input. We also present a top-k query processing algorithm to efficiently compute the top-k answers with respect to the new ranking method by integrating a series of early termination techniques.

Experiments are conducted on real datasets that cover several applications of QACPA. The results demonstrate the effectiveness of QACPA: it saves an average of around 20% keystrokes compared to traditional QAC. The experiments also show that the proposed ranking method remarkably improves the accuracy, and the proposed query processing algorithm has superior performance to alternative solutions with up to two orders of magnitude speedup.

Our main contributions are summarized as follows.

(1) We propose a novel QAC feature by which users may abbreviate and concatenate keywords by prefixes.
(2) We design an indexing and query processing method to efficiently complete the queries by the new QAC feature.
(3) We propose a ranking method and integrate it into our query processing method for efficient top-k retrieval.
(4) We perform extensive experiments on real datasets. The results demonstrate the effectiveness of the new QAC feature and the efficiency of the query processing method.

The rest of our paper is organized as follows: Section 2 defines the problem and introduces preliminaries. Section 3

[1] In this paper, we assume to perform autocompletion over a pre-defined dictionary of data strings, in line with [12].

Table 1: Example dataset S.

ID	String	Popularity
s^1	`AddNextValue`	0.3
s^2	`GenNewValue`	0.1
s^3	`GenNullValue`	0.3
s^4	`GetNextChar`	0.2
s^5	`GetNextValue`	0.6
s^6	`GetNextVector`	0.4
s^7	`GetTimerOfDay`	0.5
s^8	`GroupNewValue`	0.1
s^9	`ReadNextValue`	0.2

presents the index structure. The query processing algorithm appears in Section 4. Section 5 introduces the ranking method and the algorithm for fast top-k retrieval. Section 6 discusses miscellaneous extensions, including skipping keywords, non-prefix abbreviations, full-text search, and data updates. Experimental results and analyses are reported in Section 7. Section 8 reviews related work. Section 9 concludes the paper.

2 PRELIMINARIES

2.1 Problem Definition

is a finite alphabet of symbols; each symbol is also called a character. A string s is an ordered array of symbols drawn from . $|s|$ denotes the length of s. $s[i]$ is the i-th character of s, starting from 1. $s[i \ldots j]$ is the substring between position i and j. $*$ is a Kleene star to represent a string of any number of characters, including an empty string. Given two strings s_1 and s_2, s_1 is a prefix of s_2, denoted by $s_1 \preceq s_2$, iff $s_1 = s_2[1 \ldots i]$, $1 \le i \le |s_1|$. We also use the notation $\overleftarrow{s}$ to denote any prefix of s. $s_1 s_2$ denotes the concatenation of s_1 and s_2. An array $[\, s_1, \ldots, s_n \,]$ ($n \ge 1$) is called a segmentation of s, iff $s = s_1 s_2 \ldots s_n$. Each s_i is called a segment of s.

Let S be a dataset of strings. Each string $s^i \in S$ is segmented into a set of substrings, called *keywords*. This is done by delimiters (white space, punctuation, capital letters, etc.) or morphemes/syllables, depending on the application scenario. For ease of exposition, we assume consists of English letters only and use *capital letters* to denote the initial characters of keywords. Table 1 gives an example dataset S. `AddNextValue` is segmented into three keywords: `Add`, `Next`, and `Value`.

Next we define related concepts for prefix-abbreviated input. Consider a string s segmented into keywords $[s_1, \ldots, s_n]$. Given a query string q, we say q is a *prefix-abbreviated match* of s, denoted by $q \sqsubseteq s$, iff $q = \overleftarrow{s_1}\overleftarrow{s_2} \ldots \overleftarrow{s_i}$, $1 \le i \le n$; in other words, q is the concatenation of the prefixes of s's first i keywords. E.g., `gene` is a prefix-abbreviated match of `GetNextValue`, because `ge` and `ne` are the prefixes of `Get` and `Next`, respectively. In the rest of the paper, we use the term "**PA-match**" short for prefix-abbreviated match, while the

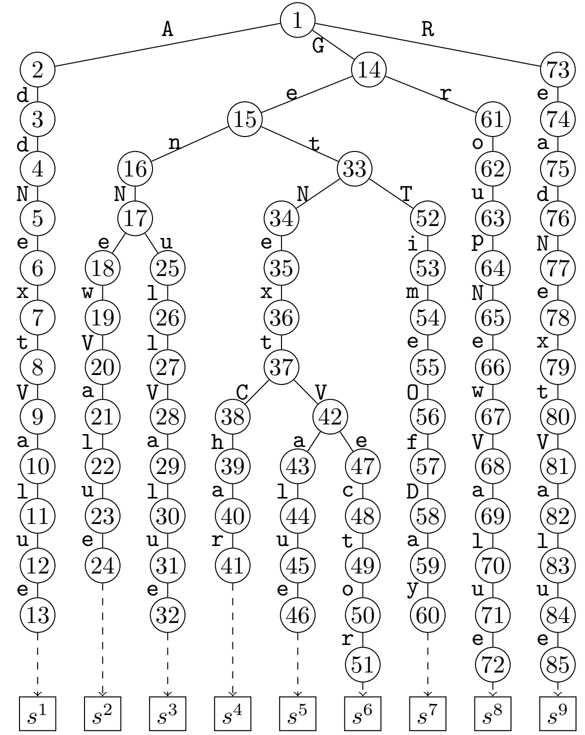


Figure 1: Trie index for the baseline method.

term "**match**" still means a character-by-character manner. Moreover, characters match **case-insensitively** unless we say they "**strictly match**" [2].

One may notice that if a PA-match occurs, the initial characters of s's keywords yield a segmentation of the query string, $[\, \overleftarrow{s_1}, \ldots, \overleftarrow{s_i} \,]$. So users do not have to explicitly specify where to segment the query. Next we define our problem.

Problem 1. *Given a dataset of strings S, a query string q, a query autocompletion for prefix-abbreviated input (QACPA) is to find all the strings $s^i \in S$, such that $q \sqsubseteq s^i$. The results are computed incrementally as the user types in characters.*

Given the dataset in Table 1 and a query `geneva`, QACPA returns s^2 and s^5 as results. Since the number of results may be large in real applications, we may sort them by a ranking function and return the top-k ones. We assume S and its index, if there is any, are stored in main memory to process QACPA.

2.2 Baseline Methods

Most prevalent QAC methods [3, 4, 25, 27, 32, 41, 42] adopt a trie index to process queries. For QACPA, we may also index data strings in a trie and design a baseline algorithm. Figure 1 shows the trie for the strings in Table 1. String IDs are linked

[2] The notion of strict match is to enforce the initial character of a keyword to match an initial one, and a non-initial character to match a non-initial one.

Algorithm 1: QACPA-Trie (q, T)

Input : q is a query string, T is a trie built on S.
Output : $\{ s^i \}$, such that $s^i \in S$ and $q \sqsubseteq s^i$.

1 $A \leftarrow \{$ the root of $T \}$; /* active node set */
2 **foreach** keystroke $q[i]$ **do**
3     $A' \leftarrow \emptyset$;
4   **foreach** $n \in A$ **do**
5   **if** n has a child n' through non-initial character $q[i]$ **then**
6             $A' \leftarrow A' \cup \{ n' \}$;
7   **foreach** n's descendant n' **do**
8   **if** the incoming edge of n' is initial character $q[i]$ **and** there does not exist any node on the path from n to n' through an initial character **then**
9                 $A' \leftarrow A' \cup \{ n' \}$;
10     $A \leftarrow A'$;
11 $R \leftarrow \emptyset$;
12 **foreach** $n \in A$ **do**
13     $R \leftarrow R \cup$ string IDs stored in the subtree rooted at n;
14 **return** TopK(R);

Table 2: Active nodes by the baseline algorithm.

Key	∅	g	e	n	e	v	a
Active Nodes	1	14	15	16 17 34	18 35	20 42	21 43

to the end of strings. The query processing algorithm (pseudo-code in Algorithm 1) consists of two phases. In the **searching phase** (Lines 1 – 10), for every keystroke, it traverses the trie to find the prefixes PA-matched by the query string. A keystroke can either match a non-initial character of the current keyword (Line 5) or the initial character of the next keyword (Line 8). The frontiers of the PA-matched prefixes are called *active nodes*. In the **result fetching phase** (Lines 11 – 14), it calculates the union of the active nodes' underlying strings and returns the top-k ones sorted by a ranking function.

Example 1. *Consider a query string* q = `geneva`*. Table 2 shows the active nodes (numbered in Figure 1) for each keystroke using the baseline algorithm. For keystroke* `n`*, nodes 16 and 17 are both active though 16 is 17's parent. This is because* `Gen` *(16) and* `GenN` *(17) are PA-matched through different segmentations of* `gen`*. We need to keep both of them for future keystrokes.* s^2 *and* s^5 *are PA-matched by the query string as they are the underlying strings of nodes 21 and 43, respectively.*

The efficiency of query processing mainly depends on two factors: the number of active nodes and the cost of finding them. Both result in considerable overhead for the baseline algorithm:

- In real data, it is common for data strings to share initial characters of keywords. However, they might be indexed in different branches in the trie, and the baseline algorithm goes through all these branches. In Example 1, s^2 and s^5 share initial characters `G`, `N`, `V` in all their keywords. The algorithm has to include both 20 and 42 as active nodes to guarantee the correctness. This yields a worst-case $O(|T|)$ number of active nodes per keystroke, where $|T|$ is the number of nodes in the trie.
- As the user types a keystroke, the algorithm has to traverse the subtree rooted at every active node to find new active ones (Line 8), because the keystroke can match initial characters that are not directly connected to the current active nodes. In Example 1, for keystroke `v`, the baseline algorithm has to find out if there is a `V` in the subtrees rooted at nodes 18 and 35, respectively. This yields a worst-case $O(|T|)$ time complexity to find an active node.

Besides this baseline algorithm, another possibility is to index keywords separately and enumerate all possible ways of query segmentation. The techniques for multiple keyword type-ahead search [25] or multi-dimensional substring search [16, 22] are then utilized to process the segments. Although it does not suffer the aforementioned drawbacks, it requires an intersection of the string ID lists for all the keywords that contain a query segment as prefix, in order to find the strings PA-matched by the whole query string. E.g., for a query segmentation [`ge`; `ne`; `va`], it has to intersect the string IDs for the keywords beginning with `ge`, `ne`, or `va`. Although the processing of such intersection can be accelerated using the method in [25, 26] (identifying the shortest list and checking if the string IDs in this list exist in all the other lists using a forward index), there are still $2^{|q|-1}$ ways to segment the query string, each involving 1 to $|q|$ string ID lists. Hence the total number of lists to be merged is $2^{|q|-2} \cdot (|q| + 1)$, resulting in an $O(2^{|q|-2} \cdot (|q| + 1) \cdot L)$ time complexity, where L is the average length of the shortest one among the 1 to $|q|$ lists to merge. Since the lists can be very long for short query segments, the cost becomes prohibitive as the query length grows.

We may also convert the QACPA semantics to a regular expression, e.g., `^G(e|[a-z]*E)(n|[a-z]*N)` for a QACPA query `gen`. Then a regular expression search algorithm [3] is applied to find the matching strings in S. It simulates a DFA in the trie that indexes the data strings. For the converted pattern, this algorithm is equivalent to the trie-based baseline algorithm, because matching sub-patterns like `[a-z]*E` is exactly the process of traversing subtrees to find the next non-initial character and the other sub-patterns are character-by-character match. Other existing regular expression search methods, such as [33, 53], either have to scan all the strings in S or rely on filtering techniques that exploit selective substrings in the pattern, which hardly exist in the converted pattern. In

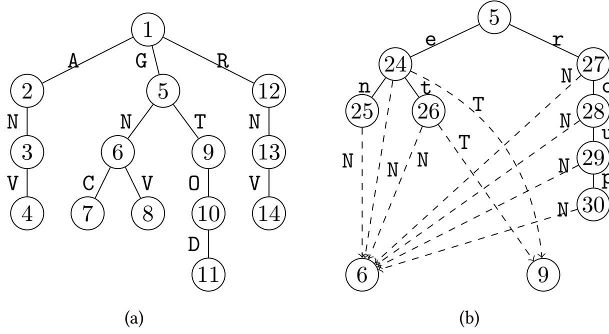


Figure 2: The outer trie (a) and an inner trie rooted at node 5 (b). Shortcuts are shown in dashed links.

addition, QACPA can be regarded as a subsequence search with prefix constraints. One may find the data strings that contain the query as a subsequence, using the subsequence search algorithm [2], and then check if they satisfy the prefix-abbreviated condition. But this method only applies to small datasets since the space complexity is $O((|S| + |\ \ |)N)$, where N is the sum of string lengths in S.

3 INDEXING

Seeing the inefficiencies of the aforementioned approaches, we design a new index to efficiently process QACPA.

Our index is called a *nested trie* in which a number of tries, called *inner tries*, are nested inside a trie, called the *outer trie*. To construct the nested trie, for each data string in S, we pick the initial characters of its keywords, and index them in the outer trie. Then for each node n in the outer trie, we index the corresponding keyword in the data string in an inner trie rooted at n. We say a node/edge is an outer trie node/edge, if it exists in the outer trie. If a node/edge exists only in an inner trie, we say it is an inner node/edge. The root of the nested trie is defined as the root of the outer trie.

Example 2. *For the strings in Table 1, we first collect the initial characters of keywords:* ANV, GNC, GNV, GTOD, *and* RNV. *Figure 2(a) shows the outer trie for the initial characters. For inner tries, we consider an example rooted at node 5, which represent the initial characters of the first keywords of* $s^2 - s^8$. *The keywords are* Gen, Get, *and* Group. *We index them in a trie rooted at node 5, shown in Figure 2(b).*

In addition to the above structure, we add links, called *shortcuts*, from inner nodes to outer nodes. For any inner node n, we use the term *initial node* to denote the root of the inner trie that contains n. For each non-initial character in a data string, if it has a succeeding keyword in the data string, then for the inner node n that corresponds to the non-initial character, we add a link from n to the outer node of the succeeding keyword. The label of the link is the initial character of the succeeding keyword. E.g., in Figure 2(b), node 27 has a succeeding keyword New, whose outer node is node 6. We add a shortcut from node 27 to 6 with label N. For the sake of space-efficiency, these links do not have to be fully materialized. Let n' be the initial node of n. We observe that the destinations of the shortcuts from n are always a subset of the destinations of the outer edges from n'. Thus, we refer to these outer edges, and at n we store this subset with a bit vector (the size is the degree of n' in the outer trie). The i-th bit represents if n has a shortcut whose destination is the same as the i-th outer edge, sorted by the alphabetical order, from n'. E.g., in Figure 2(b), to retrieve the shortcut(s) from node 27, we refer to its initial node, node 5. It has outer edges to nodes 6 and 9. We have a vector of two bits at node 27: 10, meaning that node 27 has only the first edge, which goes to node 6. By encoding shortcuts in bit vectors, the nested trie for the strings in Table 1 is shown in Figure 3.

Compared to the trie index, the nested trie combines the paths that share the initial characters of keywords. As we will see later, this reduces the number of active nodes in query processing, and the active nodes can be quickly identified. One may notice that the nested trie can be constructed by merging nodes in the original trie. But our construction method has two advantages: First, we do not need to build the original trie. Second, every inner trie can be stored in a contiguous space that enables data locality for fast access. Next we introduce the nested trie-based query processing algorithm.

4 SEARCHING ALGORITHM

The searching phase of the query processing algorithm is presented in this section. We first introduce how to generate active nodes using the nested trie and then describe the necessary list merge procedure for correct result fetching.

4.1 Finding Active Nodes

In the nested trie, an active node n is a node such that there exists at least one path, through edges and/or shortcuts, from the root to n exactly matching the query. We start from the root of the outer trie. For each keystroke, we find new active nodes using existing ones (pseudo-code shown in Algorithm 2). Given a keystroke, it can match either an initial or a non-initial character. The nested trie makes it easy for both matches. For a non-initial character, we find a new active node following an inner edge (Line 5). For an initial characters, if the current active node is an outer node, we follow an outer edge (Line 8); otherwise, we jump to the outer node for the succeeding keyword by a shortcut (Line 11).

Example 3. *Consider the query string* geneva *in Example 1. Table 3 shows the active nodes, as numbered in Figure 3, for each keystroke using the nested trie-based algorithm. For keystroke*

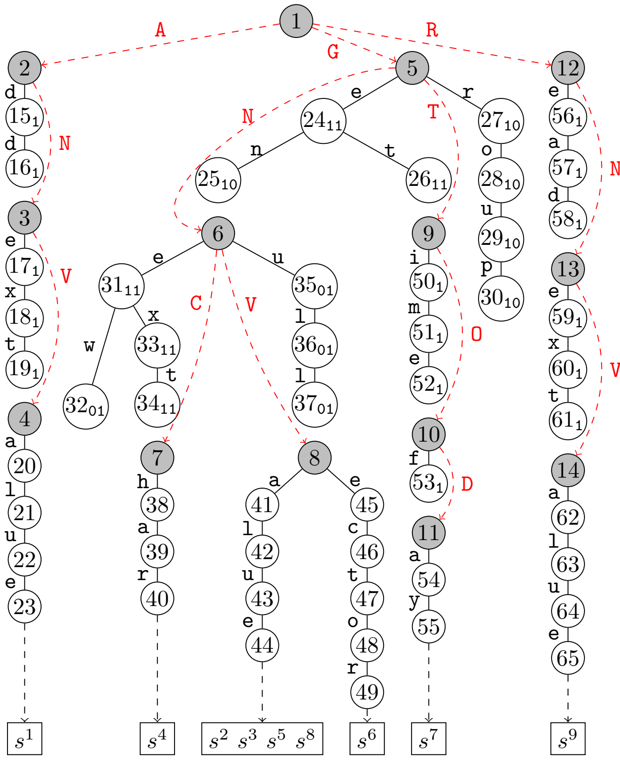


Figure 3: A nested trie. Outer nodes are marked in grey. Outer edges are shown in red dashed links. Inner edges are shown in black solid lines. Inner node numbers are subscripted by bit vectors for shortcuts, if there is any.

n, *we jump from node 24 to 6 by a shortcut, which refers to the outer edge from node 5 to 6. For keystroke* v, *we jump from node 31 to 8 in the same way. Compared with the baseline algorithm, active nodes are saved for keystrokes* n, *the second* e, v, *and* a.

For the nested trie-based algorithm, the number of active nodes per keystroke is at most equal to the number of ways to segment the query, hence $2^{|q|-1}$ (in Algorithm 2, we have one active node for the first keystroke, and each existing active node generates at most two new active nodes for any other keystroke) in contrast to the baseline algorithm's $O(|T|)$. $|q|$ is usually small in QAC tasks. The algorithm also avoids traversing entire subtrees to match characters. The time complexity of computing an active node is $O(1)$ in Algorithm 2, but we need an additional cost to report correct results, as explained in the rest of this section.

4.2 Merging Lists of Intervals

In the nested trie-based algorithm, the query might not PA-match all the underlying strings of the active nodes; e.g., in Example 3, the query string geneva has node 41 as its active node, but it PA-matches only two of the four underlying

Algorithm 2: QACPA-Nested-Trie-Search (q, T)

```
1  A ← { the root of T } ;                    /* active node set */
2  foreach keystroke q[i] do
3      A′ ← ∅;
4      foreach n ∈ A do
5          if n has a child n′ through inner edge q[i] then
6              A′ ← A′ ∪ { n′ } ;          /* for non-initial
                   character */
7          if n is an outer node then
8              if n has a child n′ through outer edge q[i] then
9                  A′ ← A′ ∪ { n′ } ;          /* for initial
                       character */
10         else
11             if n has a shortcut q[i] to n′ then
12                 A′ ← A′ ∪ { n′ } ;          /* for initial
                       character */
13     A ← A′;
```

Table 3: Active nodes by the nested trie-based algorithm.

Key	∅	g	e	n	e	v	a
Active Nodes	1	5	24	6 25	31	8	41

strings: s^2 and s^5. The reason is that all the data strings G∗Na (except those having an initial character between G and N) are indexed under node 41, while there are multiple paths from the root to node 41, each representing a different segmentation of the query string and hence different underlying strings. The searching algorithm reaches node 41 through only one of these paths.

In order not to report false positives, our remedy is to equip each node with a sorted list of intervals to indicate the strings whose prefixes strictly match one (and by the construction of nested trie, only one) path from the root of the nested trie to the node. Take node 31 in Figure 3 as an example. The prefixes of s^2, s^4, s^5, s^6, and s^8 strictly match one path from the root to node 31; e.g., for s^4, we have $1 \longrightarrow 5 \longrightarrow 24 \longrightarrow 26 \Longrightarrow 6 \longrightarrow 31$, where $\longrightarrow$ is via an edge and $\Longrightarrow$ is via a shortcut. Thus, we store a list of intervals $\{ [2, 2], [4, 6], [8, 8] \}$ at node 31 to represent these strings. To compute the intervals, given a string, for each node passed or added when building the nested trie, we insert the string ID to the list of intervals of this node. Let I_n denote the stored list of intervals of n, and $\otimes$ denote the operation of merging two lists of intervals; i.e., $\{ x_1, \dots, x_m \} \otimes \{ y_1, \dots, y_n \} = \{ x_i \cap y_j \mid 1 \leq i \leq m \wedge 1 \leq j \leq n \wedge x_i \cap y_j \neq \emptyset \}$, where each x_i or y_i denotes an interval. We have the following property.

Algorithm 3: QACPA-Nested-Trie-MonitorList (q, T)

1 $J_{root} \leftarrow [1; |S|]$;
2 **foreach** keystroke $q[i]$ **do**
3   **if** Algorithm 2 finds an active node n' from n **then**
4         $J_{n'} \leftarrow$ MergeLists$(J_n; I_{n'})$;
5   **if** $J_{n'} = \emptyset$ **then**
6   Do not insert n' into active node set A';

Proposition 1. *Consider a path $n_1; : : : ; n_k$ from the root of the nested trie to an active node n_i. All the strings in $I_{n_1} \otimes I_{n_2} : : : \otimes I_{n_k}$ are PA-matched by the query string q.*

By this property, we can merge the lists of intervals along the path while propagating active nodes in the searching phase, and ensure all the data strings in the resulting list after merge have no false positives. The union of the resulting lists for all the active nodes contains all the PA-matched strings, hence producing no false negatives. In addition, it is easy to see that if a resulting list is empty at a node n, no string will be reported for n and any future active node n' propagated from n. In this case, we do not insert n into the new active node set. With this optimization, it is guaranteed that the nested trie-based algorithm always outperforms or equals the baseline algorithm in terms of active node number:

Lemma 1. *Given a dataset S and a query q, for any keystroke of q, the number of active nodes produced by Algorithm 2 is always less than or equal to that produced by Algorithm 1.*

The pseudo-code of the above process is given in Algorithm 3. It keeps track of the merged result in a list J_n for the path from the root of the nested trie to an active node n. Initially, we start from the root and set J_{root} to include all the strings in S (Line 1). Whenever we find a new active node n' from a current one n by Algorithm 2, it computes $J_{n'}$ by merging J_n and $I_{n'}$ (Line 4). To implement Algorithm 3, we integrate it into Algorithm 2 by placing Lines 4 – 6 of Algorithm 3 after Lines 5, 8 and 11 of Algorithm 2. Next we show how this works with an example.

Example 4. *Consider Example 3. The stored lists of intervals of each active node en route is given in the table below. We start with $J_1 = [1; 9]$ and perform list merge at each step while generating active nodes. The resulting lists are also given in the table below. Node 41 is reached through the following path: 1 $\longrightarrow$ 5 $\longrightarrow$ 24 $\Longrightarrow$ 6 $\longrightarrow$ 31 $\Longrightarrow$ 8 $\longrightarrow$ 41. Finally, we have $J_{41} = \{ [2; 2], [5; 5] \}$ for the only active node 41. s^2 and s^5 are the only data strings PA-matched by the query.*

n	n^0	*List of Intervals* I_{n^0}	*Merged Result* J_{n^0}
N/A	*1*	$\{ [1; 9] \}$	$\{ [1; 9] \}$
1	*5*	$\{ [2; 8] \}$	$\{ [2; 8] \}$
5	*24*	$\{ [2; 7] \}$	$\{ [2; 7] \}$
24	*6*	$\{ [2; 6]; [8; 8] \}$	$\{ [2; 6] \}$
24	*25*	$\{ [2; 3] \}$	$\{ [2; 3] \}$
6	*31*	$\{ [2; 2]; [4; 6]; [8; 8] \}$	$\{ [2; 2]; [4; 6] \}$
31	*8*	$\{ [2; 3]; [5; 6]; [8; 8] \}$	$\{ [2; 2]; [5; 6] \}$
8	*41*	$\{ [2; 3]; [5; 5]; [8; 8] \}$	$\{ [2; 2]; [5; 5] \}$

4.3 Optimizing List Merge

We may use the sweep line algorithm [40] to process the list merge. The time complexity of computing an active node is thus $O(|J_n| + |I_{n'}|)$, where $|\cdot|$ denotes the number of intervals in a list, opposed to the baseline algorithm's $O(|T|)$ time.

Due to the merge operation, $|J_n|$ is very small and far less than $|I_{n'}|$ in practice: in our experiment on the ALLIE dataset of two million medical terms, the average $|J|$ over 1,000 queries peaks to 5.4 at a query length of 6, but the average $|I|$ is up to 387 times of $|J|$. By regarding $|J_n|$ as a constant number, the time complexity becomes $O(|I_{n'}|)$. As we go deeper in the nested trie, the intervals in the stored lists become scattered and $|I_{n'}|$ increases. This incurs significant overhead for the merge operation.

We develop two techniques to optimize list merge. The first optimization is to speed up merge operations. For each interval $[u; v]$ in J_n, we use a binary search with u as the key to seek the first interval in $I_{n'}$ that may overlap $[u; v]$. The time complexity is $O(|J_n| \log |I_{n'}| + |J'_n|)$, and reduced to $O(\log |I_{n'}|)$ when $|J_n|$ is small. The second optimization is to reduce redundant merges by the following properties:

Proposition 2 (Properties of List Merge).
For any lists of intervals X, Y, and Z, $(X \otimes Y) \otimes Z = X \otimes (Y \otimes Z)$.
For any pair of outer nodes n and n', if n is an ancestor of n', then $I_n \otimes I_{n'} = I_{n'}$.
For any pair of nodes n and n' in an inner trie, if n is an ancestor of n', then $I_n \otimes I_{n'} = I_{n'}$.

By the three properties, if we follow a path $n; n_1; : : : ; n_k$ in the nested trie such that there is no shortcut in $n_1; : : : ; n_k$, then the result of list merge $J_n \otimes I_{n_1} : : : \otimes I_{n_k} = J_n \otimes I_{n_k}$; i.e., we may consider only the nodes at the two ends and skip the others. Thus, we delay the merge operation by pinning n and updating n' as the algorithm searches for active nodes, and invoke it only when n and n' are both right before shortcuts or n' is reached by the last keystroke. The MergeLists function in Line 4, Algorithm 3 is implemented using this optimization. The pseudo-code is given in Algorithm 4. If n and n' are not connected by a shortcut, we continue Algorithm 3 until a shortcut is encountered (Line 5). Then we record J_n in a temporary list J (Line 8), and continue Algorithm 3 again until we are about to move from n' to another node through a shortcut (Line 9). The list merge is computed afterwards using J and $I_{n'}$ (Line 10). Besides, the list merge is computed instantly whenever the last character of the query is processed

Algorithm 4: MergeLists ($J_n, I_{n'}$)

```
1  if q[i] is the last character of q then
2  |   return J_n ⊗ I_n'
3  else
4  |   if n and n' are not via a shortcut then
5  |   |   Continue Algorithm 3 until q[i] is the last character of
   |   |     q or n and n' are connected via a shortcut;
6  |   |   if q[i] is the last character of q then
7  |   |   |   return J_n ⊗ I_n'
8  |   J ← J_n;
9  |   Continue Algorithm 3 until q[i] is the last character of q or
   |     n' has a shortcut q[i + 1];
10 |   return J ⊗ I_n'
```

(Lines 2, 7, and 10). This optimization saves us most merge operations, as shown by this example:

Example 5. *Recall Example 4. The first shortcut is encountered from node 24 to 6. Before that, all the merge operations are skipped. We keep $J = J_{24} = I_{24}$ until reaching another shortcut from node 31 to 8. So we have $J_{31} = J \otimes I_{31}$. Then we keep $J = J_{31}$ until reaching node 41 by the last input character. $J_{41} = J \otimes I_{41}$. List merge is invoked only twice.*

Since list merge is skipped for some nodes in the above optimization, it is probable that $J_n \otimes I_{n'}$ becomes empty at some node but we fail to realize this. It does not cause false query results because the empty set can always be found whenever a merge operation is invoked, but it makes the optimization generate false active nodes and violate Lemma 1. It can be shown that as long as a shortcut occurs in the path to n', for the current and every subsequent keystroke, no matter what type of edge – outer, inner, or shortcut – we go, there always exists a case such that $J_n \otimes I_{n'} = \emptyset$. This suggests that in the worst case, we cannot retain Lemma 1 and at the same time skip any post-shortcut list merge or any equivalent/weaker operation (such as using Bloom filter [7]) for the empty-set check. Nonetheless, the case of producing false active nodes is rare for the above optimization. It significantly reduces query processing time because of saving many merge operations, and the number of active nodes is still much smaller than the baseline algorithm, as we will see in the experimental results reported in Section 7.3.

5 RANKING AND TOP-K RESULT FETCHING

5.1 Ranking for QACPA

Despite multiple ways to abbreviate a string in the input, some prefixes are preferred by users. Based on our analysis on the human-crafted prefix-abbreviations collected from Amazon Mechanical Turk, most users prefer to input `doc` when typing an abbreviation for `document`. This motivates us to rank results by the likelihood of being the intended string for the given input. Next we introduce the ranking method.

Given a data string s segmented into $[s_1, \dots, s_n]$, we suppose its first m keywords have been abbreviated in the query and the other $(n-m)$ keywords are yet to be input. Thus, $q \sqsubseteq s$, and q can be segmented into $[q_1, \dots, q_m]$ such that $q_i \preceq s_i$, $1 \le i \le m \le n$. For ease of exposition, we add $(n-m)$ empty strings, denoted by $q_{m+1}, \dots, q_n$, into the segmentation of q, so that q and s have the same number of segments.

The score of s is defined as the probability that s is the intended string for the query string q with respect to the segmentations $[q_1, \dots, q_n]$ and $[s_1, \dots, s_n]$, denoted by $score(s, q) = P(s_1 \dots s_n \mid q_1 \dots q_n)$. If there are multiple segmentations of q yielding the PA-match (e.g., `geet` PA-matches `GetEelTail` in two ways: [`ge`, `e`, `t`] and [`g`, `ee`, `t`]), we pick the one with the maximum score of all these segmentations. For all the data strings PA-matched by q, we rank them by decreasing order of scores.

To compute $score(s, q)$, by Bayes' theorem, we have

$$\begin{aligned} score(s, q) &= P(s_1 \dots s_n \mid q_1 \dots q_n) \\ &= \frac{P(q_1 \dots q_n \mid s_1 \dots s_n) \cdot P(s_1 \dots s_n)}{P(q_1 \dots q_n)} \\ &\propto P(q_1 \dots q_n \mid s_1 \dots s_n) \cdot P(s_1 \dots s_n) \\ &= P(q_1 \dots q_n \mid s_1 \dots s_n) \cdot P(s). \end{aligned}$$

The denominator $P(q_1 \dots q_n)$ is safely discarded because it is exactly $P(q)$, which is the same for all the PA-matched strings. $P(s)$ is measured by the popularity of s, in line with many traditional QAC methods. To compute $P(q_1 \dots q_n \mid s_1 \dots s_n)$, we assume that $P(q_i \mid s_i)$, $1 \le i \le n$, are independent [3]. Thus, $P(q_1 \dots q_n \mid s_1 \dots s_n) = P(q_1 \mid s_1) \cdot \dots \cdot P(q_n \mid s_n)$. We have

$$score(s, q) \propto P(q_1 \mid s_1) \cdot \dots \cdot P(q_n \mid s_n) \cdot P(s).$$

Each $P(q_i \mid s_i)$ is the probability that a user inputs q_i as the prefix of s_i. Specifically, we assume $P(q_i \mid s_i) = 1$ when $m < i \le n$. The reason is that these keywords are yet to be input. In order not to make the score of s too low due to the multiplication of a sequence, especially when $n \gg m$, we set these probabilities always equal to 1.

To evaluate $P(q_i \mid s_i)$, $1 \le i \le m$, we observe that users abbreviate s_i to q_i according to some patterns, such as cutting off at consonants. We choose to describe such patterns using vectors with the following features: (1) the length of q_i, (2) the number of vowels in q_i, (3) the number of consonants in q_i, (4) if q_i ends with a consonant, and (5) the value of i, i.e.,

[3] Despite the independence, the value of i, i.e., the position of the keyword in the string, plays a role in the probability. E.g., `Value` is more likely to abbreviated to `val` if it is the first keyword of a data string, but to `v` if it is not.

the position of s_i in the data string. A pattern is thus a 5-dimensional vector. Note that s_i is not fully encoded in the vector. The reason is explained: Let p_i denote the pattern (vector) by which a user abbreviate s_i to q_i. Because it tells how a keyword is abbreviated, $P(q_i, s_i) = P(p_i) \cdot P(s_i)$. Because $P(q_i, s_i) = P(q_i \mid s_i) \cdot P(s_i)$, $P(p_i)$ is exactly $P(q_i \mid s_i)$.

We assume that each pattern is determined by a mixture of a finite number of Gaussian distributions with unknown parameters. A Gaussian mixture model (GMM) is utilized to evaluate the probability (density function) of a pattern p:

$$P(p) = \sum_{i=1}^{l} w_i \mathcal{N}(p \mid \mu_i, \Sigma_i).$$

l is the number of Gaussian distributions. w_i is the weight of a component Gaussian distribution. $\mathcal{N}(p \mid \mu_i, \sigma_i)$ is the probability density function of p by a component Gaussian distribution with mean μ_i and covariance matrix Σ_i. l is tunable. The other parameters can be learned by a clustering with the expectation-maximization algorithm [13] over a set of training data generated as follows: A sample of data strings are given to users. Then we collect the prefixes input by the users, and convert each (keyword, prefix) pair to a feature vector as a training instance.

Example 6. *Consider the data strings in Table 1 and a query string* `genv`. *s^2, s^3, s^5, and s^6 are PA-matched strings. Suppose $k = 2$. Suppose the $P(q_i \mid s_i)$ values evaluated by the GMM are given in the table below.*

$(q_i \mid s_i)$	(ge | Gen)	(ge | Get)	(n | New)	(n | Null)
Prob.	*0.4*	*0.3*	*0.4*	*0.2*
$(q_i \mid s_i)$	(n | Next)	(v | Value)	(v | Vector)	
Prob.	*0.5*	*0.7*	*0.6*	

The following table shows the score computation and ranking of the PA-matched data strings. We use the notation P_i short for $P(q_i \mid s_i)$ in the table. $P(s)$ is measured by data string's popularity, which has been given in Table 1. The score (last column) is the product of the four preceding values. The top-k results are s^5 and s^6.

ID	***String***	$P(s)$	P_1	P_2	P_3	$score(s, q)$
s^2	`GenNewValue`	*0.1*	*0.4*	*0.4*	*0.7*	*0.0112*
s^3	`GenNullValue`	*0.3*	*0.4*	*0.2*	*0.7*	*0.0168*
s^5	`GetNextValue`	*0.6*	*0.3*	*0.5*	*0.7*	*0.063*
s^6	`GetNextVector`	*0.4*	*0.3*	*0.5*	*0.6*	*0.036*

5.2 Efficient Top-k Result Fetching

Recall in Algorithm 3, a list of merged intervals for each active node is obtained for result fetching. A naive approach to retrieving top-k results is to iterate through all the strings in these intervals and compute their scores. The major overhead of this procedure is invoking the GMM to compute the probability $P(q_i \mid s_i)$. Since the number of strings in the intervals might be large, especially for short queries, it is necessary to devise an efficient top-k algorithm to reduce the GMM computation. We propose two optimizations for this purpose.

The first optimization is to bound the maximum possible score for the strings in the merged list of intervals. Recall the merged list J_n and the stored list I_n at node n introduced in Section 4.2. We have the following property.

Proposition 3. *For any interval $[u, v] \in J_n$, there always exists an interval $[u', v'] \in I_n$, such that $u' \leq u$ and $v' \geq v$.*

It states that every interval in J_n is a sub-interval of one in I_n. Thus, the maximum possible scores of the strings in J_n are upper-bounded by those in I_n. To compute the score for each interval, we consider the root of the inner trie having n. Let d denote the depth of this root in the outer trie. It can be seen that all the data strings in I_n have at least d keywords, and when n becomes an active node, the query q has exactly d non-empty segments. Thus, for each interval $[u, v] \in I_n$, we may offline process the strings $s^u, \ldots, s^v$ and use the maximum to bound online queries. Given a string s^i, for each of its first d keywords, denoted by s^i_j ($1 \leq j \leq d$), we enumerate every possible prefix $\overleftarrow{s^i_j}$ of s^i_j and compute $P(\overleftarrow{s^i_j} \mid s^i_j)$. Note that when $j = d$, there is only one possible prefix because of the match at n. The product of the maximum $P(\overleftarrow{s^i_j} \mid s^i_j)$ values are multiplied by the popularity of s^i to obtain the maximum score of s^i amid all possible queries. We pick the maximum among $s^u, \ldots, s^v$ and store it along with $[u, v]$ in the trie.

Then we design an online top-k result fetching algorithm (Algorithm 5). It initializes a priority queue R for top-k results (Line 1). For each active node n, it sorts the intervals in J_n by decreasing order using the maximum scores stored at the intervals of I_n (Line 3). Then for each interval $[u, v]$ in J_n, we sequentially compute the scores of the strings in it and update the priority queue (Lines 7 – 9). If we reach an interval whose maximum score is no greater than the k-th result, the processing of n is safely terminated (Lines 5 – 6).

The second optimization is to skip online GMM computation, exploiting the observation that the strings in the same interval may share keywords and hence the same $P(q_i \mid s_i)$ values. For any two adjacent strings s^i and s^{i+1} in an interval $[u, v] \in I_n$, we offline check the number of keywords they share as prefix, and record this number at s^{i+1}, denoted by $s^{i+1}.spr$. Recall Example 4. For node 8, in the interval $[5, 6]$, since s^5 and s^6 share the first two keywords `Get` and `Next`, we store $s^6.spr = 2$. For online query processing, if both s^i and s^{i+1} appear in an interval in J_n, we are able to skip the GMM computation for the first $s^{i+1}.spr$ keywords of s^{i+1}, since they have just been computed. This optimization is integrated into Line 8 of Algorithm 5. To exploit the keyword sharing effectively, we sort the strings in S by the lexicographical order.

Algorithm 5: QACPA-Nested-Trie-TopK (q, A, k)

```
1  R ← ∅ ;                    /* a priority queue of size k */
2  foreach n ∈ A do
3      Sort the intervals in J_n using the maximum scores of I_n;
4      foreach [u; v] ∈ J_n do
5          if [u; v]:max_score ≤ R[k]:score then
6              break;
7          foreach i ∈ [u; v] do
8              if |R| < k or score(s^i; q) > R[k]:score then
9                  R:insert(s^i);
10 return R
```

Example 7. *Consider Example 6. Node 8 is the only active node.* J_8 *is* $\{[2; 3]; [5; 6]\}$. *Suppose the maximum* $P(q_i \mid s_i)$ *values for* `Gen`, `Get`, `New`, `Null`, *and* `Next` *are 0.4, 0.55, 0.45, 0.4, and 0.5, respectively. The* $P(q_i \mid s_i)$ *values for* `Value` *and* `Vector` *are given in Example 6, as they are the d-th keyword and have only one possible* $P(q_i \mid s_i)$. *The maximum score of* $[2; 3]$ *is* $\max(0:4 \times 0:45 \times 0:7 \times 0:1; 0:4 \times 0:4 \times 0:7 \times 0:3) = 0:0336$. *The maximum score of* $[5; 6]$ *is* $\max(0:55 \times 0:5 \times 0:7 \times 0:6; 0:55 \times 0:5 \times 0:6 \times 0:4) = 0:1155$. $[5; 6]$ *is scanned first due to larger maximum score.* $score(s^5; q) = 0:063$. *For* s^6, *because* $s^6:spr = 2$, *the GMM computation for the first two keywords is skipped.* $score(s^6; q) = 0:036$. *Because the maximum score of* $[2; 3]$ *is* $0:0336 < R[k]:score = 0:036$, *we terminate the processing of node 8.* s^5 *and* s^6 *are returned as top-k results.*

6 EXTENSIONS

We discuss a series of major extensions of our method. The extension to the case when keywords are manually separated in the input (traditional QAC) is straightforward and omitted.

6.1 Skipping Keywords

Users may skip a number of keywords in the middle, e.g., typing `geva` for `GetNextValue`, where `Next` is skipped. In this case, we modify our index as follows: For each node in the outer trie, we add outer shortcuts from the node to its descendants. For each node in the inner trie, we refer to the outer shortcuts resident on the root of the inner trie, and use bit vectors to indicate the difference, the same as the technique proposed in Section 3. For the ranking method, we use the GMM to evaluate the probability $P(q_i \mid s_i)$ for the skipped keywords, setting q_i as an empty string. This requires some keywords to be skipped in the training data of the GMM. Then we use the searching and ranking algorithms proposed in the previous sections to process queries.

6.2 Non-prefix Abbreviated Input

Users may abbreviate keywords by non-prefixes, e.g., typing `bldg` for `building`. Since most non-prefix abbreviations are composed of consonant letters, we focus on the following matching conditions: $q \sqsubseteq s$, if there exists a segmentation $[q_1; : : : ; q_m]$ of q, such that $8i \in [1 : : m]$, (1) q_i is a subsequence of the segment s_i of s, (2) $q_i[1] = s_i[1]$, and (3) among all the alignments in which q_i is a subsequence of s_i, there exists at least one alignment such that $8j \in [1 : : |q_i|]$, if $q_i[j]$ and $q_i[j+1]$ are aligned to $s_i[j']$ and $s_i[j' + \alpha]$ where $\alpha > 1$, then $q_i[j+1]$ must be a consonant letter. In short, the initial character of a segment must match, and the non-consecutive matching part consists of consonant letters only. Note that we are not limited to this setting but just use it to describe the extension. The index is modified as follows: For each node in the inner trie, we add inner shortcuts from the node to its descendants whose incoming edges are consonant letters. For ranking, we add non-prefix abbreviations in the training data. Then the proposed algorithms are used to process queries.

6.3 Full-text Search

Our method can be extended to support full-text search on data strings, e.g., typing `vage` for `GetNextValue`. This is an extension atop of the technique for skipping keywords, by allowing keywords to match order-insensitively. Recall that to handle skipping keywords, for each node n in the outer trie, we have outer shortcuts from n to its descendants. To make the match order-insensitive, we also add backward shortcuts from these descendants to n. The inner trie nodes can refer to these backward shortcuts. Then we run the proposed algorithms on this nested trie. Note that the list merge techniques (Section 4) are useful to prevent generating too many active nodes. In addition, when traversing the nested trie, we record the keywords that have been passed to avoid processing the same keyword twice along a path; e.g., for the query `vava`, when we have encountered the keyword `Value` in `GetNextValue`, `Value` will not be processed again for the second `va` in the query. Hence the algorithm can guarantee no false matches.

6.4 Updates in Data Strings

Insertion: When a new string whose ID is str_id is inserted, we add it into the nested trie with the method introduced in Section 3. Then for each node with one path from the root strictly matching a prefix of the new string, we add $[str_id; str_id]$ to its list of intervals. Deletion: When a string whose ID is str_id is deleted, we delete the nodes and the edges in the nested trie, if they are only used for this string. Then for each node having this underlying string, we delete str_id from its list of intervals. The above insertion and deletion may cause fragments in the lists of intervals if updates are frequent. In this case, we may record insertions in an auxiliary index which is also a

Table 4: Statistics of datasets.

Dataset	$\lvert S \rvert$	Max. $\lvert W \rvert$	Avg. $\lvert W \rvert$	Max. $\lvert s \rvert$	Avg. $\lvert s \rvert$	Size
JAVA	0.29 M	15	3.3	74	19.1	5.6 MB
PINYIN	3.55 M	14	5.1	59	17.2	61.7 MB
UNIX	1.68 M	39	3.3	71	13.4	23.1 MB
ALLIE	2.36 M	43	4.8	225	27.9	65.1 MB

nested trie, and mark deleted strings in the main index but do not remove any nodes or edges. The auxiliary index can be merged with the main index through an offline logarithmic merging [30]. We may also periodically reconstruct the index. This is similar to many information retrieval solutions.

7 EXPERIMENTS

We report the most important experimental results here. Please see Appendix B for the experiments on scalability, round-trip time, updates, index, and extensions.

7.1 Experiment Setup

In the experiments, we use the following datasets that cover the four applications listed in Section 1.

- **JAVA** is a dataset of API names of the Awesome Java Project on GitHub [47]. APIs are segmented by capital letters. Popularities are collected from the source codes of 12 projects [28].
- **PINYIN** is a dataset of Sougou cloud pinyin dictionary [43]. Words are segmented by Chinese syllables. We use the word frequencies in [44] as popularities.
- **UNIX** is a dataset of files in a UNIX archive [49] at ICM, Poland. Paths and extensions are removed. Filenames are segmented into keywords using Python WordSegment [23]. We use the term frequencies in the dataset as popularities.
- **ALLIE** is a dataset of terms extracted from MEDLINE [46]. Strings are segmented into morphemes using Morfessor [48]. We use the term frequencies in MEDLINE as popularities.

Table 4 shows dataset statistics, where $|S|$ denotes the number of distinct data strings, $|W|$ denotes the number of keywords in a data string, and $|s|$ denotes the string length.

We randomly selected 5,000 data strings from each dataset. The probability to choose a string is proportional to the popularity. Then we collected human-crafted prefix-abbreviations for the 5,000 strings from Amazon Mechanical Turk. There were on average 172 workers on each dataset. We asked them how they would like to input the query using abbreviations. 1,000 out of the 5,000 strings were randomly selected as queries. Since queries are usually short in QAC, we truncated them at the end of a prefix if the length of a query exceeds 8. The remaining 4,000 strings were used to train the GMM.

The following algorithms are compared:

- Trie is the trie-based baseline algorithm for QACPA.
- SegEnum is the algorithm that enumerates all query segmentations and matches keywords individually, followed by an intersection. We choose the TASTIER method [25, 26] to handle the keyword matching and intersection. It indexes keywords in a trie and finds the intersected results with a forward index.
- NT is the nested trie-based algorithm proposed in this paper.

The multi-dimensional substring search methods [16, 22] are not compared due to less efficiency than TASTIER in the intersection of string IDs for each segmentation. Moreover, we do not consider regular expression search [3, 33, 53] or subsequence search [2]. The reasons (equivalence to Trie/prohibitive index size) have been explained in Section 2.

For ranking methods, we compare with the most popular completion method (MPC) used in previous work [4, 41, 42]. It ranks results by popularity. Our proposed ranking method is referred to as APP (for **a**bbreviation **p**robability and **p**opularity). We use the following settings for l, the number of Gaussian distributions in the GMM, for the best overall quality: JAVA − 9, PINYIN − 3, UNIX − 3, ALLIE − 6. For this parameter setting, the best l tends to be small when keyword are short or prefixes are less diversified.

The experiments were run on a PC with an Intel Xeon E5-2637 (3.50GHz) CPU and 32GB RAM, running Ubuntu 14.04.5. The algorithms were implemented in C++ and in main memory.

7.2 Evaluation of Effectiveness

We first compare the keystrokes of QACPA with traditional QAC. Table 5 reports the results with navigation, i.e., the numbers of characters entered before the intended string appears in the top-k suggestions plus the numbers of arrow keys needed to navigate in the top-k list. Table 6 reports the results without navigation, i.e., only the numbers of input characters. The results are averaged over 1,000 queries when $k = 5$ or 10. We also show the percentage of keystrokes saved from traditional QAC, where users have to input a prefix of data string but cannot skip any characters in the middle. Compared to QAC, QACPA saves on average 19.4% and 21.6% keystrokes, with and without navigation, respectively. This indicates that using prefix-abbreviation remarkably reduces the effort of typing, in general from 8 keystrokes to 6 or 7 with navigation, and from 5 or 6 keystrokes to 4 without navigation. The advantage of QACPA is more significant on JAVA and ALLIE as they have more keywords in a data string. The saving is less significant on PINYIN due to the language factor: most users prefer to input the whole first syllable (as a keyword) in PINYIN, even with the QACPA feature. Nonetheless, QACPA still saves around 9–11% keystrokes on PINYIN.

Then we compare the ranking methods. We measure the mean reciprocal ranks (MRR) here and report the success

Table 5: Keystrokes per query (with navigation).

Dataset	k = 5		k = 10	
	QAC	QACPA	QAC	QACPA
JAVA	8.84	6.78 (-23.30%)	8.83	6.76 (-23.44%)
PINYIN	8.55	7.74 (-9.47%)	8.54	7.73 (-9.48%)
UNIX	8.78	7.13 (-18.79%)	8.73	7.10 (-18.67%)
ALLIE	8.76	6.47 (-26.15%)	8.70	6.45 (-25.89%)

Table 6: Keystrokes per query (without navigation).

Dataset	k = 5		k = 10	
	QAC	QACPA	QAC	QACPA
JAVA	6.36	4.67 (-36.33%)	5.49	4.24 (-29.33%)
PINYIN	6.40	5.74 (-11.58%)	5.83	5.31 (-9.84%)
UNIX	6.22	4.72 (-31.62%)	5.37	4.22 (-27.24%)
ALLIE	6.28	4.32 (-45.35%)	5.46	3.96 (-37.89%)

rates in Appendix B.1. MRR is defined as the average reciprocal of the intended string's ranking in the top-k suggestions (counted as 0 if not appearing). The statistical significance of the improvement is validated by paired t-tests ($p < 0{:}05$). We set $k = 5$ and 10, and vary the query length from 2 to 8. The results are reported in Table 7. We also show the relative MRR improvement over MPC. Note that even a small absolute difference in MRR could lead to considerable performance gain [41, 50]. APP, the proposed ranking method, is better than MPC for almost all settings. The relative improvement is more remarkable for short queries. The reason is that it is difficult to predict the intended string for short queries. A slight change in the suggestions may cause considerable difference. As the user types more keystrokes, the query becomes more predictable. Increasing MRRs are observed for both APP and MPC in this case, but APP still performs better than MPC. An exception is that both methods have zero MRR on PINYIN when $|q| = 2$. It is very rare to use queries as short as 2 in the collected prefixes for PINYIN. Both APP and MPC fail to return any meaningful results in this case. On the contrary, for the other query lengths on PINYIN, APP achieves the best improvement over MPC. This is because the abbreviations for PINYIN are less diversified and hence more predictable than the other three datasets which are mostly English.

To compare k settings, $k = 10$ saves more keystrokes and yields a better MRR, but $k = 5$ is also acceptable. Considering the numbers of available suggestions in different applications, we suggest using $k = 10$ for Web search and text editors, and $k = 5$ for mobile applications.

7.3 Evaluation of Efficiency

For efficiency, we first evaluate the searching phase of the query processing. We vary the query length and plot the average numbers of active nodes per query in Figures 4(a) – 4(d).

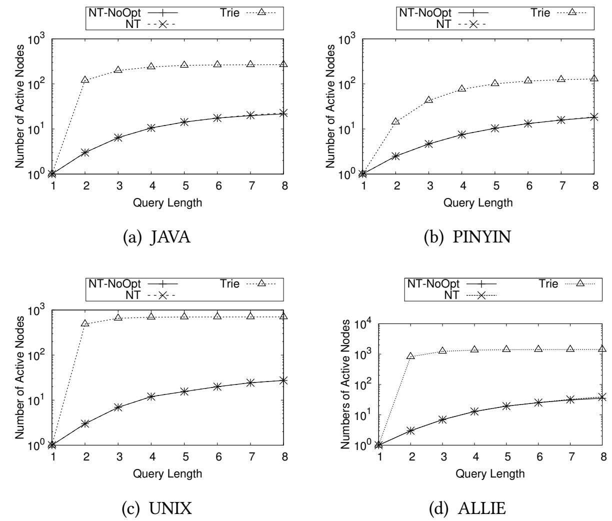


Figure 4: Number of active nodes.

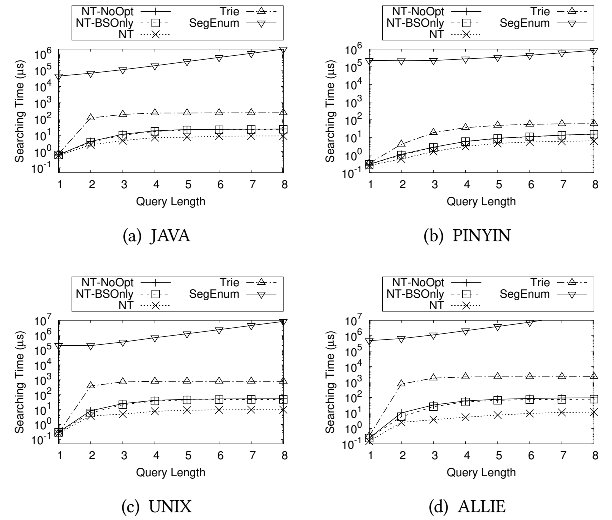


Figure 5: Searching time.

The results are accumulated, i.e., every active node en route is counted towards the number. So it increases with the query length. We also plot the results of our algorithm without the optimization of skipping list merge (Proposition 2), referred to as NT-BSOnly. It may produce fewer active nodes than NT since the optimization brings about false active nodes. It can be seen that Trie's number of active nodes surges at a query length of 2. This is expected as it has to search the nodes whose incoming edge is an initial character and matches the second character of the query. The number of such nodes is huge in the trie. NT drastically reduces the active node number and achieves a smooth increment w.r.t the query length. The gap

Table 7: Mean reciprocal rank (in percentage).

Dataset	k	$\|q\| = 2$		$\|q\| = 4$		$\|q\| = 6$		$\|q\| = 8$	
		APP	MPC	APP	MPC	APP	MPC	APP	MPC
JAVA	5	0.74 (+40.90%)	0.52	25.89 (+7.67%)	24.04	60.14 (+3.19%)	58.27	84.95 (+0.13%)	84.83
	10	1.04 (+29.39%)	0.80	27.27 (+5.35%)	25.88	61.10 (+3.27%)	59.16	85.04 (+0.13%)	84.92
PINYIN	5	0.00 (+0.00%)	0.00	4.06 (+49.01%)	2.72	40.37 (+18.34%)	34.11	73.59 (+6.07%)	69.37
	10	0.00 (+0.00%)	0.00	4.88 (+34.45%)	3.63	41.66 (+15.14%)	36.18	74.38 (+5.72%)	70.35
UNIX	5	0.90 (+29.86%)	0.69	15.50 (+5.07%)	14.75	44.10 (+1.93%)	43.26	72.11 (+0.60%)	71.68
	10	1.44 (+27.28%)	1.13	17.08 (+0.70%)	16.96	45.32 (+1.77%)	44.53	72.20 (+0.28%)	72.00
ALLIE	5	5.53 (+44.75%)	3.82	26.93 (+4.34%)	25.81	62.80 (+1.78%)	61.70	78.72 (+1.12%)	77.84
	10	6.45 (+39.11%)	4.64	28.19 (+4.16%)	27.06	62.84 (+0.74%)	62.37	78.93 (+1.26%)	77.94

between the two algorithms is more remarkable on UNIX and ALLIE due to the more diversity in the initial characters of keywords, which makes Trie even worse. When the query length is 2 on ALLIE, Trie produces 273 times more active nodes than NT. NT reports at most 35 active nodes at a query length of 8, and hence only 4.4 nodes per keystroke. Besides, the increase of active nodes caused by skipping list merge is not obvious. The number of false active nodes is small and only observed when the query length exceeds 5. NT reports at most 10% more active nodes than NT-BSOnly.

The times of the searching phase with varying query length are reported in Figures 5(a) – 5(d). To show the effect of the optimizations on list merge, we also plot the performance of NT without any optimization (referred to as NT-NoOpt) or with the binary search optimization only (referred to as NT-BSOnly). The result of SegEnum is also reported. NT is much faster than Trie. The gap is even larger than that in active nodes, because NT uses shortcuts to efficiently process the match for initial characters while Trie traverses subtrees. The maximum speedups over Trie on the four datasets are 44, 11, 144, and 486 times, respectively. SegEnum is the slowest of these competitors. The intersection of string IDs is very expensive and becomes even worse for longer queries. As for the optimizations on list merge, both techniques are useful, and skipping list merge saves more time than binary search.

We then evaluate the result fetching phase of the query processing. For Trie, each node in the trie is equipped with an interval to quickly identify the underlying strings. Then we scan the underlying strings of active nodes and compute the top-k strings by our ranking function. To study the effect of optimizating top-k computation, we also show the performance of NT without any optimization (referred to as NT-NoOpt) or with the maximum score optimization only (referred to as NT-MSOnly). The result fetching time of SegEnum is very close to Trie's because both compute scores for all the PA-matched strings, and thus it is not repeatedly shown. We set $k = 10$. Figures 6(a) – 6(d) show the result fetching times with varying query length. The times decrease for longer queries, because

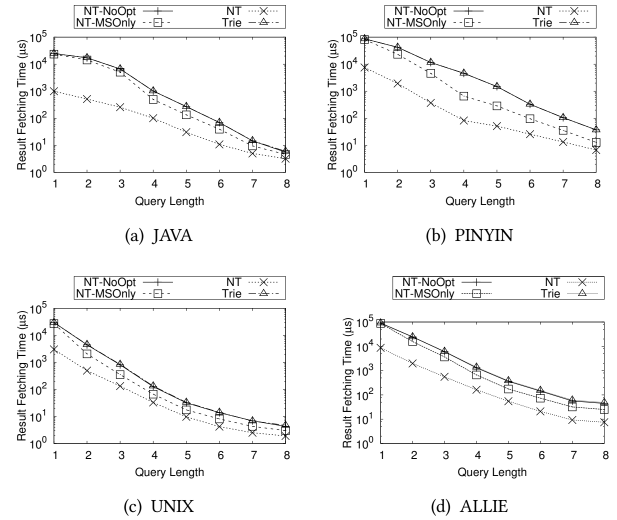
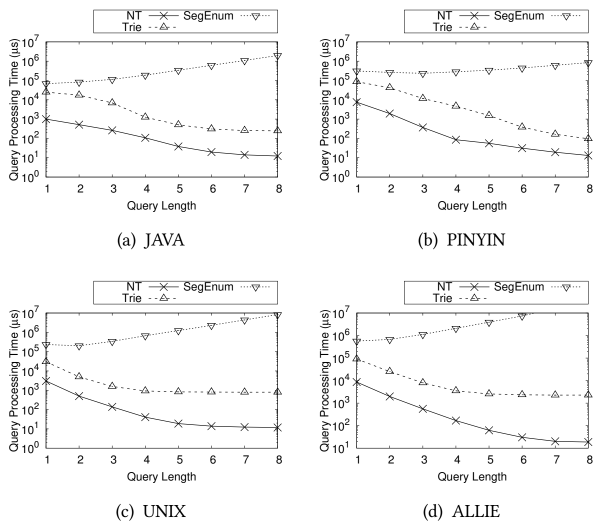


Figure 6: Result fetching time.

they are more selective and thus fewer data strings are computed for scores. Without any optimization, the speed of NT's result fetching is similar to Trie's. Both optimizations are effective, and sharing keywords is more useful than materializing maximum scores. As a result, NT outperforms Trie by up to 33, 56, 10, and 12 times on the four datasets, respectively. The gap is more significant on PINYIN for its less diversified spelling from which more keyword sharing can be exploited. For the result fetching times w.r.t. k, we refer readers to Appendix B.2.

The overall query processing times ($k = 10$) are plotted in Figures 7(a) – 7(d). The overall trend is the query processing time decreases with the query length for Trie and NT but increases for SegEnum. The reason is the proportions of searching and result fetching are different; e.g., for short queries, Trie and NT spend more time on searching but for long queries, they spend more time on result fetching. SegEnum is the slowest for its expensive enumeration of segmentations. NT is always the fastest. The overall speedup over the runner-up, Trie, is up to 33, 54, 67, and 121 times on the four



(a) JAVA (b) PINYIN (c) UNIX (d) ALLIE

Figure 7: Overall query processing time.

datasets, respectively. Another interesting observation is that Trie regularly spends tens (and up to hundreds) of milliseconds processing a query, which is too long for online editors and search engines. The time is reduced by NT to milliseconds and even less, showing the benefit of improving efficiency.

8 RELATED WORK

Due to its important applications, especially for Web search engines, the research on QAC has received much attention in the last few decades. We refer readers to two surveys for various kinds of QAC [8, 24]. Early studies considered completing the query at word [5] or phrase level [18, 32]. Fan *et al.* [15] studied the suggestion on topic-based query terms. Bhatia *et al.* [6] investigated the case when query logs are absent. Recent trends feature a boom in context-aware QAC where user interactions are important [4, 31], as well as plenty of work on presenting time-sensitive [42, 50], personalized [41], or diversified results [9]. As for the quality of QAC, an experimental evaluation was reported in [38] to compare various ranking methods. In the database research community, Li *et al.* designed the TASTIER system for type-ahead search on relational data [25]. It employs a two-tier trie, but the second-tier tries only reside on the leaf nodes of the first-tier trie, as opposed to our nested-trie in which inner tries may reside on the internal nodes of the outer trie. Some effort was dedicated to error-tolerant QAC or fuzzy type-ahead search to allow errors in the input, using edit distance constraints [12, 26, 52] or Markov n-gram transformation model [14]. Cetindil *et al.* [11] proposed a ranking method for error-tolerant QAC using proximity information. Another popular direction is location-aware QAC [21, 36, 54], which is useful for navigation tools. Another body of work aims at query reformulation by taking a full query and making arbitrary modifications to assist users [10, 20, 37]. In some studies, query autocompletion and reformulation are both called query suggestion.

The tolerant retrieval problem [30] has been studied for decades. An important query model is wildcard query, in which a user may use a Kleene star to denote any number of characters. A prevalent approach is to use permuterm index [17]. Another related problem is multi-dimensional substring search [16, 22]. Both data objects and queries are tuples consisting of d strings, regarded as d dimensions. The goal is to find data objects such that on each dimension, the query string is a substring of the query string. The difference from our problem is that Kleene stars or dimensions are not explicitly given in QACPA. Other related problems include subsequence search [2] and regular expression search [3, 33, 53].

QACPA can be categorized as processing queries involving abbreviations. Related work comes from human-computer interaction [35, 51] and database research community [45]. In addition, there are studies targeting transformation rules [1] or synonyms [29] in the database area.

9 CONCLUSION

We proposed a new feature of query autocompletion which takes prefix-abbreviated keywords as input. Compared to traditional query autocompletion, the new feature supports more application scenarios, especially for those in which users may not explicitly specify delimiters of keywords. We first analyzed the inefficiencies of the trie-based algorithm, which was adopted by traditional query autocompletion, as well as a few other possibilities, and then developed an efficient indexing and query processing method to handle the new autocompletion feature. To return meaningful results, we devised a ranking method specific to the proposed autocompletion paradigm and an efficient top-k result fetching algorithm. A series of useful extensions were discussed. Experiments on real datasets showed the effectiveness of the new type of query autocompletion and the superiority of the query processing method over alternative solutions in terms of efficiency.

ACKNOWLEDGMENTS

This work was supported by JSPS Kakenhi Grant No. 16H01722 and MIC SCOPE Grant No. 172307001.

REFERENCES

[1] A. Arasu, S. Chaudhuri, and R. Kaushik. Transformation-based framework for record matching. In *ICDE*, pages 40–49, 2008.
[2] R. A. Baeza-Yates. Searching subsequences. *Theor. Comput. Sci.*, 78(2):363–376, 1991.
[3] R. A. Baeza-Yates and G. H. Gonnet. Fast text searching for regular expressions or automaton searching on tries. *J. ACM*, 43(6):915–936,

1996.
[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] S. Bhatia, D. Majumdar, and P. Mitra. Query suggestions in the absence of query logs. In *SIGIR*, pages 795–804, 2011.
[7] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.
[8] F. Cai and M. de Rijke. A survey of query auto completion in information retrieval. *Foundations and Trends in Information Retrieval*, 10(4):273–363, 2016.
[9] F. Cai, R. Reinanda, and M. de Rijke. Diversifying query auto-completion. *ACM Trans. Inf. Syst.*, 34(4):25:1–25:33, 2016.
[10] H. Cao, D. Jiang, J. Pei, Q. He, Z. Liao, E. Chen, and H. Li. Context-aware query suggestion by mining click-through and session data. In *KDD*, pages 875–883, 2008.
[11] I. Cetindil, J. Esmaelnezhad, T. Kim, and C. Li. Efficient instant-fuzzy search with proximity ranking. In *ICDE*, pages 328–339, 2014.
[12] S. Chaudhuri and R. Kaushik. Extending autocompletion to tolerate errors. In *SIGMOD*, pages 707–718, 2009.
[13] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society, Series B*, 39(1):1–38, 1977.
[14] H. Duan and B. P. Hsu. Online spelling correction for query completion. In *WWW*, pages 117–126, 2011.
[15] J. Fan, H. Wu, G. Li, and L. Zhou. Suggesting topic-based query terms as you type. In *APWeb*, pages 61–67, 2010.
[16] P. Ferragina, N. Koudas, S. Muthukrishnan, and D. Srivastava. Two-dimensional substring indexing. In *PODS*, 2001.
[17] P. Ferragina and R. Venturini. The compressed permuterm index. *ACM Trans. Algorithms*, 7(1):10:1–10:21, 2010.
[18] K. Grabski and T. Scheffer. Sentence completion. In *SIGIR*, pages 433–439, 2004.
[19] S. Han, D. R. Wallace, and R. C. Miller. Code completion of multiple keywords from abbreviated input. *Autom. Softw. Eng.*, 18(3-4):363–398, 2011.
[20] Q. He, D. Jiang, Z. Liao, S. C. H. Hoi, K. Chang, E. Lim, and H. Li. Web query recommendation via sequential query prediction. In *ICDE*, pages 1443–1454, 2009.
[21] S. Hu, C. Xiao, and Y. Ishikawa. An efficient algorithm for location-aware query autocompletion. *IEICE Transactions*, 101-D(1):181–192, 2018.
[22] H. V. Jagadish, N. Koudas, and D. Srivastava. On effective multi-dimensional indexing for strings. In *SIGMOD*, pages 403–414, 2000.
[23] G. Jenks. Python WordSegment. http://www.grantjenks.com/docs/wordsegment/, 2018.
[24] U. Krishnan, A. Moffat, and J. Zobel. A taxonomy of query auto completion modes. In *ADCS*, pages 6:1–6:8, 2017.
[25] G. Li, S. Ji, C. Li, and J. Feng. Efficient type-ahead search on relational data: a TASTIER approach. In *SIGMOD*, pages 695–706, 2009.
[26] G. Li, S. Ji, C. Li, and J. Feng. Efficient fuzzy full-text type-ahead search. *VLDB J.*, 20(4):617–640, 2011.
[27] G. Li, J. Wang, C. Li, and J. Feng. Supporting efficient top-k queries in type-ahead search. In *SIGIR*, pages 355–364, 2012.
[28] G. Little and R. C. Miller. Keyword programming in java. *Autom. Softw. Eng.*, 16(1):37–71, 2009.
[29] J. Lu, C. Lin, W. Wang, C. Li, and X. Xiao. Boosting the quality of approximate string matching by synonyms. *ACM Trans. Database Syst.*, 40(3):15:1–15:42, 2015.
[30] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval.* Cambridge University Press, 2008.
[31] B. Mitra, M. Shokouhi, F. Radlinski, and K. Hofmann. On user interactions with query auto-completion. In *SIGIR*, pages 1055–1058, 2014.
[32] A. Nandi and H. V. Jagadish. Effective phrase prediction. In *VLDB*, pages 219–230, 2007.
[33] G. Navarro. Nr-grep: a fast and flexible pattern-matching tool. *Softw., Pract. Exper.*, 31(13):1265–1312, 2001.
[34] People's Daily Online. Sogou's revenue increased 53% in the first quarter of this year, revenues boosted beyond expectations by AI technology (in Chinese). http://it.people.com.cn/n1/2018/0426/c1009-29951829.html, Apr. 26, 2018.
[35] S. Pini, S. Han, and D. R. Wallace. Text entry for mobile devices using ad-hoc abbreviation. In *AVI*, pages 181–188, 2010.
[36] S. B. Roy and K. Chakrabarti. Location-aware type ahead search on spatial databases: semantics and efficiency. In *SIGMOD*, pages 361–372, 2011.
[37] E. Sadikov, J. Madhavan, L. Wang, and A. Y. Halevy. Clustering query refinements by user intent. In *WWW*, pages 841–850, 2010.
[38] G. D. Santo, R. McCreadie, C. Macdonald, and I. Ounis. Comparing approaches for query autocompletion. In *SIGIR*, pages 775–778, 2015.
[39] M. Sevenster, R. C. van Ommering, and Y. Qian. Algorithmic and user study of an autocompletion algorithm on a large medical vocabulary. *Journal of Biomedical Informatics*, 45(1):107–119, 2012.
[40] M. I. Shamos and D. Hoey. Geometric intersection problems.
[41] M. Shokouhi. Learning to personalize query auto-completion. In *SIGIR*, pages 103–112, 2013.
[42] M. Shokouhi and K. Radinsky. Time-sensitive query auto-completion. In *SIGIR*, pages 601–610, 2012.
[43] Sougou Labs. Sougou Pinyin Dictionary. https://pinyin.sogou.com/dict/, 2006.
[44] Sougou Labs. Sougou Pinyin Dictionary. http://www.sogou.com/labs/resource/w.php, 2006.
[45] W. Tao, D. Deng, and M. Stonebraker. Approximate string joins with abbreviations. *PVLDB*, 11(1):53–65, 2017.
[46] A. D. Team. Allie RDF Data. http://data.allie.dbcls.jp/index_en.html/, 2018.
[47] The Awesome Java Contributors. Awesome Java frameworks, libraries and software. https://github.com/akullpp/awesome-java, 2018.
[48] S. Virpioja, P. Smit, and S.-A. Grönroos. Morfessor. http://morfessor.readthedocs.io/, 2018.
[49] Warren Toomey. The Unix Archive. https://wiki.tuhs.org/doku.php?id=source:unix_archive, 2018.
[50] S. Whiting and J. M. Jose. Recent and robust query auto-completion. In *WWW*, pages 971–982, 2014.
[51] T. Willis, H. Pain, S. Trewin, and S. Clark. Informing flexible abbreviation expansion for users with motor disabilities. In *ICCHP*, pages 251–258, 2002.
[52] C. Xiao, J. Qin, W. Wang, Y. Ishikawa, K. Tsuda, and K. Sadakane. Efficient error-tolerant query autocompletion. *PVLDB*, 6(6):373–384, 2013.
[53] X. Yang, T. Qiu, B. Wang, B. Zheng, Y. Wang, and C. Li. Negative factor: Improving regular-expression matching in strings. *ACM Trans. Database Syst.*, 40(4):25:1–25:46, 2016.
[54] R. Zhong, J. Fan, G. Li, K. Tan, and L. Zhou. Location-aware instant search. In *CIKM*, pages 385–394, 2012.

A PROOFS

A.1 Proposition 1

Proof. We construct a string t by concatenating the label of the edge or shortcut between n_i and n_{i+1} for $i \in [1 :: k-1]$. By Algorithm 2, $|q| = k-1$ and $t[i]$ matches $q[i]$, $8i \in [1 :: |q|]$.

Suppose t is segmented into $[t_1, \ldots, t_l]$ by the initial characters along the path $n_1, \ldots, n_k$. Given any string s (suppose it is segmented into $[s_1, \ldots, s_m]$) in $I_{n_1} \otimes I_{n_2} \ldots \otimes I_{n_k}$, by the construction of the nested trie and the definition of the stored list of intervals, for any node n_i in $\{n_2, \ldots, n_k\}$, a prefix of s strictly matches exactly one path from n_1 to n_i. Therefore, $l \leq m$, and $\forall i \in [1 \ldots l]$, $t_i[1] = s_i[1]$; and $\forall i \in [1 \ldots l]$, we have $\forall j \in [2 \ldots |t_i|]$, $t_i[j] = s_i[j]$. Therefore, $\forall i \in [1 \ldots l]$, $t_i \preceq s_i$.

Because $\forall i \in [1 \ldots |q|]$, $t[i]$ matches $q[i]$, q can be segmented into $[q_1, \ldots, q_l]$ by the initial characters in t, and $\forall i \in [1 \ldots l]$, $q_i \preceq s_i$. Therefore, $q = \overleftarrow{s_1}\,\overleftarrow{s_2} \ldots \overleftarrow{s_l}$, i.e., q PA-matches s. □

A.2 Lemma 1

Proof. We create a map f from the nodes in the trie of S to the nodes in the nested trie of S: For any node n in the trie, it is mapped to a node n' in the nested trie, such that the path from the root of the trie to n strictly matches one path from the root of the nested trie to n'. Because a path from the root of the trie is exactly a prefix of a data string, by the definition of the nested trie, each data string is strictly matched by at most one path in the nested trie. Therefore f is surjective. Suppose we are processing a character of q, and n' becomes an active node by Algorithm 2. By Proposition 1, there exists at least one underlying string of n PA-matched by q. Without loss of generality, we suppose there is only one such string, denoted by s. By the definition of stored list of intervals, a prefix of s matches one path from the root to n'. This prefix also matches a path from the root of the trie, because otherwise Algorithm 1 misses s as a result. Therefore the end of this path is an active node in Algorithm 1. By the definition of f, f maps this node to n'. Because f is surjective, the number of active nodes by Algorithm 2 is at most equal to that by Algorithm 1. □

A.3 Proposition 2

Proof. Property 1: Let $X = \{x_1, \ldots, x_m\}$, $Y = \{y_1, \ldots, y_n\}$, and $Z = \{z_1, \ldots, z_o\}$. By the definition of the $\otimes$ operation, $(X \otimes Y) \otimes Z = \{x_i \cap y_j \cap z_k \mid 1 \leq i \leq m \wedge 1 \leq j \leq n \wedge 1 \leq k \leq o \wedge x_i \cap y_j \cap z_k \neq \emptyset\}$. Likewise, $X \otimes (Y \otimes Z) = \{x_i \cap y_j \cap z_k \mid 1 \leq i \leq m \wedge 1 \leq j \leq n \wedge 1 \leq k \leq o \wedge x_i \cap y_j \cap z_k \neq \emptyset\}$. Therefore, $(X \otimes Y) \otimes Z = X \otimes (Y \otimes Z)$.

Property 2: By the definition of the stored list of intervals, for any string $s \in I_{n'}$, s strictly matches one path from the root of the nested trie to n'. Because n is an ancestor of n' in the outer trie, this path must pass n. Therefore, $s \in I_n$, and hence $I_n \otimes I_{n'} = I_{n'}$. Property 3 can be proved in the same way. □

B ADDITIONAL EXPERIMENTS

B.1 Success Rates

Table 8 reports the success rates @k (i.e., whether the intended string appears in the top-k results) of APP and MPC for top-1, top-2, and top-3 suggestions. We also show the relative improvement of APP over MPC. Similar to the results of MRR (Table 7), APP performs better than MPC for almost all settings, and the relative improvement is more remarkable when queries are short. A query length of 2 is a hard case for both methods, as it is almost impossible to predict the intended string given the very short input. When the query length is 4, APP exhibits substantial and meaningful improvement over MPC on JAVA, UNIX, and ALLIE. When the query length reaches 6 or 8, the success rates of the two methods become close on the three datasets, but APP still performs better. On PINYIN, because of the less diversified spelling, only long queries ($|q| \geq 6$) are predictable. APP delivers much higher success rates than MPC in this case.

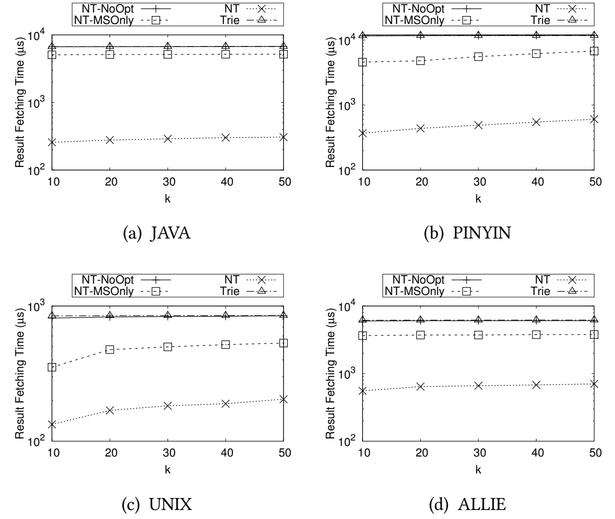


Figure 8: Result fetching time w.r.t. k.

B.2 Result Fetching w.r.t. k

Figures 8(c) – 8(d) show the result fetching times w.r.t. k. The query length is 3. For Trie, we witness approximately constant time, as it has to scan all the underlying strings of the active nodes. For NT, we witness moderate increase (about 1.5 times) when k moves from 10 to 50. The superiority over Trie remains.

B.3 Scalability

We evaluate the scalabilities by varying dataset size. 20% to 100% data strings were sampled from the four datasets. The query processing times when $|q| = 3$ are given in Figures 9(c) – 9(d). SegEnum is not plotted for its slow speed. An approximately linear growth w.r.t. the dataset size is observed for both algorithms. NT has a slower growth rate than Trie.

Table 8: Success rate (in percentage).

Dataset	k	jqj = 2		jqj = 4		jqj = 6		jqj = 8	
		APP	MPC	APP	MPC	APP	MPC	APP	MPC
JAVA	1	0.40 (+100%)	0.20	12.80 (+0%)	12.80	46.00 (+5.02%)	43.80	81.70 (+0.12%)	81.60
	2	0.40 (+100%)	0.20	26.90 (+10.24%)	24.40	68.30 (+3.01%)	66.30	87.20 (+0.23%)	87.00
	3	0.80 (+33.33%)	0.60	36.50 (+13.00%)	32.30	73.90 (+0.95%)	73.20	88.50 (+0%)	88.50
PINYIN	1	0 (+00%)	0.00	1.70 (+∞%)	0.00	25.00 (+20.19%)	20.80	63.00 (+10.72%)	56.90
	2	0 (+00%)	0.00	3.60 (+44.00%)	2.50	38.70 (+11.85%)	34.60	71.30 (+3.18%)	69.10
	3	0 (+00%)	0.00	4.90 (+28.94%)	3.80	47.40 (+7.48%)	44.10	77.40 (+2.38%)	75.60
UNIX	1	0.70 (+16.66%)	0.60	7.20 (+9.09%)	6.60	29.80 (+1.36%)	29.40	65.10 (+0.30%)	64.90
	2	0.80 (+14.28%)	0.70	14.10 (+0.00%)	14.10	50.20 (+2.86%)	48.80	75.40 (+0.26%)	75.20
	3	1.40 (+75.00%)	0.80	23.00 (+4.07%)	22.10	58.40 (+1.38%)	57.60	79.40 (+0.76%)	78.80
ALLIE	1	2.90 (+163.63%)	1.10	17.00 (+0%)	17.00	53.60 (+2.87%)	52.10	70.30 (+0.71%)	69.80
	2	4.50 (+36.36%)	3.30	30.00 (+7.52%)	27.90	69.50 (+0.14%)	69.40	84.90 (+3.28%)	82.20
	3	7.50 (+19.04%)	6.30	35.70 (+4.69%)	34.10	77.40 (+0.25%)	77.20	88.00 (+0.80%)	87.30

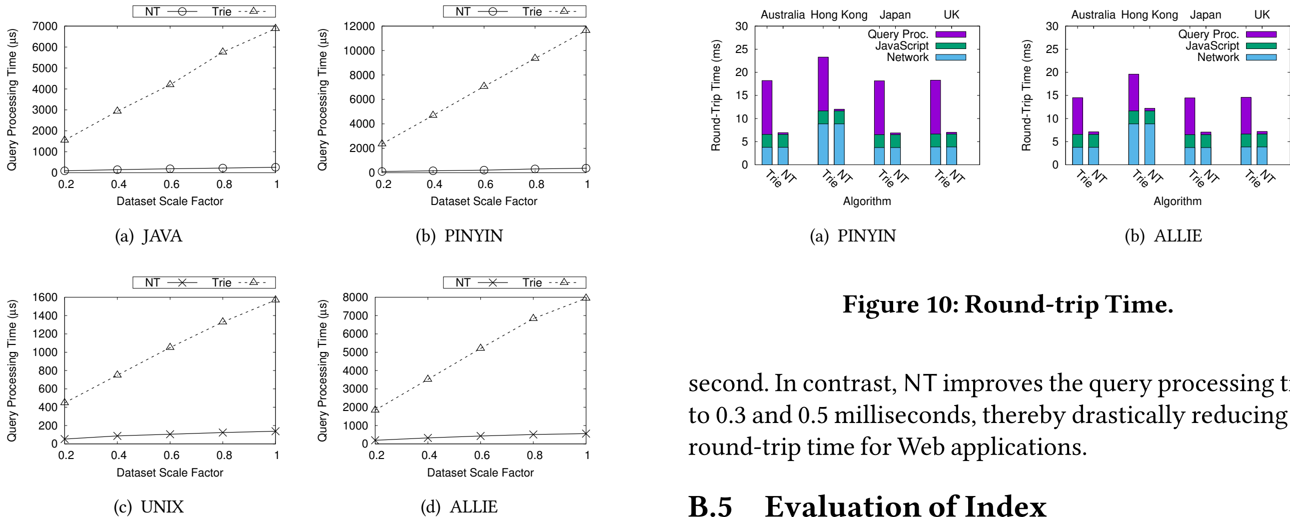


Figure 9: Scalability.

Figure 10: Round-trip Time.

B.4 Round-trip time

We evaluate the round-trip time in a Web server setting. The round-trip time is composed of three parts: network delay, JavaScript front-end, and back-end query processing. Connections to our server were launched from Australia, Hong Kong, Japan, and UK. PINYIN and ALLIE were chosen for this set of experiments since they are used in Web applications (cloud IME and search engine, respectively). The query length is 3. The decomposed round-trip times averaged over 1,000 queries are plotted in Figures 10(a) – 10(b). We observe that Trie consumes 11.6 and 7.9 milliseconds in query processing, taking around 60% and 51% round-trip time on the two datasets, respectively. This means query processing is the bottleneck if we use Trie. When there are 20 simultaneous connections, its query processing exceeds the acceptable response time of 0.1 second. In contrast, NT improves the query processing time to 0.3 and 0.5 milliseconds, thereby drastically reducing the round-trip time for Web applications.

B.5 Evaluation of Index

Table 9 reports index sizes and construction times. For Trie, the index includes the trie ($|T|$ nodes) and the intervals for result fetching. For SegEnum, it includes the trie, the intervals for keyword fetching, and the forward index. For NT, it includes the nested trie (worst-case $|T|$ nodes plus $(|T| - 1)$ shortcuts), the lists of intervals (worst-case $(|T| \cdot d_{\max})$ intervals, where $d_{\max}$ is the maximum depth in the trie), and the shared numbers of prefix keywords (worst-case $(|T| - 2)$ numbers) for adjacent strings. We use the radix tree to compress all of them. SegEnum has the smallest index size among the three algorithms because its trie is built on keywords rather than data strings. NT's index size is moderately larger than Trie because of keeping additional information in the index such as the lists of intervals. The construction time includes building the index and computing offline information such as maximum scores and shared numbers of keywords. All the three algorithms are acceptable in index construction speed. Trie is the fastest. NT and SegEnum are slower due to the construction of additional data structure.

Table 9: Index size and construction time.

Dataset	Size (MB)			Time (s)		
	Trie	SegEnum	NT	Trie	SegEnum	NT
JAVA	44	11	62	0.80	2.09	6.37
PINYIN	613	161	680	8.89	12.30	38.11
UNIX	275	96	461	3.37	13.03	12.58
ALLIE	359	118	685	7.00	88.81	58.38

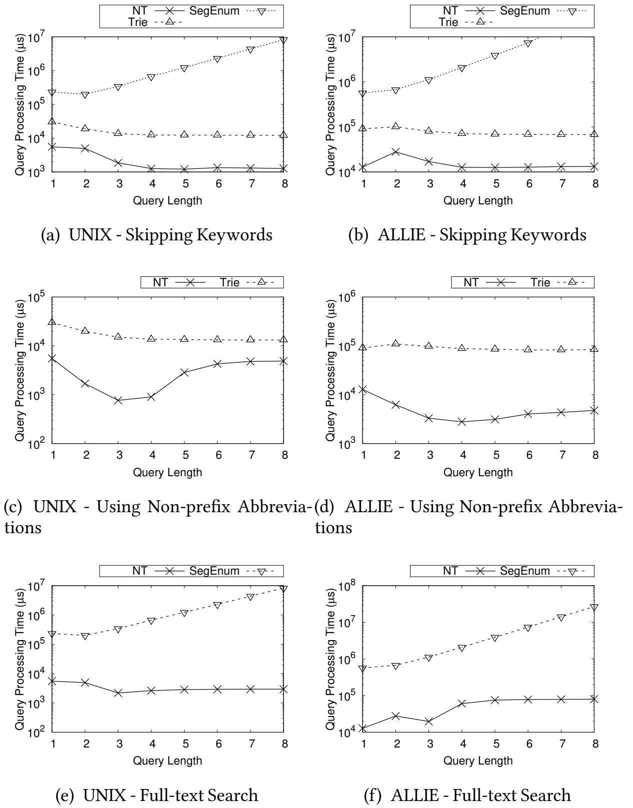
B.6 Extensions

We evaluate major extensions of QACPA – skipping keywords, using non-prefix abbreviations, and full-text search – on UNIX and ALLIE, which we believe are more suitable for such extensions from the application perspective. For skipping keywords and using non-prefix abbreviations, queries and training examples were generated using the method described in Section 7.1, except that users may skip keywords or use non-prefix abbreviations. For full-text search, we reused the queries for skipping keywords but randomly changed the order of keywords' prefixes.

We evaluate the effectiveness for the first two extensions. Compared to standard QACPA, a slight decrease is observed in keystroke saving and MRR for both extensions. E.g., for the top-5 results on ALLIE, on average we need to input 0.33 more keystrokes than standard QACPA if keywords are skipped and 0.26 more keystrokes than standard QACPA if queries are abbreviated by non-prefixes. This is because some strings become top-k results by the semantics of the two extensions but they are not intended strings.

Compared to standard QACPA, the index sizes of NT for extensions are larger. On UNIX, the size increases by 2.2, 6.0, and 4.2 times for the three extensions, respectively. On ALLIE, it increases by 2.6, 6.3, and 4.7 times, respectively.

We adapt Trie and SegEnum for skipping keywords, Trie for non-prefixes, and SegEnum for full-text search. The query processing times are shown in Figures 11(a) – 11(f). Compared to standard QACPA, both Trie and NT spend more time processing queries, while SegEnum has almost the same performance. This is expected, because compared to the algorithms for standard QACPA, Trie and NT need to traverse more nodes to seek results, but SegEnum only differs in the keyword order check, which is the final step after the list merge. Nonetheless, NT is still significantly faster than Trie and SegEnum. When skipping keywords, the speedup over the runner-up, Trie, is up to 10 times on UNIX and 7 times on ALLIE. When using non-prefix abbreviations, the speedup is up to 20 and 31 times, respectively. For full-text search, despite utilizing techniques tailored to full-text search, SegEnum suffers from the enumeration of segmentations due to the absence of delimiters in the input. NT is faster than SegEnum by one to three orders of magnitude.



(a) UNIX - Skipping Keywords (b) ALLIE - Skipping Keywords

(c) UNIX - Using Non-prefix Abbreviations (d) ALLIE - Using Non-prefix Abbreviations

(e) UNIX - Full-text Search (f) ALLIE - Full-text Search

Figure 11: Query processing time for extensions.

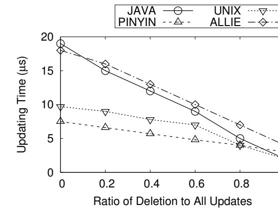


Figure 12: Update processing time.

B.7 Updates

We evaluate the performance of processing update. 1,000 strings were randomly generated for insertions and 1,000 strings were randomly sampled from the datasets for deletions. We use the techniques introduced in Section 6.4 for the single index setting (i.e., no auxiliary index). Figure 12 shows the processing times averaged over 1,000 updates by varying the percentage of deletions from 0% to 100%. The processing times are mostly in microseconds per update, and the general trend is that they decrease when we have more deletions on the four datasets. This is expected, because for most deletion queries we only need to delete the string ID from the lists of intervals for the nodes having this underlying string.