Efficient Text Autocompletion for Online Services

HU Sheng

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.

However, there are many challenges laid on various applications other than Web search engines. Thus, we focus on the solutions of *Autocompletion for Online Services*.

First, we investigate *location-aware query autocompletion*. As mobile devices become more and more popular, one of the main applications is location-aware service, such as Web mapping. In this work, we propose a new solution to location-aware query autocompletion. We devise a trie-based index structure and integrate spatial information into trie nodes. Our method is able to answer both range and top-k queries. In addition, we discuss the extension of our method to support the error tolerant feature in case user's queries contain typographical errors. Experiments on real datasets show that the proposed method outperforms existing methods in terms of query processing performance.

Second, we study a novel QAC paradigm. For existing QAC methods, users have to manually type delimiters to separate keywords in their inputs. In this work, 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, input method editors, as well as the tasks of searching long proper names comprising multiple morphemes. 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 on real datasets demonstrate the effectiveness of the new QAC paradigm and the efficiency of the proposed query processing method.

Finally, we explore the problem of code completion, which is a traditional popular feature for API access in integrated development environments (IDEs). It not only frees programmers from remembering specific details about an API but also saves keystrokes and corrects

typographical errors. Existing methods for code completion usually suggest APIs based on statistics in code bases described by language models. However, they neglect the fact that the user's input is also very useful for ranking, as the underlying patterns can be used to improve the accuracy of predictions of intended APIs.

To improve users' satisfactions, we propose a novel method to improve the quality of code completion by incorporating the users' acronym-like input conventions and the APIs' scope context into a discriminative model. The users' input conventions are learned using a logistic regression model by extracting features from collected training data. The weights in the discriminative model are learned using a support vector machine (SVM). To improve the real-time efficiency of code completion, we employ a trie to index and store the scope context information. An efficient top-*k* algorithm is developed. Experiments show that our proposed method outperforms the baseline methods in terms of both effectiveness and efficiency.

Generally, an overall view of autocompletion across different application domains is provided. We believe that our contributions are practical and easy to be applied in many other applications. At first, we think our efficient index design and early termination pruning techniques can be applied in either geo-graphical or textual databases. Secondly, our proposed ranking method and novel autocompletion paradigm can practically improve the top-*k* accuracy and effective performance in textual editors, mapping services and any other Web retrieval systems. Last but not least, extensive experimental evaluations are conducted to illustrate the performance on real-world datasets.

Acknowledgements

First of all, I would like to gratefully acknowledge the guidance of my joint supervisors, Prof. Yoshiharu Ishikawa and Assistant Prof. Chuan Xiao. Their patience, motivation and immense knowledge assisted me throughout my academic studies. Their strong insight and broad knowledge will continuously benefit me in my future life.

I would also like to thank the rest of my thesis committee: Prof. Kenji Mase, Prof. Yousuke Watanabe, and my mentor professor Prof. Naoki Nishida for their careful reading, insightful comments, and fruitful discussions.

My special thanks go to Dr. Jianquan Liu and Dr. Shoji Nishimura, for their generous help when I started my internship in NEC Japan.

I am grateful to the professors and staff members from RWDC program for helping me to build my career vision and increase my leadership experience. I have learned many inter-disciplinary skills from them.

I would like to thank the members of our laboratory at Nagoya University, for their help in many different ways.

At last, I would like to thank my parents for their love and various support during my Ph.D study.

Contents

List of Figures

List of Tables

Abbreviations

QAC	**Q**uery **A**uto**c**ompletion
POI	**P**oint **O**f **I**nterest
GMM	**G**aussian **M**ixtual **M**odel
MRR	**M**ean **R**eciprocal **R**ank
DNN	**D**eep **N**eural **N**etwork
LB	**L**ower **B**ound
UB	**U**pper **B**ound
LM	**L**anuage **M**odel
SVM	**S**pport **V**ector **M**achine

Chapter 1

Introduction

1.1 Research Background

In recent years, the rapid boost of massive data volumes has led to the increasing need for efficient and effective search activities. One common search activity often requires the user to submit a query to a search engine and receiving answers as a list of documents in ranked order. Such search task is called document retrieval and has been studied by the information retrieval community for many years. Before receiving the ranked documents, users often need to submit a query to the search system. Then the documents are sorted by the relevance calculated according to the query. Query formulation thus arose and focused on improving the overall quality of the document ranking with regard to the submitted query. As an important subtask of query formulation, query autocompletion (QAC) aims at helping the user formulate his query while he is typing only the prefix, e.g., several characters. The main purpose of QAC is to predict the user's intended query and thereby save his keystrokes. QAC has become a main feature of today's modern search engines, e.g., Google, Bing and Yahoo.

In some cases, it is possible to pre-compute and index all the completion candidates in efficient data structures. However, when query autocompletion is applied in some complex applications, such as Web mapping service, input method editors (IMEs) and programming integrated development environments (IDEs), keeping fast response time and accurate performance becomes challenging. This has created a need for efficient and effective *autcompletion for online services*.

Following this trend, the database and software engineering community has begun to investigate autocompletion for various applications recently. Some research work has been done on autocompletion for Web mappings [1–4] and several autocompletion systems for IDEs have been proposed [5–9].

The general approach to achieving efficient and effective autocompletion is to use an index structure, which can be taken as a search tree for fast lookups. For effective autocompletion, a straightforward way is to collect popularity statistics for each candidates and rank them according to this criterion.

In this thesis, we focus on both efficient and effective autocompletion methods. In other words, we use novel index structures to improve search performance and also collect various features for more accurate ranking. Under this setting, we combine these two ideas together to design our algorithms and data structures. We focus our effort on applications for Web mappings, IMEs, IDEs and desktop search.

In Fig. 1.1, we show an example of autocompletion for Web search engines. When a prefix is input, it will give a candidate list to show all the completion candidates. When large amount of users are intending to obtain completions from the server, they have to wait in a priority queue thus incurring response time lags.

In the first work, we consider *location-aware query autocompletion*, one of the most important applications for Web mapping service. A query for such service includes a prefix and a location. It retrieves all the objects that are near the given location and begin with the given prefix. The location can either be a point or a range. In practice, this query can help a user to find a proper point of interest (POI) with the prefix and location provided quickly.

Our second work studies *prefix-abbreviated query*, which has a wide usage in IMEs, IDEs and desktop search. This query can retrieve relevant results when the query is a prefix-like abbreviation. We focus both efficiency and effectiveness of this approach. An application example is to find the candidates by inputting letters as few as possible.

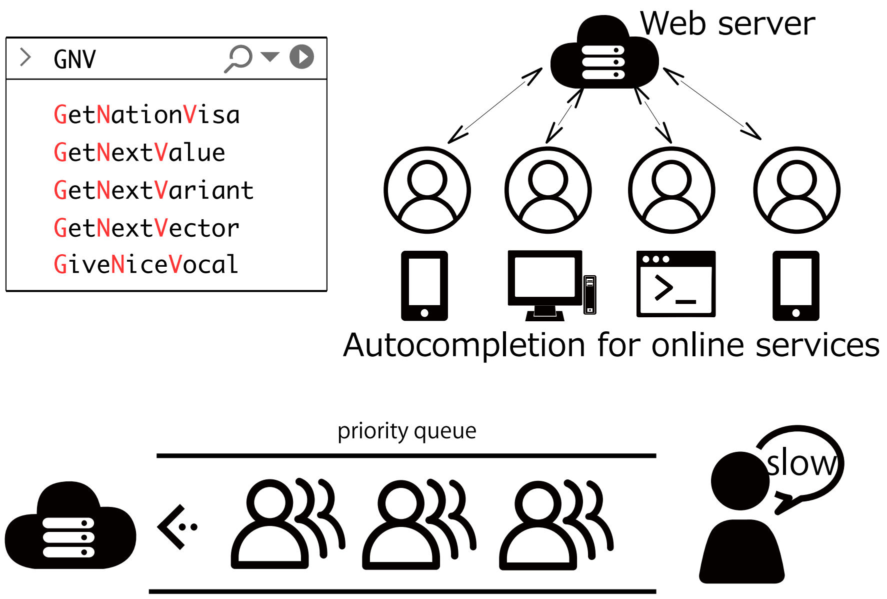


FIGURE 1.1: Query autocompletion service.

The third work explores *scope-aware code completion*. Given a free acronym-like input of a programming method name, we can suggest the completions in a fast and accurate way. This is extremely crucial when online IDEs such as IBM Bluemix and Overleaf are becoming popular these days. We utilize the discriminative model to measure the relevance between input and different completion candidates.

1.2 Research Objectives and Contributions

We have conducted three works on autocompletion for online services. Our first work is motivated by the prevalence of autocompletion in Web mapping services. In the first work, we adopt two kinds of query settings: range queries and top-k queries. Both queries contain two kinds of information. One is a location, such as a point or an area. The other one is a string prefix. A point of interest will be returned if it is close to the spatial location and its textual contents begin with the given prefix.

Although there have been several solutions [1–4] to location-aware query autocompletion that are based on a combination of spatial and textual indexes to process queries, all of them suffer from inefficiency when the dataset is large or when large amount of simultaneous queries occur. Most existing works can be classified into text-first, space-first, and tightly-combined methods, according to how the indexes are combined. The text-first methods first index the text descriptions and then apply spatial constraints as filters to verify the objects. E.g., if a user searches for “Starbucks” around “New York”, text-first methods will first find all the objects matching “Starbucks” and then verify whether they are around “New York”. The space-first methods adopt the reverse order. The tightly-combined methods will transfer “Starbucks” and “New York” into one combined token and then use it as a key for lookups. After analyzing the characteristics of queries, we develop a novel text-first indexing method to answer both range and top-k queries. We choose to store the spatial information of data objects in our trie index for fast lookups. We develop several pruning techniques which use pointers to quickly locate data objects and avoid unnecessary access. Due to these techniques, only a small part of objects need to be checked. We also extend our method to deal with error-tolerant problem which can suggest correct queries when there are typos in the user input.

In the second work, we study a novel QAC paradigm. We propose a novel QAC paradigm through which users may abbreviate keywords by prefixes and do not have to explicitly separate them. There is no previous work on such implicit autocompletion paradigm. Moreover, the naïve approach of performing iterative search on a trie is also computationally expensive for the problem in this work. Thus a novel indexing and query processing scheme called nested trie is proposed to efficiently complete queries. Moreover, 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. The proposed approaches can efficiently and effectively suggest completion results under the new problem settings.

The third work takes a different view and focuses on the ranking performance of code completion, which is a traditional popular feature for API access in IDEs. The prompted suggestions not only free programmers from remembering specific details about an API but also save keystrokes and correct typographical errors. Existing methods for code completion usually suggest APIs based on popularity in code bases or language models on the scope context. However, they neglect the fact that the user’s input habit is also useful for ranking, as the underlying patterns can be used to improve the accuracy for predictions of intended

APIs. In this work, we propose a novel method to improve the quality of code completion by incorporating the users' acronym-like input conventions and the APIs' scope context into a discriminative model. The weights in the discriminative model are learned using a support vector machine. We employ a trie index to store the scope context information. An efficient top-k suggestion algorithm is developed.

Our contributions are summarized as follows:

- For Web mapping service, we propose a novel indexing method to support efficient query processing.
- To mine the underlying input behavior, we formally define the problem of prefix-abbreviated autocompletion and propose efficient and effective approaches to solve such a non-trivial problem.
- For the practical IDE completion, we formalize the problem of scope-aware code completion, and describe this problem with a discriminative model.
- We propose to use advanced machine learning techniques to obtain better performances while keep competitive response speed.
- We demonstrate the efficiency and effectiveness of our approaches through comprehensive experimental performance studies.
- Our works are orthogonal to studies in other domains such as information retrieval and software engineering thus can be applied directly to those domains.

1.3 Real World Data Circulation

We also realize the real world data circulation by adapting our works in this thesis. Such a circulation consists of three steps: Acquisition, Analysis and Implementation. Acquisition here represents the observations of entire users' activities that occur from the user's first keystroke in the search box to the last selection of an autocompletion candidate provided. In Analysis step, we focus on revealing the input behavior patterns of users by employing machine learning and statistical modeling techniques for accurate predications. In Implementation step, we demonstrate superior performances using our prototype autocompletion systems.

We also show such a data circulation can benefit our society in terms of e-commerce search system, Location-based Service and online text editors. More details can be found in Chapter 5.

1.4 Related Work

Due to its important applications for online services, the research on QAC has received much attention in various domains the last few decades. We refer readers to two surveys for various kinds of QAC [10, 11].

In this thesis, we develop three works from the prospectives of database (DB) [12, 13], information retrieval (IR) [13] and software engineering (SE) [14]. There are some common ground for these three domains. Studies in IR [15–17] and SE [5, 7, 18] tend to utilize statistical models or language models to describe the QAC as a recommendation problem. They usually use statistics estimated or learned from the raw log data or code bases to evaluate the effectiveness of such recommendation systems. At the same time, works in DB [19–21] and SE [5, 7, 18] focus more on the indexing data structures to pursue fast response time. Intriguingly, novel data structures can also help to mine the underlying patterns hiding in the raw log data or code bases thus leading to improvements on recommendations.

QAC is a classical problem in the community of information retrieval. Darragh and Witten [22] developed the reactive keyboard to predict future keystrokes based on past interactions. Other early studies considered completing the query at word [23] or phrase level [24, 25]. Fan *et al.* [26] studied the suggestion on topic-based query terms. Bhatia *et al.* [27] investigated the case when query logs are absent. To help users quickly identify useful results, Jain and Mishne [28] proposed to cluster suggestions and label them. Recent trends feature a boom in context-aware QAC where user interactions are important [15, 29–37], as well as plenty of work on presenting time-sensitive [16, 38–41], personalized [17, 42, 43], or diversified results [44, 45]. Besides these studies, Mitra and Craswell investigated the case for rare prefix [46]. Cai and de Rijke proposed to use homologous queries and semantically related terms [47]. Zhang *et al.* targeted the QAC for mobile devices [48]. There are also investigations in the direction of trie compression techniques to seek space efficiency [49, 50]. As for the quality of QAC, an experimental evaluation was reported in [51] to compare various ranking methods, including the traditional method by static score (past popularity), context-aware/user-interactive ranking, and learning to rank [52, 53]. Markov model was also adopted [33] to produce more meaningful results.

In the database research community, Li *et al.* designed the TASTIER system for type-ahead search on relational data [54]. 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 [19–21, 55–57] or Markov n-gram transformation model [58]. Cetindil *et al.* [59] proposed a ranking method for error-tolerant QAC using proximity information. Another popular direction is location-aware QAC [1–4, 12], which is useful for navigation tools. By adding the spatial constraints, the spatial keyword search problem is to return the relevant POIs considering both spatial proximity and textual relevance. This problem has been extensively studied in the database community. Existing solutions are based on R-tree [60–64], grid [65, 66], and space filling curve [67]. We also refer users to an experimental evaluation [68] that compares these methods. Besides, there are studies on QAC for specific applications, e.g., command line shells [69, 70], Chinese pinyin input [71, 72], IDEs [5, 7, 18, 73, 74], and medical vocabulary [75].

Another body of work aims at query reformulation by taking a full query and making arbitrary modifications to assist users. The solutions are based on query clustering [76–78], session analysis [79, 80], search behavior [81], and Markov models [80, 82, 83], etc. In some studies, query autocompletion and reformulation are both called query suggestion.

The tolerant retrieval problem [84] 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 [85]. Another related problem is multi-dimensional substring search [86, 87]. 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. Other related problems include subsequence search [88] and regular expression search [89–91].

A special type of QAC can be categorized as processing queries involving abbreviations. Related work comes from human-computer interaction [92–95] and database research community [96]. In addition, in the database area, many studies targeted the more general case of abbreviations in the name of transformation rules [97–99] or synonyms [100, 101].

TABLE 1.1: Related work of text autocompletion

Type	+spatial constraints			+abbreviated matching
	+range	+top-k	**+error**	+top-k
text QAC	[64], etc.	[1], etc.	**[12]**	**[13, 14]**
keyword search	[68], etc.	[68], etc.	[102], etc.	[5], etc.

Table 1.1 summarizes the closest related work of text autocompletion and describes the position of our works[12–14]. There are two categories of research about text retrieval. One is text QAC, the other is keyword search. Many studies [1, 64, 68] have been conducted to apply spatial constraints such as range queries, top-k queries and error-tolerant queries in both domains. Our work [12] is the first one who apply spatial constraints and address error-tolerant problem in the domain of text QAC. Our work [13, 14] is also the first one to apply abbreviated matching paradigms and solve the top-k ranking problem in the domain of text QAC.

1.5 Thesis Organization

The rest of the thesis is organized as follows. In Chapter 2, we introduce our approach of location-aware query autocompletion. Then we present the autocompletion for prefix-abbreviated input in Chapter 3. Chapter 4 describes the scope-aware code completion with discriminative modeling. Chapter 5 introduces the data circulation in this thesis. Eventually, in Chapter 6, we conclude all the works and discuss the future studies.

Chapter 2

An Efficient Algorithm for Location-aware Query Autocompletion

2.1 Introduction

Query autocompletion is an important feature in search engines, command shells, desktop search, software development environments, and mobile applications. It reduces the number of keystrokes input by the users and helps improve the throughput of the system as query or intermediate results can be effectively cached and reused. With the growing popularity of mobile devices, a recent trend is to integrate query autocompletion into location-based services. One of the main applications is to complete the queries with the textual descriptions of nearby points of interest in a Web mapping service as illustrated in Example 2.1.

Example 2.1. *In Fig. 2.1, a user at point P wants to search for nearby Starbucks. He zooms in the region around and types in the first few characters such as "starb", and then the system automatically suggests "starbucks" and "starboost". The two results are also marked as points on the map as A and B.*

We call this problem *location-aware query autocompletion*. A query of this problem includes a location, such as the point *P* or the area *R* in Fig. 2.1. The query also includes a string prefix, a point of interest will be returned if it is close to the spatial location and its textual contents begin with the given string prefix.

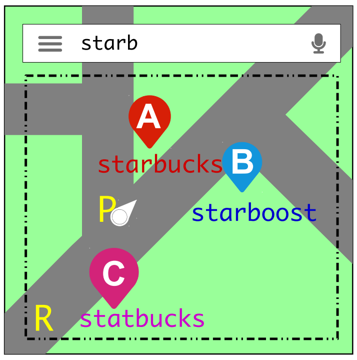


FIGURE 2.1: Location-aware error-tolerant autocompletion.

In addition, the error-tolerant autocompletion also becomes very popular, because misspellings may occur due to typos in the queries or data uploaded by users, especially when users are typing with the error-prone keyboards of mobile devices. Error-tolerant feature can help to suggest correct results even when there are typos on both query prefix and data sides as showed in Example 2.2.

Example 2.2. *In Fig. 2.1, suppose the user types in characters such as "sdarb", and then the system with error-tolerant feature suggests "starbucks", "starboost" and "statbucks". The three results are marked as points on the map as A, B and C.*

There have been several solutions to location-aware query autocompletion. All of them are based on a combination of spatial and textual indexes to process queries. They can be categorized into text-first [1], space-first [2, 3], and tightly-combined [4] methods, according to the ways in which the indexes are combined. For text-first methods, string descriptions of data objects are indexed in a trie, where objects as well as their locations can be retrieved on leaf nodes of the trie. For space-first methods, data objects are indexed in an R-tree or quadtree by their locations, and textual filters are applied when processing queries. For tightly-combined methods, textual and spatial information are both considered to build the index. However, all of the existing approaches suffer from inefficiency when the dataset is large, and the performance is deteriorated when large amount of simultaneous queries occur.

In this work, we investigate the problem of location-aware query autocompletion and aim at answering range and top-k queries. We discuss the advantages of text-first indexes over space-first and tightly-combined indexes, and propose a novel trie-based text-first method to efficiently process the two types of queries. The existing text-first approach MT (Materialized

Trie) [1] is only applicable to top-k queries and computes score upper bounds by enumerating query locations. Hence the memory consumption is huge, and it has to materialize score upper bounds for regions with coarse granularity and only a subset of trie nodes. Unlike MT which stores the spatial information of queries on trie nodes , we choose to store the spatial information of data objects instead. Several pruning techniques are developed on top of our index. We propose to use pointers to quickly locate data objects, in contrast to MT which traverses subtrees to find data objects. In addition, the error-tolerant feature is also taken into consideration. We extend our method to handle this feature and suggest correct queries. Finally, we demonstrate the superiority of our solution through extensive experimental evaluations on real datasets.

Our contributions can be summarized as follows:

- We propose a novel location-aware query autocompletion method to efficiently answer range and top-k queries with an acceptable index size. We are able to index a sufficiently large dataset with 13 million points of interest in 32GB main memory on a commodity machine, and answer both range queries and top-k queries in microseconds or even faster.
- We integrate the error-tolerant feature into our method so as to handle the case when users input queries with error-prone devices.
- We conduct experiments to evaluate the efficiency of the proposed methods with comparisons to existing solutions.

The rest of this chapter is organized as follows. Section 2.2 defines the problems. Section 2.3 surveys related work. Section 2.4 presents our index structure for location-aware query autocompletion. Section 2.5 introduces the query processing algorithms. Section 2.6 reports experiment results and analysis. Section 2.7 concludes this chapter.

2.2 Preliminaries

2.2.1 Problem Definition

Let $\mathcal{O}$ be a set of data objects in a spatial database. Each object $o \in \mathcal{O}$ is represented by a tuple $\{o.str, o.loc, o.scr\}$. $o.str$ is the string description. $o.loc = (x, y)$ and describes the

TABLE 2.1: An example database $\mathcal{O}$.

Object ID	$o.str$	$o.loc$	$o.scr$
o_1	`navitime`	$(24, 25)$	0.4
o_2	`nagoyadome`	$(18, 12)$	0.9
o_3	`nagoyaport`	$(11, 19)$	0.8
o_4	`nursing`	$(1, 19)$	0.7
o_5	`stone`	$(7, 27)$	0.1
o_6	`studio`	$(27, 12)$	0.1
o_7	`starbucks`	$(22, 18)$	1.0
o_8	`starboost`	$(5, 5)$	0.3
o_9	`station`	$(19, 9)$	0.8
o_{10}	`school`	$(15, 29)$	0.6

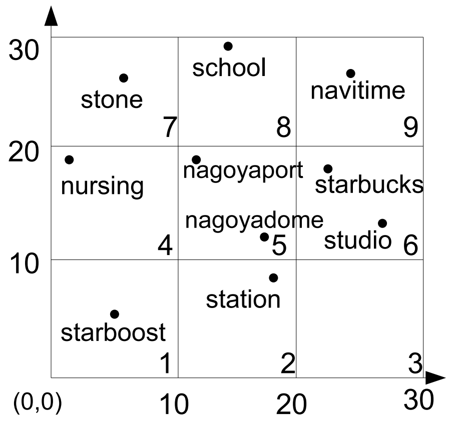


FIGURE 2.2: $\mathcal{O}$ in 2-dimensional space.

location in 2-dimensional space. $o.scr$ is the static score which can be used to reflect the popularity of the object. $global\_max\_scr$ denotes the maximum static score of the objects. $global\_max\_dist$ denotes the maximum distance between two objects in $\mathcal{O}$. An example is shown in Table 2.1 and Fig. 2.2.

Given two strings s and s', "$s' \preceq s$" denotes that s' is a prefix of s; i.e., $s' = s[1..|s'|]$. In this work, we focus on supporting two types of queries:

Range Query. The query q consists of a query string $q.str$ which the user is typing in and a range $q.rng$ defined by a rectangle. The answer to the query q consists of the objects $o \in \mathcal{O}$ such that $q.str \preceq o.str$ and $o.loc$ is in the range $q.rng$.

Top-k Query. The query q consists of a query string $q.str$ which the user is typing in and a location $q.loc$. The answer to the query q consists of the top-k objects $o \in \mathcal{O}$ such

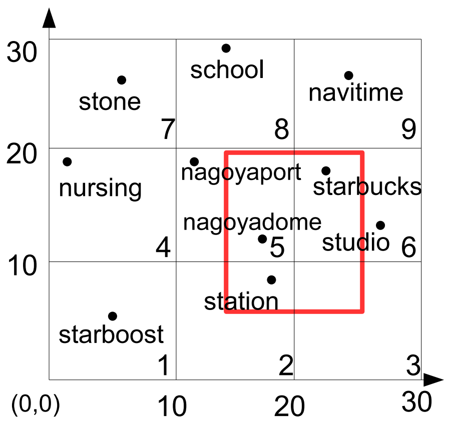


FIGURE 2.3: A range query.

that $q.str \preceq o.str$, sorted by a ranking function $F(o,q)$. We focus on the following ranking function that gives an overall score of an object with respect to the query q, but our method can be extended to support other monotonic functions.

$$F(o,q) = \alpha \times \frac{o.scr}{global\_max\_scr} + (1-\alpha) \times \left(1 - \frac{dist(o.loc, q.loc)}{global\_max\_dist}\right). \tag{2.1}$$

The first component $\frac{o.scr}{global\_max\_scr}$ in the ranking function measures the popularity with the object's static score normalized into the range of [0, 1]. The second component $(1 - \frac{dist(o.loc,q.loc)}{global\_max\_dist})$ measures the spatial proximity by subtracting from 1 the normalized Euclidean distance between the object and the query. The two components are balanced by a weight variable α. It is determined by application. A higher α indicates that the user is more towards popularity. When $\alpha = 0$, the user only cares about proximity, and hence the query becomes k nearest neighbors. When $\alpha = 1$, the user only cares about popularity, and hence the results are the top-k objects ranked by static score.

Example 2.3. *Figure 2.3 shows an example of a range query. The query range is defined by the red rectangle. Suppose the query string is "*`sta`*". The results are o_7 and o_9. Figure 2.4 shows an example of a top-k query. The query location is shown by the red cross. Suppose the query string is "*`na`*", $k = 2$, and $\alpha = 0$. The results are o_2 and o_3.*

In some applications, especially for mobile devices, the user's input tends to contain typographical errors. In this case, we also support the error-tolerant feature so that a number

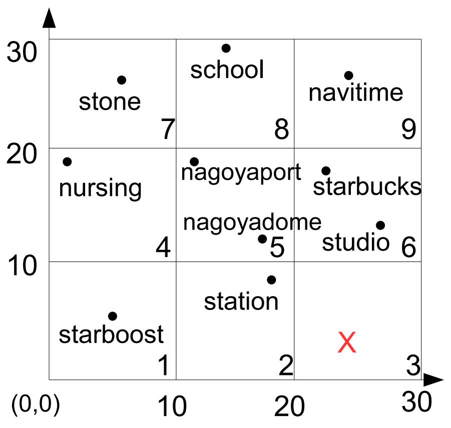


FIGURE 2.4: A top-*k* query.

of errors are allowed in the query. We choose to use edit distance to measure the input errors. $ed(s,t)$ 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. Given a threshold τ, we return the results such that $\exists o.str' \preceq o.str, ed(o.str', q.str) \leq \tau$; i.e., the edit distance between the query string and a prefix of the object string is within τ. Note that we can incorporate the edit distance as a feature into Equation 2.1 linearly or nonlinearly for better ranking performance. In this work, we do not consider such techniques but focus on indexing techniques.

In this work, we compute the results incrementally as the user types in characters. In addition, we focus on the in-memory and stand-alone implementation of the algorithms.

2.3 Related Work

Location-aware Query Autocompletion. There have been several existing studies to support location-aware query autocompletion. These methods combine spatial and textual indexes. They can be divided into three categories according to the way the two indexes are combined: text-first, space-first, and tightly-combined.

Materialized Trie (MT) is a text-first method proposed by Roy and Chakrabarti [1] to find top-*k* results ranked by a linear combination of static score and physical distance. The strings of data objects are indexed in a trie, where objects as well as their locations can be retrieved on leaf nodes. Spatial information is stored on trie nodes to speed up query processing. For each node, it divides the whole space into a grid, and stores for each region the score upper bound if the query location is in the region. MT suffers from the consumption of large amount of memory. Although a remedy was proposed to materialize a subset of M trie nodes and store R bounds in each of them, it is at the expense of runtime performance.

Filtering-Effective Hybrid Indexing (FEH) is a space-first method proposed by Ji *et al.* [2] to answer range queries and kNN queries. The method builds an R-tree to index data objects by their locations. Textual filters are used in each R-tree node to check whether the query string is a prefix of the objects in the subtree. INSPIRE [3] is a space-first method developed for a variety of spatial-textual queries. Data objects are indexed in a quadtree and the nodes are encoded by Hilbert curve. When traversing the quadtree, textual filtering is carried out with the help of an inverted index on the q-grams of object strings. The inverted index is partitioned as per the Hilbert curve for fast lookup. The major drawback of the space-first methods is that when the query string is short or frequent, the pruning power of the textual filters becomes very poor and thus the runtime overhead drastically increases.

Prefix Region Tree (PR-Tree) [4] is a tightly-combined method that considers textual and spatial partitioning simultaneously to build the index. Data objects are indexed in a trie, and then each node is divided into four nodes, each representing a region in a quadtree, with centroids selected as the center for partitioning. The major problem of PR-Tree is that although spatial conditions can be checked with the quadtree, more nodes have to be accessed for query processing due to the division of trie nodes.

Error-tolerant Query Autocompletion. Query autocompletions with edit distance to tolerate errors were first studied in [103] and [19]. Li *et al.* [20] improved the method proposed in [103] for space and runtime performance. More efficient methods were proposed in [55] and [57]. Apart from edit distance, cosine similarity [15] and Markov n-gram transformation model [58] are also adopted for error tolerance in the autocompletion task.

Spatial Keyword Search. Given a database of points of interest (POIs) and a query composed of keywords and a location, the spatial keyword search problem is to return the relevant POIs considering both spatial proximity and textual relevance. This problem has been extensively studied in the database community. Existing solutions are based on R-tree [60–64], grid [65, 66], and space filling curve [67]. We also refer users to an experimental evaluation [68] that compares these methods.

TABLE 2.2: Related work of location-aware QAC

Type	+range	+top-k	**+error**
location-aware QAC	[64], etc.	[1], etc.	**[12]**
spatial keyword search	[68], etc.	[68], etc.	[102], etc.

Table 2.2 shows our position in the related work. The closest existing studies can be categorized into two categories: location-aware QAC and spatial keyword search. A lot of works [1, 64, 68] have been proposed to handle range queries and top-k queries in both domains. Our work [12] is the first one to handle location-aware queries with an error-tolerant feature.

2.4 Index Structure

In this section, we introduce our indexing method for location-aware query autocompletion.

Our index belongs to the text-first category. The reasons why we resort to a text-first index are:

- For location-aware query autocompletion, a common scenario is that the query string is short, hence rendering the text filter of space-first indexes unselective. E.g., the frequency of the q-gram "an" is 0.18 in the FSQ dataset, which consists of 1 million worldwide points of interest collected from Foursquare, meaning that on average at least 18% objects under a tree node will be fetched for verification if the query string contains a character pair "an". In contrast, text-first indexes are based on a trie. We only need to traverse the trie to match the query string, and thus good query processing performance can be achieved.
- Tightly-combined indexes exhaustively divide nodes in the tree using spatial information, thus resulting in redundant node access when processing queries. For text-first indexes, because a trie is compact (given a query string, there is only one matching path), node access can be saved.

- To verify spatial information in text-first indexes, we may equip trie nodes with spatial filters and verify spatial conditions efficiently when fetching data objects under a trie node.
- It is easy to implement and flexible to choose data structures for spatial access, e.g., R-tree, quadtree, grids, etc.

Note that a brief explanation of choosing text-first index also appears in [1], which also adopts the text-first method, but from different perspectives: (1) text-first method is able to support any ranking function, and (2) text-first method is optimum in space requirement.

The basic structure of our index is a trie built on the set of object strings. Each node is uniquely identified by a string corresponding to the labeled path from the root to the node, so that search can be easily performed by identifying the node that matches the query string. Given a node in the trie, we say a data object is an *underlying object* of this node if the path from the root to the node is a prefix of the object string. If we run a pre-order traversal in the trie, each node can be assigned a number by the order in which they are accessed. Consider the objects in Table 2.1. Figure 2.5 shows the trie in which the object strings are indexed. Nodes are numbered by the pre-order traversal. o_1 is an underlying object of node 2, because `na` is a prefix of `navitime`.

Next we introduce how to integrate spatial data structure into the trie. First, the global space is partitioned into a set of spatial regions. This step can be done using common data structures for spatial objects, such as grid, R-tree, quadtree, etc. The partitions can be either overlapping (e.g., by an R-tree) or non-overlapping (by a grid). A region represents a cell if we use a grid to partition the space, and a leaf node if we use a tree-based data structure. For ease of illustration, we describe our method by assuming the space is partitioned by a grid, although experimental evaluation shows that partitioning by a quadtree yields better query processing performance.

Each object belongs to a region with respect to the spatial partitioning. We equip each node in the trie with a bit array, each bit representing a region. A bit is set to 1 if the node has an underlying object in this region; or 0, otherwise. We call it *region bit array*. With this bit array, when searching in the trie, we are able to check whether there is an underlying object in the region that intersects the query range. Note that the bit arrays on the trie only need to be constructed once when the trie is built. As the length of the bit array equals the number of cells of the quadtree partition. It can be easily controlled by enlarging the quadtree partition capacity for each cell. In our experiments, the length of the bit array is at most 1024.

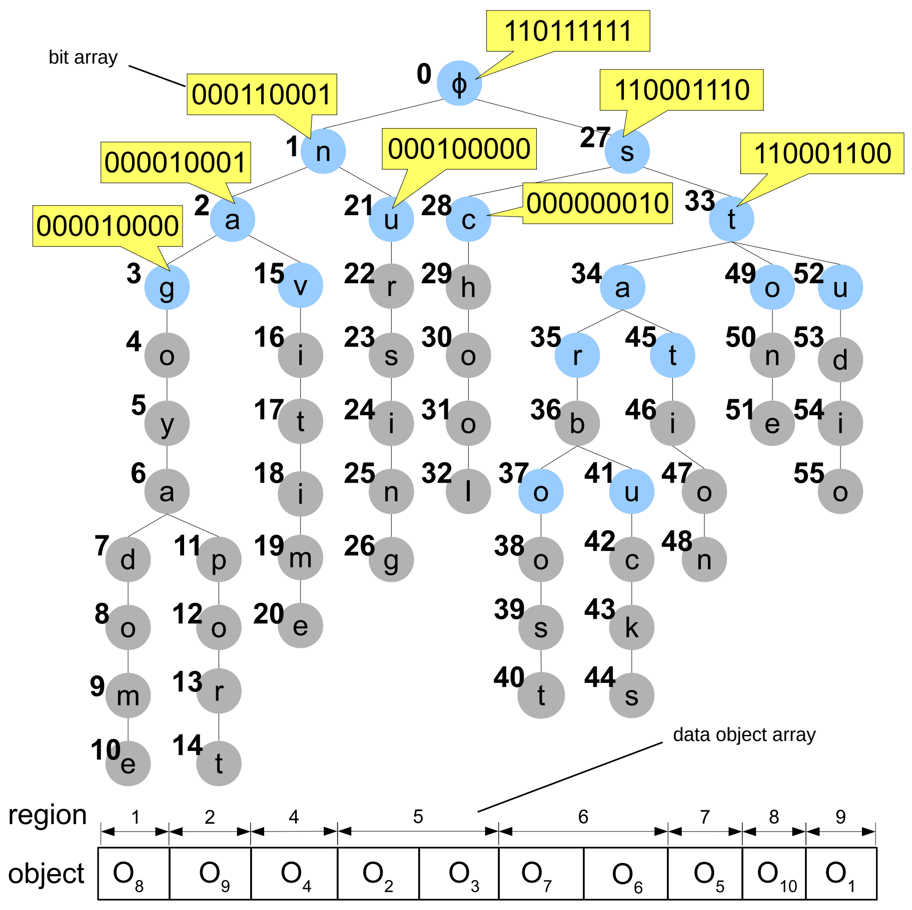


FIGURE 2.5: The bit trie index.

Example 2.4. *Assume that the space is partitioned as per the grid in Fig. 2.2. We show in Figure 2.5 the region bit arrays of the nodes in the first three levels. The n-th bit represents the cell numbered n in Figure 2.2.*

We observe that some nodes share the same region bit arrays as their parents in the trie. In this case, the child nodes' bit arrays are redundant and have no pruning power for query processing, and hence we only keep the bit array of the parents to save space and search time. We also use a flag **SameAsParent** on these child nodes so that when we need to retrieve their bit arrays, their parents are referred to. For example, in Figure 2.5, all the nodes under 3 have the same bit array 000010000 as node 3. We only keep the bit array 000010000 at the node 3, and remove those at its descendant nodes. In Figure 2.5, a node is colored grey if it shares the same bit array as its parent.

Data objects are stored in an array called *data object array*, which is partitioned into spatial regions as well. Since an object string corresponds to a leaf node in the trie, within each region in the data object array, we sort objects in the order of their corresponding leaf nodes in the trie. To quickly identify underlying objects in the array, each node in the trie is equipped with a list called *region list*, whose entries are in the form of $\langle$region ID, maximum static score, starting pointer, ending pointer$\rangle$. The maximum static score of an entry is the maximum static score of the node's underlying objects in this region, and it is used to efficiently answer top-k queries. The starting and ending pointers are used to fetch results in the data object array. They are linked to the starting and ending positions in the partial array that contains the underlying objects of the node in this region, respectively. Entries in the list are sorted by descending maximum static score order. Note that the sorting can be finished within a trivial time offline.

Example 2.5. *Consider node 2 in Fig. 2.5. Its region list is* $[\langle 5, 0.9, 4, 5\rangle, \langle 9, 0.4, 10, 10\rangle]$. *Note that this list is stored at the trie node showed in Fig. 2.5. The maximum static score of this node is 0.9.*

For index construction, one may notice that if siblings in the trie are arranged in alphabetical order, the objects in each region in the data object array follow the lexicographical order. This property facilitates the index construction: Given the set of data objects $\mathcal{O}$, we divide it with the partitioning first, and then sort the objects in each region by lexicographical order. The trie is initialized as empty. For each region, the objects are scanned one by one, and their strings are inserted into the trie. Once a string is inserted, we update the region bit arrays and the region lists of the nodes on the path, as well as the maximum static scores of the nodes. The time complexity of the index construction is $O(|\mathcal{O}|log|\mathcal{O}| + S)$, where S is the sum of string lengths of the objects.

2.5 Query Processing Algorithms

The query processing algorithms are introduced in this section. We first present the algorithms for answering range queries and top-k queries, respectively, and then show how to extend them to cope with the error-tolerant case.

2.5.1 Processing Range Queries

The query processing is divided into two phases: (1) searching phase, in which the query string is looked up in the trie and the spatial condition is checked; and (2) result fetching phase, in which the data object array is accessed to fetch and return results.

We begin with the searching phase. To process a query $\langle q.str, q.rng \rangle$, we first compare $q.rng$ with the spatial partitioning and obtain the regions occupied by $q.rng$. We initialize a bit array, where a bit is set to 1 if $q.rng$ intersects a region; or 0, otherwise. The bit array is called *region status*. For example, the initial region status for the query in Fig. 2.3 is 011011000 because the query range intersects regions 2, 3, 5, and 6 in the grid. Note that the length of the bit array is influenced by the partitions of quadtree. In our experiments, the length of bit array is at most 1024.

Then we start to traverse the trie with the query string. As the user types in the query, we follow the path that matches the query string. For each node, the region status is updated by a bitwise AND operation with the region bit array of the node. Since the region bit array keeps track of whether there is an underlying object in a region, if a bit becomes 0 after the bitwise AND operation, it means that there is no underlying object in this region for the query. We say a node is an *active node* if its path matches the query string and the region status is not all zero. The traversal in the trie is essentially to check if there is an active node for the next keystroke input by the user. Whenever there is no path matching the query string or the region status becomes all zero, we can stop the traversal of the trie and return no results.

The above process is shown in Algorithm 1. It takes as input the query and the trie. First, a region status is initialized (Line 1). Then it traverses the trie to match the next keystroke (Line 4) and update the region status (Line 5). If this is no match for the keystroke or the region status becomes all zero, it exits the traversal. It returns the active node and the region status for result fetching, or null to indicate there is no result (Line 14). The time complexity is $O(|q.str|)$, where $||$ denotes the length of a string.

We also observe that if a bit in the region status becomes 0 during the traversal, it will never return 1. Hence a blocking technique can be devised to save bitwise operations. We divide region bit arrays and the region status into equi-width blocks (e.g., 64-bit blocks with the quadtree partition), and only keep the blocks with at least a 1 in the region status.

Algorithm 1: RangeQuerySearch (q, T)

```
1  $b \leftarrow$ InitRegionStatus$(q.rng)$;
2  $n \leftarrow$ the root of $T$;
3  foreach keystroke $q.str[i]$ do
4      if $n$ has a child $n'$ through label $q.str[i]$ then
5          $b \leftarrow b$ AND the region bit array of $n'$;
6          if $b \neq 0$ then
7              $n \leftarrow n'$;
8          else
9              $n \leftarrow$ null;
10             break;
11     else
12         $n \leftarrow$ null;
13         break;
14 return $n, b$
```

Algorithm 2: RangeQueryFetchResult (q, n, b, T, A)

```
1  if $n =$ null then return $\emptyset$ ;
2  $R \leftarrow \emptyset$;
3  foreach $\langle r, m, s, e \rangle \in n$'s region list do
4      if $b[r] = 1$ then
5          foreach $o \in A[s, e]$ do
6              if $o$ is in $q.rng$ then
7                  $R \leftarrow R \cup \{o\}$;
8  return $R$
```

The result fetching phase of range query is as follows. We obtain the bits equal to 1 in the region status, and scan the corresponding regions in the region list. With the starting and ending pointers, the objects in the data object array are located. Each object between the two pointers is verified with the query for the spatial constraint; i.e., if the location of the object is within the query range. The object is returned as a result if it passes this verification. The pseudocode of the result fetching phase is shown in Algorithm 2, which reads in the active node n and the region status b, and returns the result set R. The time complexity is $O(\Sigma_{i=1}^{|L|} e_i - s_i + 1)$, where L denotes the region list, s_i and e_i denote the starting and ending pointers of the i-th entry in the list, respectively.

Example 2.6. *Consider the query range in Fig. 2.3 and a query string* s. *The region status is initialized as 011011000. We start with the root and traverse the trie. Since there is an edge* s, *we follow this edge and reach node 27. Its region bit array is 110001110.*

Algorithm 3: TopKQuerySearch (q, T)

```
1 $n \leftarrow$ the root of $T$;
2 foreach keystroke $q.str[i]$ do
3     if $n$ has a child $n'$ through label $q.str[i]$ then
4         $n \leftarrow n'$;
5     else
6         $n \leftarrow$ null;
7         break;
8 return $n$
```

Algorithm 4: TopKQueryFetchResult (q, n, T, A)

```
1 if $n =$ null then return $\emptyset$ ;
2 $R \leftarrow \emptyset$ ;                                /* a priority queue of size $k$ */
3 foreach $\langle r, m, s, e \rangle \in n$'s region list do
4     foreach $o \in A[s, e]$ do
5         $scr \leftarrow F(o, q)$;
6         if $|R| < k$ or $scr > R[k].scr$ then
7             $R.Insert(o, scr)$;
8 return $R$
```

By bitwise AND *operation, the region status becomes 010001000. The node's region list is* $[\langle 1, 0.3, 1, 1 \rangle, \langle 2, 0.8, 2, 2 \rangle, \langle 6, 1.0, 6, 7 \rangle, \langle 7, 0.1, 8, 8 \rangle, \langle 8, 0.6, 9, 9 \rangle]$. *With the region status, regions 2 and 6 in the list are accessed. For region 2, the pointers are 2 and 2. So we scan the 2nd object* o_9 *in the data object array, and verifies it as a result. For region 6, the pointers are 6 and 7. So we scan the 6th and 7th objects in the array.* o_7 *is in the query range and becomes a result, while* o_6 *fails the verification when the red rectangle is applied to check if* o_6 *is within it. The final results are* o_7 *and* o_9.

2.5.2 Processing Top-k Queries

2.5.2.1 Basic Algorithm

The algorithm framework of processing top-k queries is similar to processing range queries, except that the region status is not involved as there is no spatial constraint. The pseudocode of the searching phase is given in Algorithm 3. It matches the input keystrokes and follows the path in the trie to find the active node.

Algorithm 4 captures the basic result fetching algorithm for top-k queries. It initializes a priority queue of size k to store temporary results (Line 2). Then it iterates through the region list of the active node (Line 3), retrieves objects in the data object array (Line 4), and computes the overall score by Equation 2.1 (Line 5). If the priority queue has less than k results or the score is greater than the k-th temporary result, we insert into the queue the object accompanied with its score. The queue containing the top-k results is returned eventually as the final results (Line 8). The time complexity is $O(\Sigma_{i=1}^{|L|} e_i - s_i + 1)$, where L denotes the region list, s_i and e_i denote the starting and ending pointers of the i-th entry in the list, respectively.

The basic result fetching has to scan all the entries in the active node's region list and all the objects bounded by the pointers. If we scan less number of elements in the two processes, the query processing performance can be improved. Next we present two major optimizations for the purpose of early termination.

2.5.2.2 Region Level Pruning

The first optimization is to scan only part of entries in the region list. Recall that the entries in the list are sorted by descending order of maximum static score. Given the score of the k-th temporary result and the maximum static score of the i-th entry in the list L, a distance threshold can be computed:

$$d_t = global\_max\_dist \times \left(1 - \times \frac{R[k].scr - \alpha \times \frac{L[i].m}{global\_max\_scr}}{1-\alpha} \right). \tag{2.2}$$

$R[k].scr$ and $L[i].m$ represents the score of the k-th temporary result and the maximum static score of the i-th entry in the list L, respectively. Then, a distance threshold d_t can be computed from Equation 2.1. It means that if any unseen object is better than the k-th temporary result, its distance to the query must be smaller than the distance threshold d_t. With the location of the query, we can compute the regions that are close enough to the query to meet this condition, and record these regions in a bit array. A bit is set to 1 if the distance from the region to the query is less than d_t; or 0, otherwise. We may invoke a bitwise AND operation between this bit

Algorithm 5: RegionLevelPruning

```
1  $b \leftarrow n$'s region bit array;
2  foreach $\langle r, m, s, e \rangle \in n$'s region list do
3      if $|R| < k$ then
4          $b' \leftarrow 1$ ;                                  /* set all bits in $b'$ to 1 */
5      else
6          $d_t \leftarrow$ ComputeDistThreshold$(R[k].scr, m)$;
7          $b' \leftarrow$ ComputeBoundedRegion$(q.loc, d_t)$;
8      $b \leftarrow b$ AND $b'$;
9      if $b = 0$ then
10         break;
11     else
12         if $b[r] \neq 0$ then
13             foreach $o \in A[s, e]$ do
14                 $scr \leftarrow F(o, q)$;
15                 if $|R| < k$ or $scr > R[k].scr$ then
16                     $R.Insert(o, scr)$;
```

array and the active node's region bit array. If the result is all zero, all the regions are beyond the distance threshold, and hence all the entries in the region list can be pruned.

With the above property, a pruning algorithm is developed. Its pseudocode is given in Algorithm 5, and we use it to replace Lines 3 – 7 of Algorithm 4. A bit array b is initialized as the active node's region bit array (Line 1). When processing each region r in the region list, the distance threshold is computed first (Line 6) by Equation 2.2. With the threshold, we generate the bounded regions (Line 7) that are close enough to the query. The regions are represented by a bit array b'. We take a bitwise AND operation between b and b', and write the result to b (Line 8). If b becomes all zero, it is guaranteed that no regions in the remaining list contain an object better than the k-th temporary result, and thus we stop scanning the region list (Line 10). Otherwise, (1) if the bit that represents r in b is zero, r is skipped; (2) otherwise, we fetch results in r and update the temporary results (Lines 13 – 16). Because this pruning technique is applied on the region level, we name it *region level pruning*.

Example 2.7. *Consider the objects in Table 2.1. The query string is* na*. Its location is shown in Figure 2.4.* $k = 2$*, and* $\alpha = 0.5$*. Suppose* $global\_max\_dist = 1$*. The distances from the query to* o_2*,* o_3*, and region 9 are 0.3, 0.5, and 0.4, respectively.*

TABLE 2.3: Bounded regions with varying distance thresholds.

Distance Threshold	Bit Array
$t = 0$	000000000
$0 < t \leq 0.082$	001000000
$0.082 < t \leq 0.164$	011000000
$0.164 < t \leq 0.183$	011001000
$0.183 < t \leq 0.318$	011011000
$0.318 < t \leq 0.358$	111011000
$0.358 < t \leq 0.400$	111111000
$0.400 < t \leq 0.408$	111111001
$0.408 < t \leq 0.511$	111111011
$t \geq 0.511$	111111111

We follow the path na*, the active node is node 2 in the trie. Its region bit array is 000010001, and its region list is* $[\langle 5, 0.9, 4, 5\rangle, \langle 9, 0.4, 10, 10\rangle]$*. We first process region 5. After that, the top-k temporary results as well as their scores are* $\langle o_2, 0.8\rangle$ *and* $\langle o_3, 0.65\rangle$*. The bit array b is 000010001. Next we process region 9. By Equation 2.2, the distance threshold is 0.1. The bounded regions are 2 and 3. So* b' *is 011000000. A bitwise* AND *operation on b and* b' *yields all zero. So we can prune region 9 and return* o_2 *and* o_3 *as the final results.*

There are two minor optimizations on the region level pruning. First, since the bounded regions only depend on the query location and the distance threshold, we can precompute all the possible bounded regions once the query location is received. We do not have to enumerate all the distance thresholds but only consider the values at which the bounded regions change. For example, for the query whose location is shown in Figure 2.4, Table 2.3 lists the bit arrays for the bounded regions under different distance thresholds. Second, due to the bitwise AND operation, a bit in b never returns 1 if it becomes 0. So we may use the blocking technique to divide b into equi-width blocks, and only keep those with at least a 1 to save bitwise operations.

2.5.2.3 Subtree Level Pruning

The second optimization is to reduce the number of objects to access in the data object array. Recall that in the basic result fetching algorithm we directly go through the objects between the two pointers. On the other hand, it can be observed that for a node n and a region r, if we fetch objects of r from all n's children, they exactly constitute the underlying objects of n in

Algorithm 6: SubtreeLevelPruning

```
1 foreach $n$'s child node $n'$ do
2     get $\langle r', m', s', e' \rangle$ from the region list of $n'$ where $r' = r$;
3     $scr_{ub} \leftarrow$ ComputeUpperBoundScore$(m', r', q.loc)$;
4     if $|R| < k$ or $scr_{ub} > R[k].scr$ then
5         foreach $o \in A[s', e']$ do
6             $scr \leftarrow F(o, q)$;
7             if $|R| < k$ or $scr > R[k].scr$ then
8                 $R.Insert(o, scr)$;
```

r. Since the maximum static scores have been recorded in the region lists of the child nodes, we may compute a score upper bound from Equation 2.1:

$$scr_{ub} = \alpha \times \frac{m'}{global\_max\_scr} + (1-\alpha) \times \left(1 - \frac{dist(r, q.loc)}{global\_max\_dist}\right), \tag{2.3}$$

where m' denotes the maximum static score of region r in n's child node, and $dist(r, q.loc)$ denotes the distance between region r and the query location. If the upper bound is no better than the k-th temporary result, we can skip all the objects between the starting and ending pointers of the child node in this region; i.e., all the objects specified by the subtree rooted as this child node. We leverage this property and devise a pruning algorithm (called *subtree level pruning*) as follows.

Algorithm 6 provides the pseudocode of the subtree level pruning. It replaces Lines 13 – 16 in Algorithm 5. Consider we are fetching objects in region r for node n. Instead of directly fetching from node n, we iterate through all its child nodes (denoted by n'). The entry with region ID equals to r in the region list of n' is identified (Line 2). We retrieve the corresponding maximum static score m', and compute a score upper bound (Line 3). If the upper bound is no greater than the score of the k-th temporary result, we skip node n'. Otherwise, we access the data object array and fetch the objects between the corresponding pointers s' and e' to update the temporary results (Lines 5 – 8).

Example 2.8. *Consider the objects in Table 2.1. The query string is* `nagoya`. *Its location is shown in Fig. 2.4. Suppose that* $k = 1$, *and* $\alpha = 0.5$. *Suppose* $global\_max\_dist = 1$. *The distances from the query to* o_2 *and* o_3 *are 0.3, 0.5, respectively.*

We follow the path `nagoya`, *the active node is node 6 in the trie. There is only region 5 in the node's region list. By subtree level pruning, we first process the first child – node 7 and find one temporary result* o_2, *whose score is 0.8. Then we process the second child – node 11. By Equation 2.3, the score upper bound is 0.65. It is smaller than the k-th temporary result's score. So node 11 is pruned, and we return* o_2 *as the final result.*

To apply subtree level pruning, one subtlety is to find the entry with region ID equals to r from the region list of n'. We first test the corresponding bit in the region bit array of n'. If the bit is 0, n' is skipped, meaning there is no region r in the region list. Otherwise, we scan its region list until the region ID r appears. The following optimization is applied while we scan the region list. Since the list has been sorted by descending order of maximum static score, for each m' we see in the list, no matter which region it corresponds to, we may compute a score upper bound by Equation 2.3 as if m' was the maximum static score of region r. If the upper bound is no greater than the k-th temporary result's score, we can stop the scan and skip node n' even if we have not seen region r yet.

One may notice that subtree level pruning can be carried out recursively; i.e., to go deeper in the trie and prune more objects using the region lists of node n's grand descendants. We choose not to do so because probing the region lists poses overhead, and our experiments show that conducting subtree level pruning only on n's children achieves a balance between pruning objects and list entry access.

2.5.3 Supporting Error-tolerant Features

Our method can be easily integrated into the existing solutions to error-tolerant query autocompletion; such as [19, 55, 57, 103]. We choose the trie-based method proposed in [19] for ease of illustration.

The basic idea of the method in [19] is to process the keystrokes in the query and compute a set of active nodes in the trie. The trie is exactly the same as ours except that we store spatial information on it. The path from the root to an active node is a string whose edit distance to the query is within the threshold τ. For example, suppose $\tau = 1$. Given a query string "`ni`" and the trie in Fig. 2.5, nodes 1, 2, and 21 are active nodes, because `n`, `na`, and `nu` are the strings whose edit distances to `ni` are 1. Therefore, we need to extend our method from single active node to the multiple active node case.

For range queries, we replace Line 4 in Algorithm 1 with the active node propagating method in [19]. In Algorithm 4, we change its input by replacing *n* with a set of active nodes, and fetch result for every active node inside the algorithm.

For top-*k* queries, we do the same replacement in Algorithms 3 and 4. In addition, an optimization technique is applied for faster result fetching. Since the region list of a node is sorted by descending maximum static score order, the first entry gives the maximum static score among all the underlying objects of the node. We choose to sort the nodes by the descending order of this value, and then process them one by one using Algorithm 4. For an active node *n*, we can compute an upper bound of its underlying objects from Equation 2.1:

$$scr_{ub} = \alpha \cdot \frac{max\_scr(n)}{global\_max\_scr} + 1 - \alpha, \tag{2.4}$$

where $max\_scr(n)$ denotes the maximum static score among the underlying objects of n. The upper bound is compared with the k-th temporary result. If it is no greater than the k-th temporary result's score, we can terminate the result fetching phase and output final results, because the remaining active nodes cannot produce any underlying object better than the current top-k objects. We call this optimization *active node level pruning*.

Before reporting experiment results, we briefly discuss the differences between our method and the MT method [1]:

- Although both methods use trie-based text-first indexes, we use region bit array for pruning to speed up query processing, while there is no such data structure or pruning technique in MT.

- In the indexing step, MT enumerates the query location across all the regions and stores the corresponding score upper bounds in the trie nodes. This drastically increases memory consumption, and thus MT has to materialize the score upper bounds for regions with coarser granularity and only a subset of trie nodes. Although this technique makes MT meet the space requirement, it compromises query processing performance. In contrast, our method materializes region lists without any compromise because the regions under most trie nodes are sparse (see Example 2.5).
- For query processing, we use pointers to locate the objects in the data object array to fetch results, while MT traverses the subtree rooted at the active node.

2.6 Experiments

We report the experiment results in this section.

2.6.1 Experiment Setup

The following algorithms are compared in the experiment.

- **Tregion** is our proposed method. It is based on a **trie** integrated with **region** information. Space is partitioned by a quadtree, and each leaf node is regarded as a region.
- **MT** is a trie-based text-first method for top-*k* queries [1].
- **PR-Tree** is a tightly-combined method that merges trie and quadtree into a single index [4]. It was designed for processing *k*nn queries.
- **INSPIRE** is a quadtree-based space-first method [3]. It was proposed for the spatial-textual query relaxation problem. It answers range queries in an error-tolerant manner.

For our method, we adjust the capacity in the quadtree node, and the number of resulting regions is no more than 64, which gives best query processing performance. We use the method proposed in [19] to process error-tolerant queries. For the other competitors, we make minor modifications so they can answer both range queries and top-*k* queries as well as error-tolerant queries. For MT, we use the same region setting as for Tregion, and materialize score upper

TABLE 2.4: Dataset Statistics

Dataset	$\|\mathcal{O}\|$	size	avg. string length
FSQ	1,021,447	31 MB	9.4
GNIS	2,193,355	67 MB	10.9
SGP	12,705,409	394 MB	11.5

bounds for all trie nodes. We do not compare with the FEH method [2] as it has been shown to be significantly outperformed by PR-Tree and INSPIRE [3, 4].

We select three publicly available datasets:

- **FSQ** is a dataset collected from Foursquare, which contains 1M worldwide points of interest.
- **GNIS** is a dataset of 2M geographic names collected from the U.S. Government Geographic Names Information System.
- **SGP** is a dataset of 13M records obtained from SimpleGeo's Places.

Table 2.4 shows statistics about the datasets.

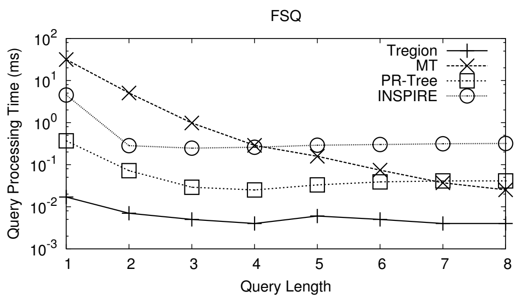
For each type of query, we generate 1,000 random queries by choosing strings that appear in the dataset. Longitude and latitude are normalized to [0, 1]. The default query range is a 0.08×0.08 square. Note that as we adopt quadtree instead of grids in our implementations, the number of objects located in a 0.08×0.08 square can be very small in sparse regions. The default value of k is 10.

We measure (1) average query response time, including both searching time and result fetching time, (2) index construction time, and (3) index size.

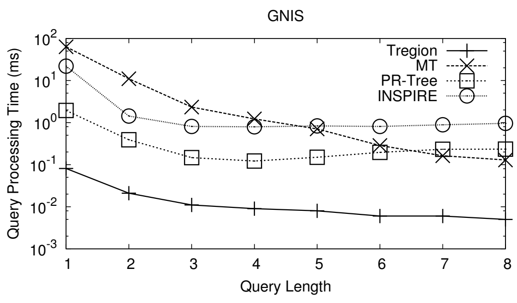
The experiments were carried out on a PC with an Intel i5 2.6GHz Processor and 32GB RAM, running Ubuntu 14.04.3. The algorithms were implemented in C++ and in a main memory fashion.

2.6.2 Range Queries

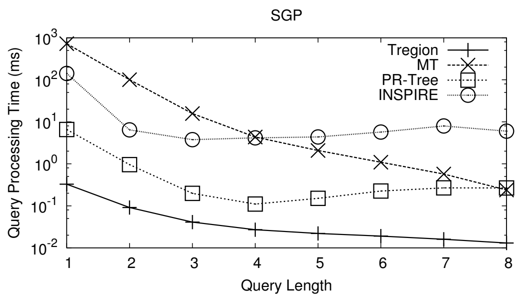
The performance of processing range queries is evaluated first. Fig. 2.6(a) – 2.6(c) show the query processing times of the four algorithms on the three datasets, varying query string length. Because the number of results decreases when the query becomes longer, the general trend



(a) FSQ



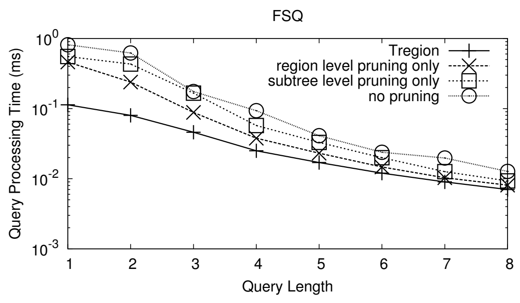
(b) GNIS



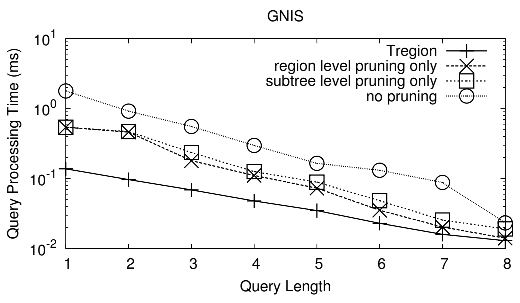
(c) SGP

FIGURE 2.6: Performance on range queries.

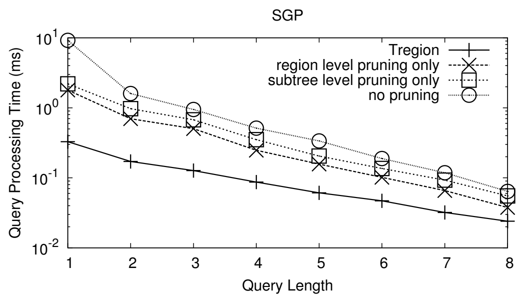
is that the query processing times decrease with the query string length, though PR-Tree and INSPIRE show some rebounds due to more traversal cost when the query is longer than 5 characters. Thanks to the region bit array, Tregion is always faster than the other competitors. The speedup can be up to one to two orders of magnitude. PR-Tree is the second fasters, and the INSPIRE is the third. We observe that the number of node access of PR-Tree is significantly higher than Tregion. E.g., on FSQ dataset, when the query string length is 4, the average numbers of node access of PR-Tree and Tregion are 154.3 and 3.3, respectively. The reason why INSPIRE is slow is that most query processing time is spent (e.g., 87.9% on the SGP dataset when the query string length is 2) on the text filter based on q-grams, whose frequencies are high for short strings (e.g., "an") and thus not selective. These results show the drawbacks of the tightly-combined method and the space-first method, hence justifying our analysis on the three types of methods in the beginning of Section 2.4. MT is the slowest in most cases, because it does not have any spatial filter in the searching phase and the spatial condition is checked when we fetch results.



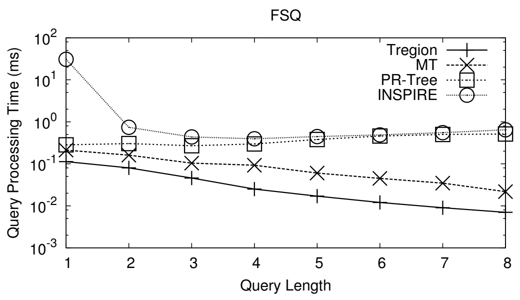
(a) FSQ



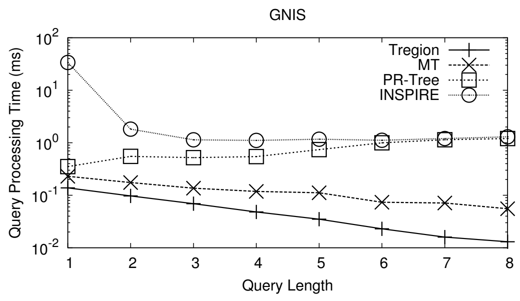
(b) GNIS



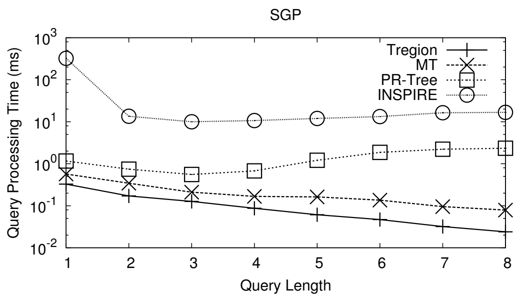
(c) SGP



(d) FSQ



(e) GNIS



(f) SGP

FIGURE 2.7: Performance on top-*k* queries.

2.6.3 Top-*k* Queries

For top-*k* queries, we first evaluate the effects of the pruning techniques. Fig. 2.7(a) – 2.7(c) show the result fetching times on the three datasets for the four algorithms: Tregion, Tregion with region level pruning only, Tregion with subtree level pruning only, and Tregion without the two pruning techniques. It can be observed that both pruning techniques effectively decrease the query processing time of Tregion by 2 to 28 times, and they are more effective for short queries. The comparison with the other methods on the three datasets are shown in Fig. 2.7(d) – 2.7(f). Due to the algorithm design for top-*k* queries and the effects of two pruning techniques, Tregion is the fastest of the four algorithms, and it is up to 4 times faster than the runner-up MT. PR-Tree is the third and INSPIRE is the slowest method. Both are slower than Tregion by one to two orders of magnitude. The drawbacks of the tightly-combined method and the space-first method are also seen on top-*k* queries. E.g., on FSQ dataset, when the query string length is 4, PR-Tree accesses 289.3 nodes on average while Tregion accesses 14.2 nodes. For INSPIRE, when query string length is 2 on the SGP dataset, 90.5% query processing time is spent on the text filter, and thus the query processing speed becomes slow.

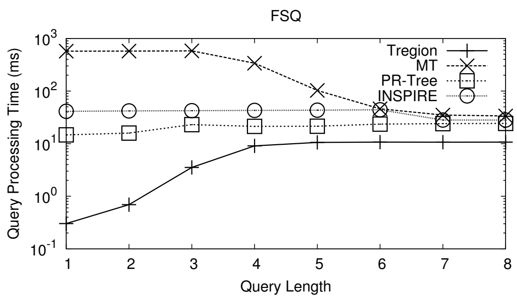
2.6.4 Error-tolerant Queries

We enable the error-tolerant feature and show the query processing times of range queries in Figs. 2.8(a) – 2.8(c). The edit distance threshold is 3. The query processing time of Tregion increases when more characters are input. The reason is that we have more traversal in the trie to tolerate errors, and it increases the overall cost when the query becomes longer. Nonetheless, Tregion is the fastest among the four algorithms, and the speedup can be up to two orders of magnitude.

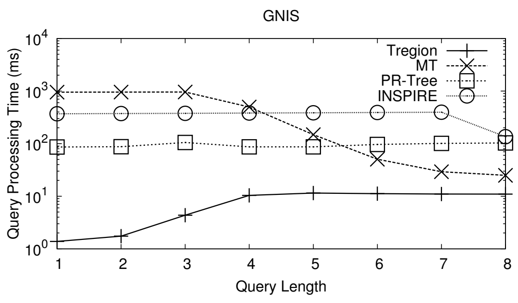
For error-tolerant top-*k* queries, we first evaluate the effect of active node level pruning and show the results in Fig. 2.8(d) – 2.8(f). The active node level pruning is more effective for short queries, because there are more active nodes for the first few characters and hence more chance to prune. The comparison with the competitors on error-tolerant top-*k* queries is shown in Fig. 2.8(g) – 2.8(i). Tregion is the fastest algorithm, and the runner-up is MT. Both are faster than PR-Tree and INSPIRE by a remarkable margin (up to 9 times).

2.6.5 Scalability

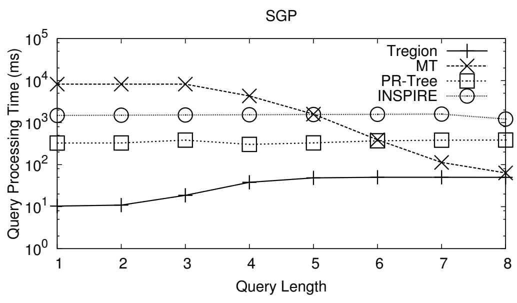
We evaluate the scalability of the algorithms with varying dataset size. 20% to 100% objects are sampled from the largest dataset SGP. Note that as SGP contains 13M POIs which is sufficiently large, it is convincing to sample and use it in the scalability evaluations. Figures 2.9(a) and 2.9(b) show the range query and top-*k* query performances on the SGP dataset, respectively. The query processing times increase with the dataset size for the four algorithms. Tregion is always the fastest, and it has slower growth rate than the other three competitors.



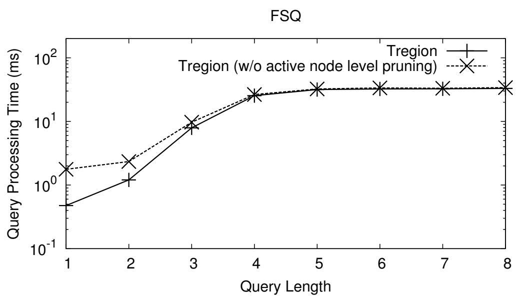
(a) Range Query - FSQ



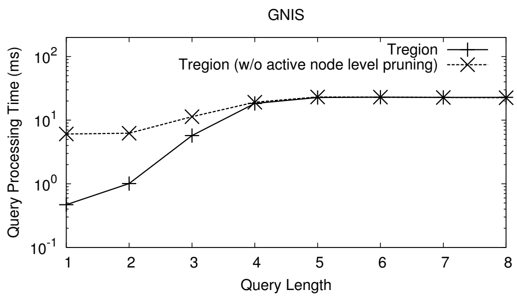
(b) Range Query - GNIS



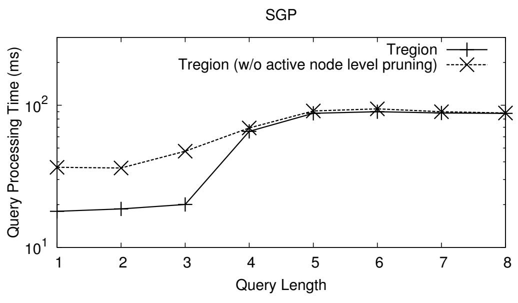
(c) Range Query - SGP



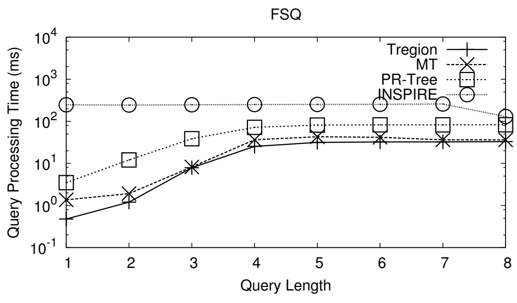
(d) Top-*k* Query - FSQ



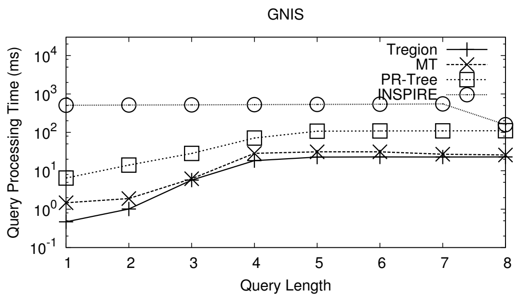
(e) Top-*k* Query - GNIS



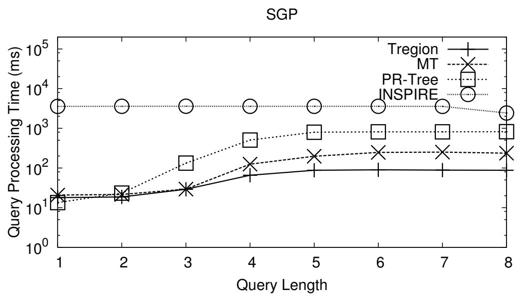
(f) Top-*k* Query - SGP



(g) Top-*k* Query - FSQ

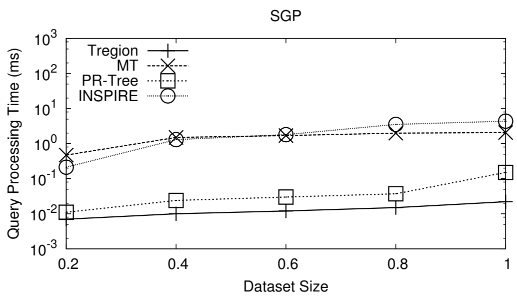


(h) Top-*k* Query - GNIS

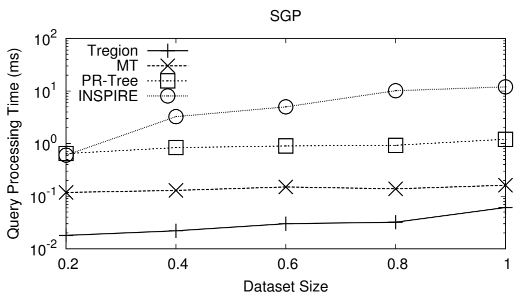


(i) Top-*k* Query - SGP

FIGURE 2.8: Performance with error-tolerant feature.



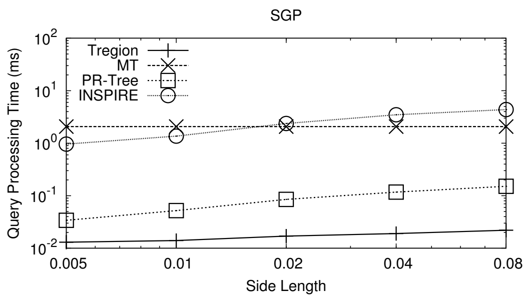
(a) Range Query - SGP



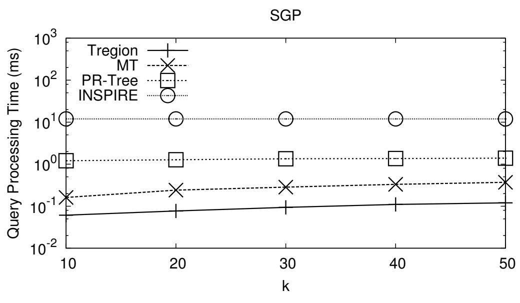
(b) Top-*k* Query - SGP

FIGURE 2.9: Performance when varying dataset size.

We also test the performances of the four algorithms when varying range for range queries or *k* for top-*k* queries. Figures 2.10(a) shows the query processing times for the following range queries: 0.005×0.005, 0.01×0.01, 0.02×0.02, 0.04×0.04, and 0.08×0.08. MT's query processing time is regardless of the query range because it does not have any spatial filter for range queries. The other three algorithms exhibit increasing query processing time when the query range expands. This is expected as there are more tree nodes satisfying the spatial condition and the number of results also increases. Figure 2.10(b) shows the query processing times for the following *k* values: 10, 20, 30, 40, and 50. INSPIRE has almost constant query processing time when *k* varies because it was designed for range queries, lacking specific filters for top-*k* queries. For the other three algorithms, the running times slightly increase when *k* moves towards larger values. Tregion is always the fastest among the four competitors.



(a) Range Query - SGP



(b) Top-*k* Query - SGP

FIGURE 2.10: Performance when varying range or *k*.

TABLE 2.5: Index Size (GB)

Dataset	Tregion	MT	PR-Tree	INSPIRE
FSQ	1.4	8.6	0.5	0.5
GNIS	1.6	8.9	1.0	0.9
SGP	13.4	32.0	5.3	6.3

TABLE 2.6: Index Construction Time (seconds)

Dataset	Tregion	MT	PR-Tree	INSPIRE
FSQ	11.3	81.4	4.7	12.7
GNIS	17.5	141.1	9.0	23.2
SGP	178.5	958.1	54.4	190.6

2.6.6 Index Construction

Table 2.5 shows the index sizes of the four algorithms on the three datasets. Table 2.6 shows the corresponding index construction times. MT has the largest index size due to its enumeration of query locations. Tregion's index size is the second among the four, but is much smaller than MT. PR-Tree and INSPIRE have similar index sizes. All of the four algorithms are able to build index in reasonable amount of time. MT is the slowest of the four. Tregion spends less time than INSPIRE but more time than PR-Tree.

2.7 Conclusion

In this chapter, we proposed a novel method for location-aware query autocompletion. We aimed at answering range and top-k queries on a large scale. We proposed a method by which data objects are indexed in a trie integrated spatial information. Several pruning techniques were proposed to further improve the query processing performance. We also discussed how to extend our method to support the error tolerant feature. The experiment results demonstrate the efficiency of the proposed method and its superiority over existing methods.

Chapter 3

Autocompletion for Prefix-Abbreviated Input

3.1 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 [15, 17, 38] and database research communities [1, 19, 25, 54, 59].

For existing QAC methods [15, 17, 19, 25, 38] (including type-ahead search [21, 54] 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 $\underline{\texttt{fusospirochetal}}$ $\underline{\texttt{gingivitis}}$, where morphemes are separated by underline.

In this work, 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 [104], 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 [75]. 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 [5]. 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

[1]In this work, we assume to perform autocompletion over a pre-defined dictionary of data strings, in line with [19].

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 3.2.2) 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 [105].

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 3.4.2) 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 this chapter is organized as follows: Section 3.2 defines the problem and introduces preliminaries. Section 3.3 presents the index structure. The query processing algorithm appears in Section 3.4. Section 3.5 introduces the ranking method and the algorithm for fast top-k retrieval. Section 3.6 discusses miscellaneous extensions, including skipping keywords, non-prefix abbreviations, full-text search, and data updates. Experimental results and analyses are reported in Section 3.7. Section 3.8 reviews related work. Section 3.9 concludes the chapter. Section 3.10 gives the detailed proofs. Section 3.11 shows more additional experiments.

3.2 Preliminaries

3.2.1 Problem Definition

Σ is 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 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 $s_1 = s_2[1\,..\,i]$, $1 \leq i \leq |s_1|$. We also use the notation $\overleftarrow{s}$

TABLE 3.1: 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

to denote any prefix of s. s_1s_2 denotes the concatenation of s_1 and s_2. An array $[s_1,\dots,s_n]$ ($n \geq 1$) is called a segmentation of s, iff $s = s_1s_2\dots s_n$. 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 Σ consists of English letters only and use *capital letters* to denote the initial characters of keywords. Table 3.1 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 $[s_1,\dots,s_n]$. Given a query string q, we say q is a *prefix-abbreviated match* of s, denoted by $q \sqsubseteq s$, iff $q = \overleftarrow{s_1}\,\overleftarrow{s_2}\dots\overleftarrow{s_i}$, $1 \leq i \leq n$; 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 work, we use the term "**PA-match**" short for prefix-abbreviated match, while the 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, $[\overleftarrow{s_1},\dots,\overleftarrow{s_i}]$. So users do not have to explicitly specify where to segment the query. Next we define our problem.

[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.

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 3.1 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.

3.2.2 Baseline Methods

Most prevalent QAC methods [15–17, 21, 25, 54, 89] adopt a trie index to process queries. For QACPA, we may also index data strings in a trie and design a baseline algorithm. Figure 3.1 shows the trie for the strings in Table 3.1. String IDs are linked to the end of strings. The query processing algorithm (pseudo-code in Algorithm 7) 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 3.1. *Consider a query string $q =$ `geneva`. Table 3.2 shows the active nodes (numbered in Fig. 3.1) 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:

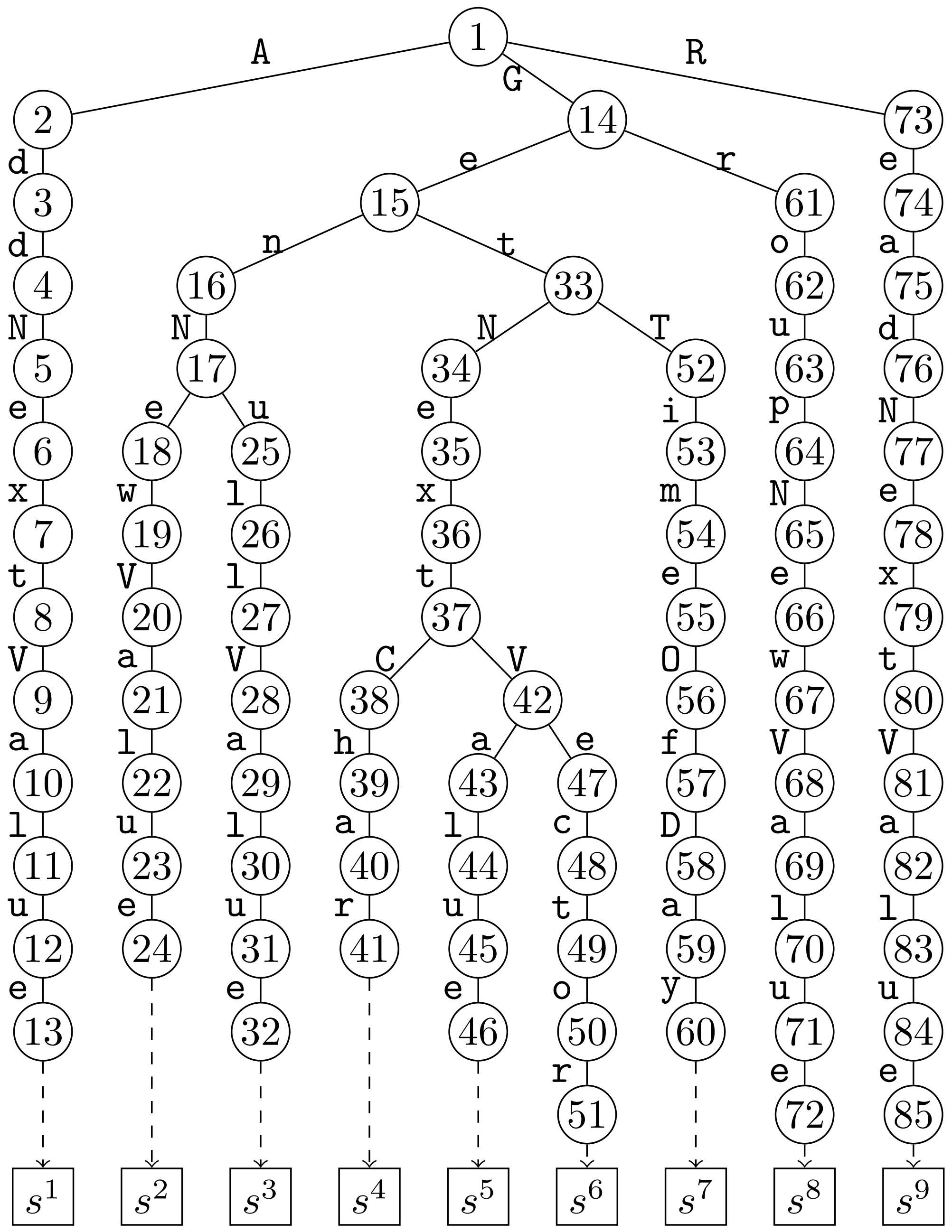


FIGURE 3.1: Trie index for the baseline method.

- 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 3.1, 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

Algorithm 7: 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 3.2: Active nodes by the baseline algorithm.

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

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 3.1, 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 [54] or multi-dimensional substring search [86, 87] 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 [20, 54] (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 [89] 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 [90, 91], 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 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 [88], and then check if they satisfy the prefix-abbreviated condition. But this method only applies to small datasets since the space complexity is $O((|S|+|\Sigma|)N)$, where N is the sum of string lengths in S.

3.3 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.

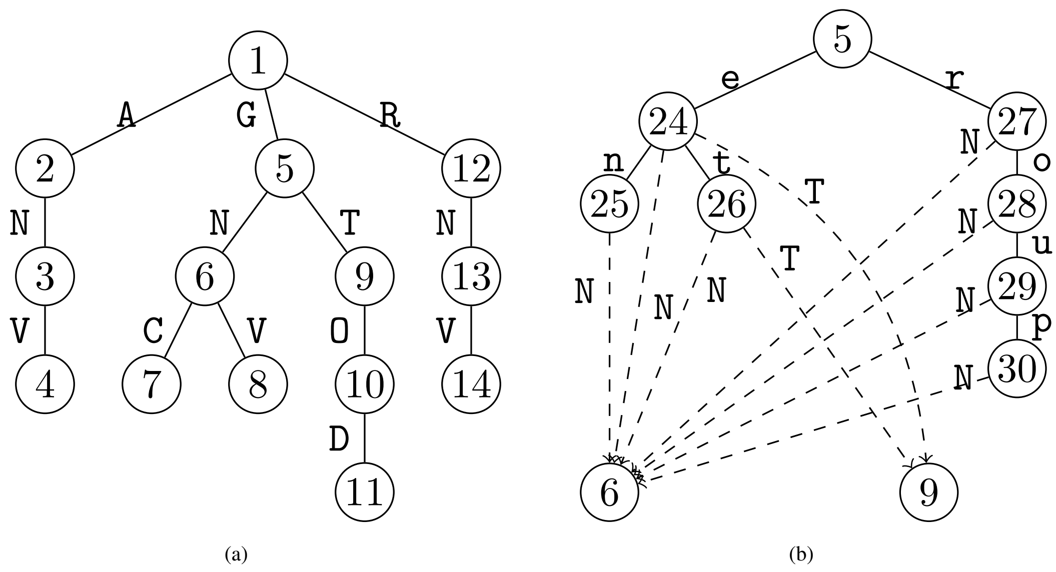


FIGURE 3.2: The outer trie (a) and an inner trie rooted at node 5 (b). Shortcuts are shown in dashed links.

Example 3.2. *For the strings in Table 3.1, we first collect the initial characters of keywords:* `ANV`*,* `GNC`*,* `GNV`*,* `GTOD`*, and* `RNV`*. Figure 3.2(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 Fig. 3.2(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 Fig. 3.2(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 Fig. 3.2(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 3.1 is shown in Fig. 3.3.

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.

3.4 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.

3.4.1 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 8). 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 3.3. *Consider the query string* `geneva` *in Example 3.1. Table 3.3 shows the active nodes, as numbered in Fig. 3.3, for each keystroke using the nested trie-based algorithm. For keystroke* `n`*, we jump from node 24 to 6 by a shortcut, which refers to the outer edge from*

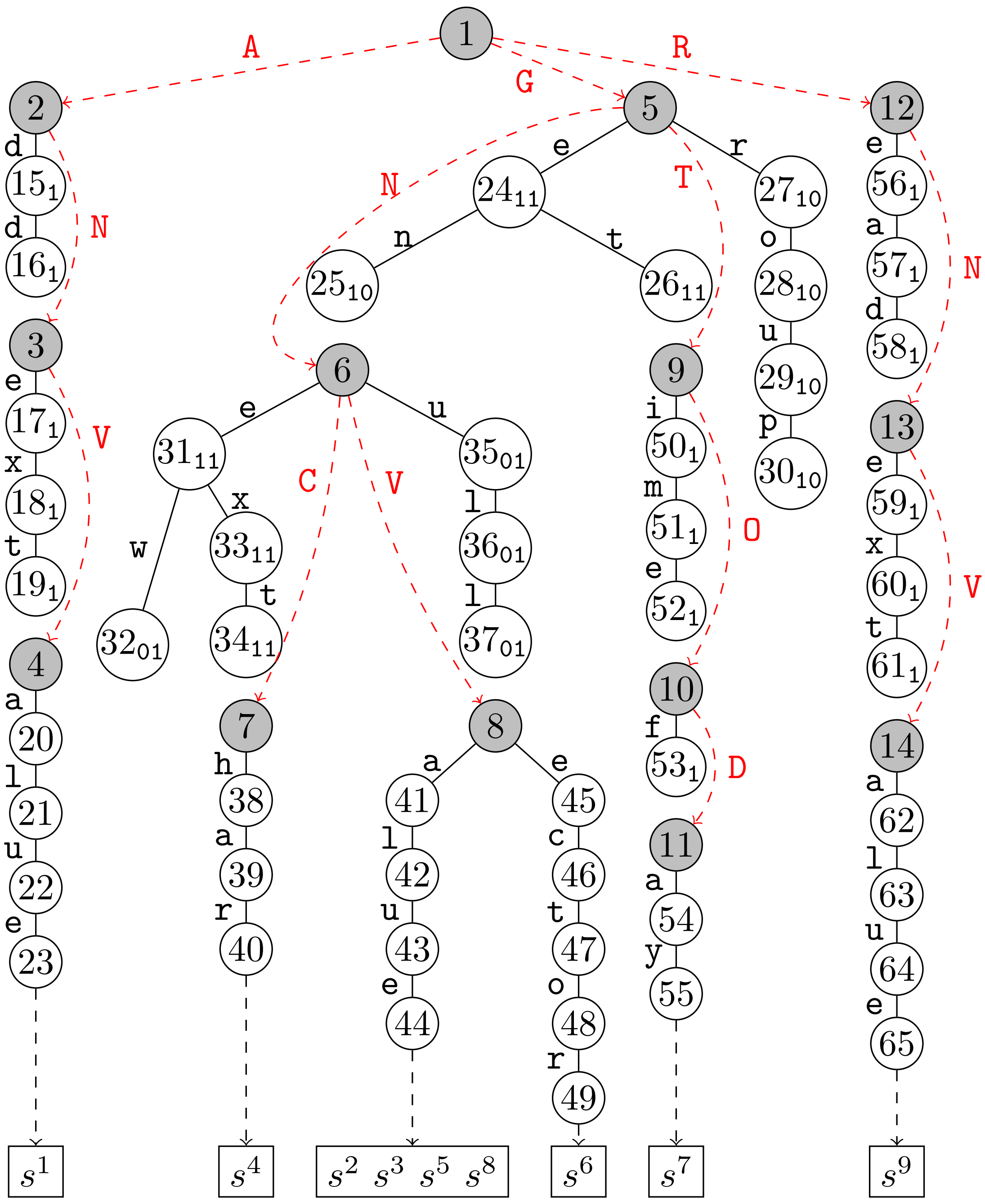


FIGURE 3.3: A nested trie. Outer nodes are marked in grey. Outer edges are shown in red dashed links. Inner edges are shown in black solid lines. Inner node numbers are subscripted by bit vectors for shortcuts, if there is any.

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 8, we have one active node for the first keystroke, and each existing active node generates at most two new

Algorithm 8: QACPA-Nested-Trie-Search (q, T)

```
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 inner edge $q[i]$ then
6              $A' \leftarrow A' \cup \{n'\}$ ;          /* 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                  $A' \leftarrow A' \cup \{n'\}$ ;          /* for initial character */
10         else
11             if $n$ has a shortcut $q[i]$ to $n'$ then
12                 $A' \leftarrow A' \cup \{n'\}$ ;          /* for initial character */
13     $A \leftarrow A'$;
```

TABLE 3.3: Active nodes by the nested trie-based algorithm.

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

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 8, but we need an additional cost to report correct results, as explained in the rest of this section.

3.4.2 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 3.3, the query string `geneva` has node 41 as its active node, but it PA-matches only two of the four underlying 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 Fig. 3.3 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 $1 \rightarrow 5 \rightarrow 24 \rightarrow 26 \Rightarrow 6 \rightarrow 31$, where $\rightarrow$ is via an edge and $\Rightarrow$ 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., $\{x_1,\ldots,x_m\} \otimes \{y_1,\ldots,y_n\} = \{x_i \cap y_j \mid 1 \leq i \leq m \wedge 1 \leq j \leq n \wedge x_i \cap y_j \neq \emptyset\}$, where each x_i or y_i denotes an interval. We have the following property.

Proposition 3.1. *Consider a path $n_1,\ldots,n_k$ from the root of the nested trie to an active node n_i. All the strings in $I_{n_1} \otimes I_{n_2} \ldots \otimes I_{n_k}$ 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 3.2. *Given a dataset S and a query q, for any keystroke of q, the number of active nodes produced by Algorithm 8 is always less than or equal to that produced by Algorithm 7.*

The pseudo-code of the above process is given in Algorithm 9. 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 8, it computes $J_{n'}$ by merging J_n and $I_{n'}$ (Line 4). To implement Algorithm 9, we integrate it into Algorithm 8 by placing Lines 4 – 6 of Algorithm 9 after Lines 5, 8 and 11 of Algorithm 8. Next we show how this works with an example.

Algorithm 9: QACPA-Nested-Trie-MonitorList (q, T)

```
1 $J_{root} \leftarrow [1, |S|]$;
2 foreach keystroke $q[i]$ do
3     if Algorithm 8 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 $A'$;
```

Example 3.4. *Consider Example 3.3. 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 $\rightarrow$ 5 $\rightarrow$ 24 $\Rightarrow$ 6 $\rightarrow$ 31 $\Rightarrow$ 8 $\rightarrow$ 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'	List of Intervals $I_{n'}$	Merged Result $J_{n'}$
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	$\{[2,3],[5,5],[8,8]\}$	$\{[2,2],[5,5]\}$

3.4.3 Optimizing List Merge

We may use the sweep line algorithm [106] 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 3.3 (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 $n, n_1, \ldots, n_k$ in the nested trie such that there is no shortcut in $n_1, \ldots, n_k$, then the result of list merge $J_n \otimes I_{n_1} \ldots \otimes I_{n_k} = J_n \otimes I_{n_k}$; 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 9 is implemented using this optimization. The pseudo-code is given in Algorithm 10. If n and n' are not connected by a shortcut, we continue Algorithm 9 until a shortcut is encountered (Line 5). Then we record J_n in a temporary list J (Line 8), and continue Algorithm 9 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 (Lines 2, 7, and 10). This optimization saves us most merge operations, as shown by this example:

Example 3.5. *Recall Example 3.4. 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

Algorithm 10: MergeLists ($J_n, I_{n'}$)

```
1  if $q[i]$ is the last character of $q$ then
2      return $J_n \otimes I_{n'}$
3  else
4      if $n$ and $n'$ are not via a shortcut then
5          Continue Algorithm 9
             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 $J_n \otimes I_{n'}$
8      $J \leftarrow J_n$;
9      Continue Algorithm 9 until $q[i]$ is the last character of $q$ or $n'$ has a shortcut $q[i+1]$;
10     return $J \otimes I_{n'}$
```

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 3.2. 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 3.2 and at the same time skip any post-shortcut list merge or any equivalent/weaker operation (such as using Bloom filter [107]) 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 3.7.3.

3.5 Ranking and Top-k Result Fetching

3.5.1 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 $[s_1, \ldots, s_n]$, 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 $[q_1, \ldots, q_m]$ such that $q_i \preceq s_i$, $1 \leq i \leq m \leq n$. For ease of exposition, we add $(n-m)$ empty strings, denoted by $q_{m+1}, \ldots, q_n$, 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 $[q_1, \ldots, q_n]$ and $[s_1, \ldots, s_n]$, denoted by $score(s,q) = P(s_1 \ldots s_n \mid q_1 \ldots q_n)$. 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

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

The denominator $P(q_1 \ldots q_n)$ 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 $P(q_1 \ldots q_n \mid s_1 \ldots s_n)$, we assume that $P(q_i \mid s_i)$, $1 \leq i \leq n$, are independent[3]. Thus, $P(q_1 \ldots q_n \mid s_1 \ldots s_n) = P(q_1 \mid s_1) \cdot \ldots \cdot P(q_n \mid s_n)$. We have

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

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 $m < i \leq n$. 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.

[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.

To evaluate $P(q_i \mid s_i)$, $1 \leq i \leq m$, 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., 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 [108] 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 3.6. *Consider the data strings in Table 3.1 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 3.1. 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

3.5.2 Efficient Top-k Result Fetching

Recall in Algorithm 9, 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 3.4.2. We have the following property.

Proposition 3.4. *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 11). It initializes a priority queue R for top-k results (Line 1). For each active node n, it sorts the intervals in J_n

Algorithm 11: QACPA-Nested-Trie-TopK (q, A, k)

```
1  $R \leftarrow \emptyset$ ;                                /* a priority queue of size $k$ */
2  foreach $n \in A$ do
3      Sort the intervals in $J_n$ using the maximum scores of $I_n$;
4      foreach $[u,v] \in J_n$ do
5          if $[u,v].max\_score \le R[k].score$ then
6              break;
7          foreach $i \in [u,v]$ do
8              if $|R| < k$ or $score(s^i, q) > R[k].score$ then
9                  $R.insert(s^i)$;
10 return $R$
```

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 3.4. 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 11. To exploit the keyword sharing effectively, we sort the strings in S by the lexicographical order.

Example 3.7. *Consider Example 3.6. 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 3.6, 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.*

3.6 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.

3.6.1 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 3.3. 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.

3.6.2 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 $[q_1, \dots, q_m]$ of q, such that $\forall i \in [1\,..\,m]$, (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 $\forall j \in [1\,..\,|q_i|]$, 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.

3.6.3 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 3.4) 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.

3.6.4 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 3.3. 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 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 [84]. We may also periodically reconstruct the index. This is similar to many information retrieval solutions.

TABLE 3.4: Statistics of datasets.

Dataset	$\|S\|$	**Max.** $\|W\|$	**Avg.** $\|W\|$	**Max.** $\|s\|$	**Avg.** $\|s\|$	**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

3.7 Experiments

We report the most important experimental results here. Please see Appendix 3.11 for the experiments on scalability, round-trip time, updates, index, and extensions.

3.7.1 Experiment Setup

In the experiments, we use the following datasets that cover the four applications listed in Section 3.1.

- **JAVA** is a dataset of API names of the Awesome Java Project on GitHub [109]. APIs are segmented by capital letters. Popularities are collected from the source codes of 12 projects [110].
- **PINYIN** is a dataset of Sougou cloud pinyin dictionary [111]. Words are segmented by Chinese syllables. We use the word frequencies in [112] as popularities.
- **UNIX** is a dataset of files in a UNIX archive [113] at ICM, Poland. Paths and extensions are removed. Filenames are segmented into keywords using Python WordSegment [114]. We use the term frequencies in the dataset as popularities.
- **ALLIE** is a dataset of terms extracted from MEDLINE [104]. Strings are segmented into morphemes using Morfessor [115]. We use the term frequencies in MEDLINE as popularities.

Table 3.4 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 [20, 54] 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 work.

The multi-dimensional substring search methods [86, 87] 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 [89–91] or subsequence search [88]. The reasons (equivalence to Trie/prohibitive index size) have been explained in Section 3.2.

For ranking methods, we compare with the most popular completion method (MPC) used in previous work [15–17]. 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.

3.7.2 Evaluation of Effectiveness

We first compare the keystrokes of QACPA with traditional QAC. Table 3.5 reports the results with navigation, i.e., the numbers of characters entered before the intended string appears

TABLE 3.5: 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 3.6: 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%)

TABLE 3.7: Mean reciprocal rank (in percentage).

Dataset	k	$\lvert q\rvert = 2$		$\lvert q\rvert = 4$		$\lvert q\rvert = 6$		$\lvert q\rvert = 8$	
		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

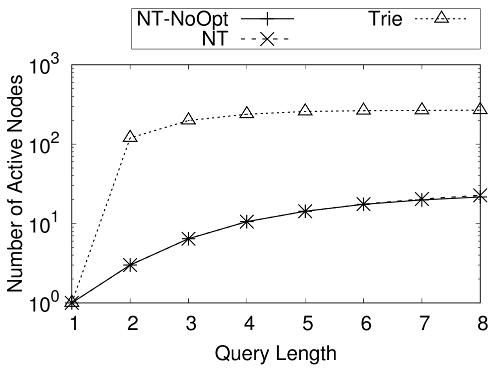
in the top-k suggestions plus the numbers of arrow keys needed to navigate in the top-k list. Table 3.6 reports the results without navigation, i.e., only the numbers of input characters. The results are averaged over 1,000 queries when $k = 5$ or 10. We also show the percentage of keystrokes saved from 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 rates in Appendix 3.11.1. 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 3.7. We also show the relative MRR improvement over MPC. Note that even a small absolute difference in MRR could lead to considerable performance gain [17, 38]. 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.

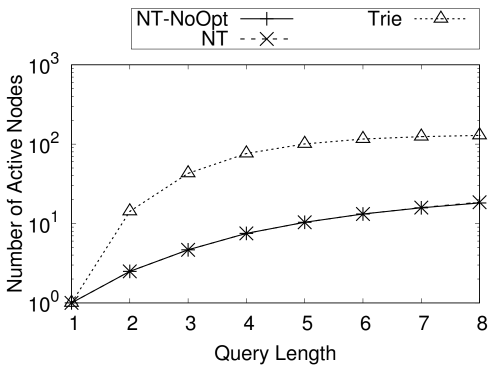
To compare k settings, $k = 10$ saves more keystrokes and yields a better MRR, but $k = 5$ is also acceptable. Considering the numbers of available suggestions in different applications, we suggest using $k = 10$ for Web search and text editors, and $k = 5$ for mobile applications.

3.7.3 Evaluation of Efficiency

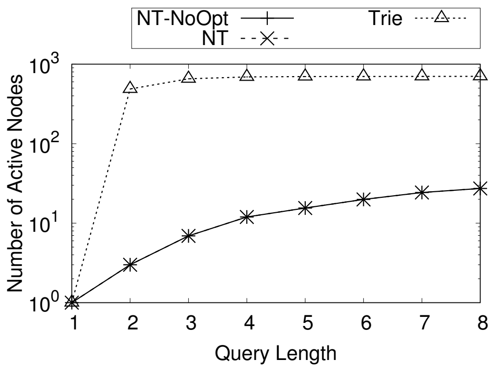
For efficiency, we first evaluate the searching phase of the query processing. We vary the query length and plot the average numbers of active nodes per query in Fig. 3.4(a) – 3.4(d). The results are accumulated, i.e., every active node en route is counted towards the number. So it increases with the query length. We also plot the results of our algorithm without the optimization of skipping list merge (Proposition 3.3), 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



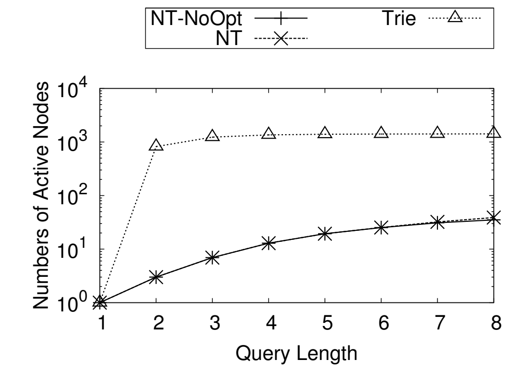
(a) JAVA



(b) PINYIN



(c) UNIX

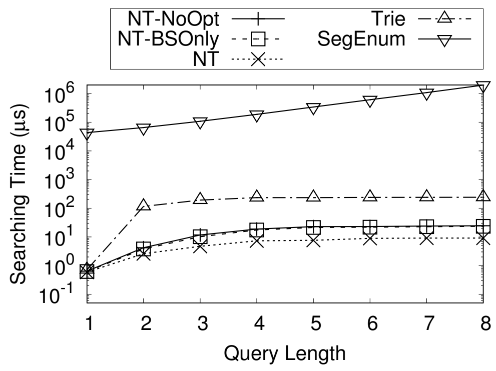


(d) ALLIE

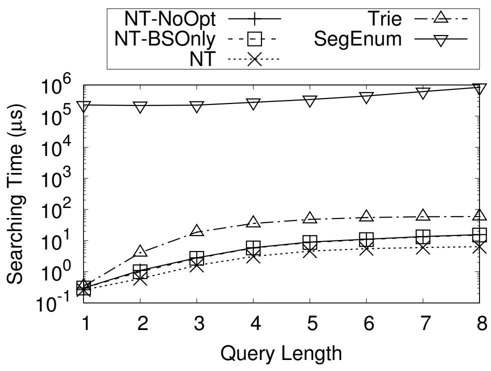
FIGURE 3.4: Number of active nodes.

and achieves a smooth increment w.r.t the query length. The gap 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.

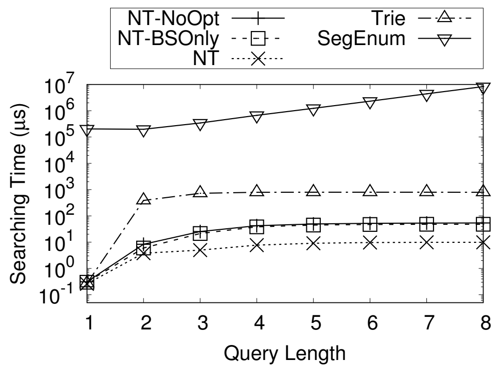
The times of the searching phase with varying query length are reported in Fig. 3.5(a) – 3.5(d). To show the effect of the optimizations on list merge, we also plot the performance of NT without any optimization (referred to as NT-NoOpt) or with the binary search optimization only (referred to as NT-BSOnly). 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



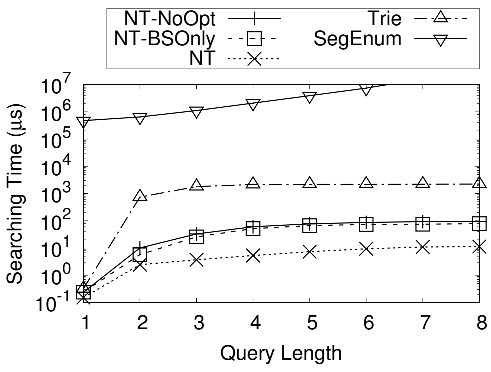
(a) JAVA



(b) PINYIN



(c) UNIX



(d) ALLIE

FIGURE 3.5: Searching time.

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 $k = 10$. Fig. 3.6(a) – 3.6(d) show the result fetching times with varying query length. The times decrease for longer queries, because 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

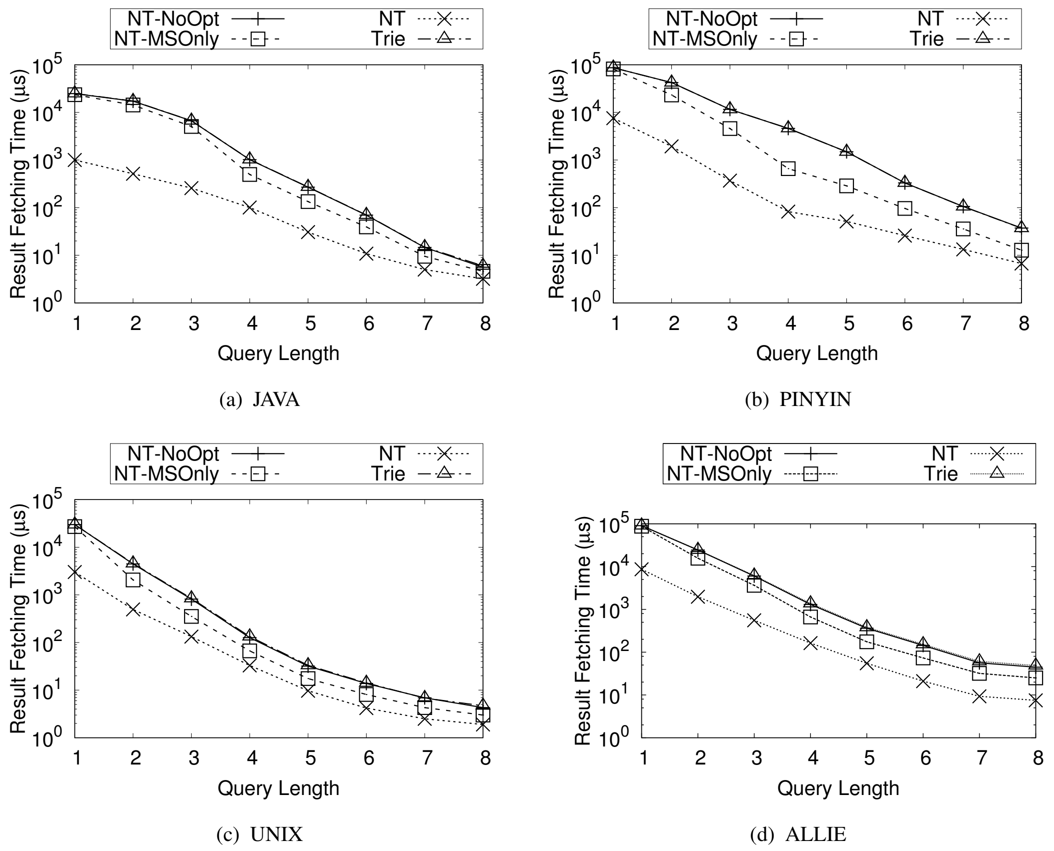
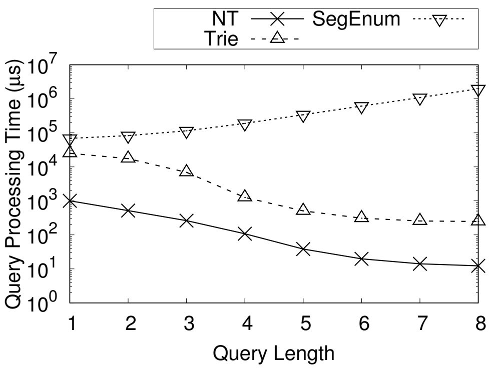


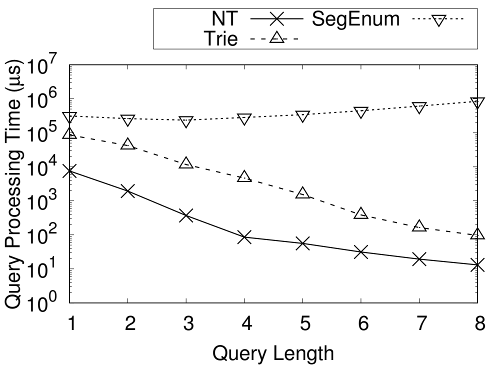
FIGURE 3.6: Result fetching time.

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 3.11.2.

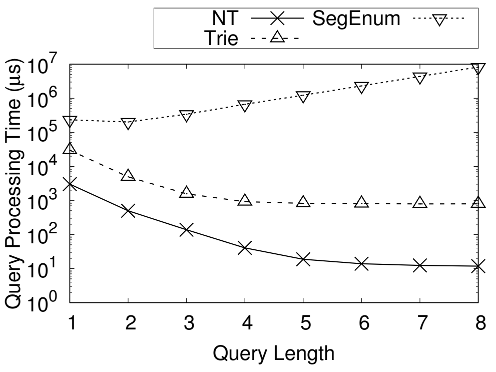
The overall query processing times ($k = 10$) are plotted in Fig. 3.7(a) – 3.7(d). The overall trend is the query processing time decreases with the query length for Trie and NT but increases for SegEnum. The reason is the proportions of searching and result fetching are different; e.g., for short queries, 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 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.



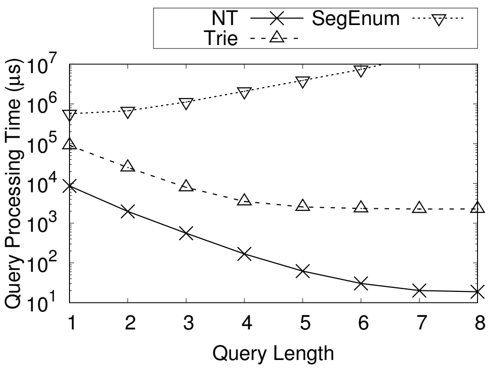
(a) JAVA



(b) PINYIN



(c) UNIX



(d) ALLIE

FIGURE 3.7: Overall query processing time.

3.8 Related Work

Darragh and Witten [22] developed the reactive keyboard to predict future keystrokes based on past interactions. Other early studies considered completing the query at word [23] or phrase level [24, 25]. Fan *et al.* [26] studied the suggestion on topic-based query terms. Bhatia *et al.* [27] investigated the case when query logs are absent. To help users quickly identify useful results, Jain and Mishne [28] proposed to cluster suggestions and label them. Recent trends feature a boom in context-aware QAC where user interactions are important [15, 29–37], as well as plenty of work on presenting time-sensitive [16, 38–41], personalized [17, 42, 43], or diversified results [44, 45]. Besides these studies, Mitra and Craswell investigated the case for rare prefix [46]. Cai and de Rijke proposed to use homologous queries and semantically related terms [47]. Zhang *et al.* targeted the QAC for mobile devices [48]. There are also investigations in the direction of trie compression techniques to seek space efficiency [49, 50]. As for the quality of QAC, an experimental evaluation was reported in [51] to compare various ranking methods, including the traditional method by static score (past popularity), context-aware/user-interactive ranking, and learning to rank [52, 53]. Markov model was also adopted [33] to produce more meaningful results. In the database research community, Li *et al.* designed the TASTIER system for type-ahead search on relational data [54]. 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 [19–21, 55–57] or Markov n-gram transformation model [58]. Cetindil *et al.* [59] proposed a ranking method for error-tolerant QAC using proximity information. Another popular direction is location-aware QAC [1–4, 12], which is useful for navigation tools. Besides, there are studies on QAC for specific applications, e.g., command line shells [69, 70], Chinese pinyin input [71, 72], IDEs [5, 7, 18, 73, 74], and medical vocabulary [75].

3.9 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.

3.10 Proofs

3.10.1 Proposition 3.1

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 8, $|q| = k-1$ and $t[i]$ matches $q[i], \forall i \in [1..|q|]$.

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

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

3.10.2 Lemma 3.2

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 8. By Proposition 3.1, 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 12 misses s as a result. Therefore the end of this path is an active node in Algorithm 12. By the definition of f, f maps this node to n'. Because f is surjective, the number of active nodes by Algorithm 8 is at most equal to that by Algorithm 12. □

3.10.3 Proposition 3.3

Proof. Property 1: Let $X = \{x_1, \dots, x_m\}$, $Y = \{y_1, \dots, y_n\}$, and $Z = \{z_1, \dots, z_o\}$. 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. □

3.11 Additional Experiments

3.11.1 Success Rates

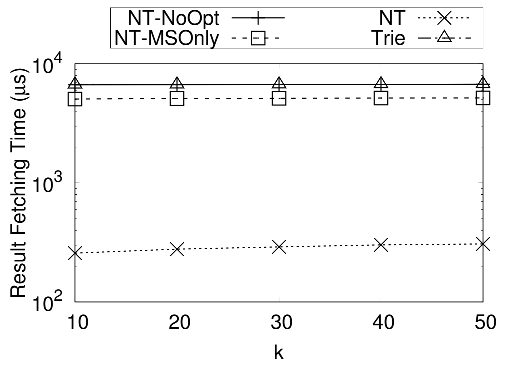
Table 3.8 reports the success rates @k (i.e., whether the intended string appears in the top-k results) of APP and MPC for top-1, top-2, and top-3 suggestions. We also show the relative improvement of APP over MPC. Similar to the results of MRR (Table 3.7), APP performs better than MPC for almost all 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.

3.11.2 Result Fetching w.r.t. k

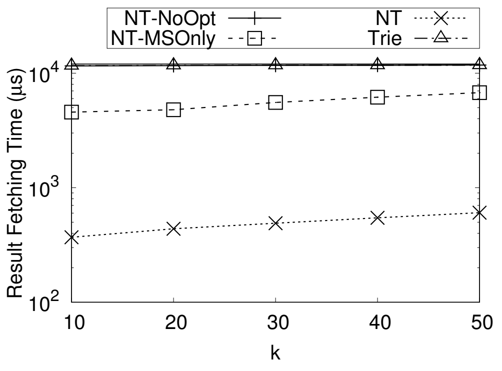
Figures 3.8(c) – 3.8(d) show the result fetching times w.r.t. k. The query length is 3. For Trie, we witness approximately constant time, as it has to scan all the underlying strings of the

TABLE 3.8: Success rate (in percentage).

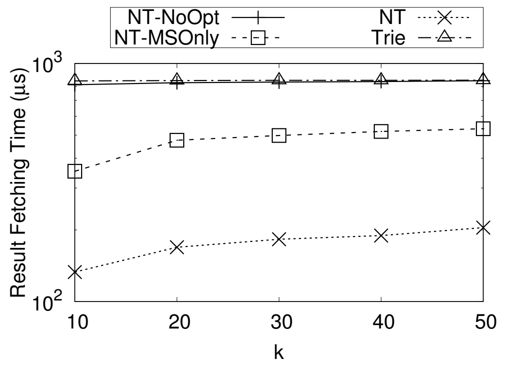
Dataset	k	$\lvert q \rvert = 2$		$\lvert q \rvert = 4$		$\lvert q \rvert = 6$		$\lvert q \rvert = 8$	
		APP	MPC	APP	MPC	APP	MPC	APP	MPC
JAVA	1	0.40 (+100%)	0.20	12.80 (+0%)	12.80	46.00 (+5.02%)	43.80	81.70 (+0.12%)	81.60
	2	0.40 (+100%)	0.20	26.90 (+10.24%)	24.40	68.30 (+3.01%)	66.30	87.20 (+0.23%)	87.00
	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



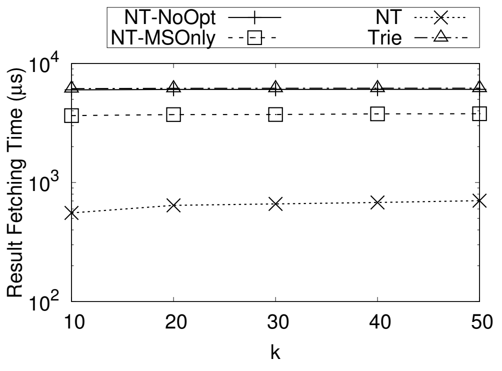
(a) JAVA



(b) PINYIN



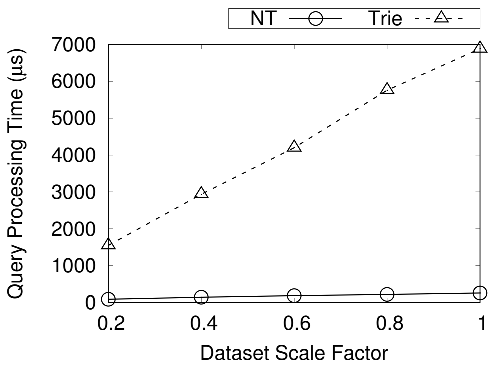
(c) UNIX



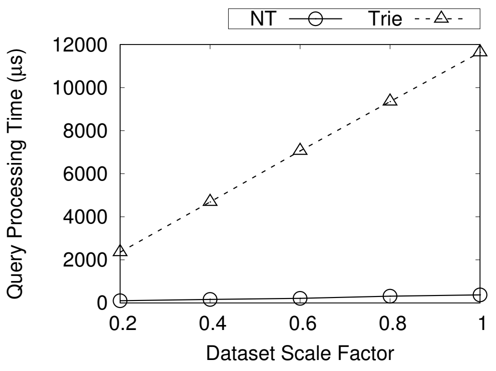
(d) ALLIE

FIGURE 3.8: Result fetching time w.r.t. k.

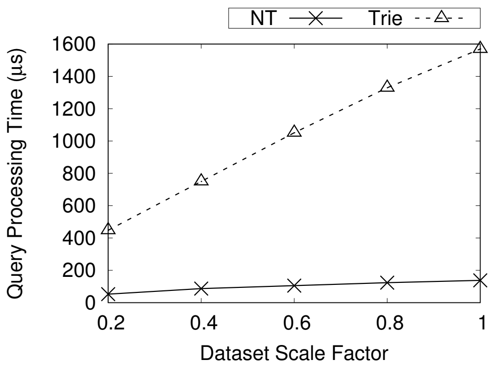
active nodes. For NT, we witness moderate increase (about 1.5 times) when k moves from 10 to 50. The superiority over Trie remains.



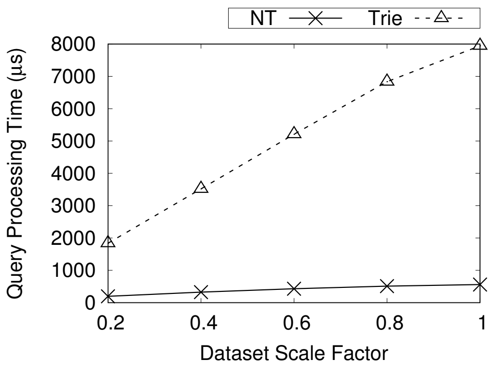
(a) JAVA



(b) PINYIN



(c) UNIX



(d) ALLIE

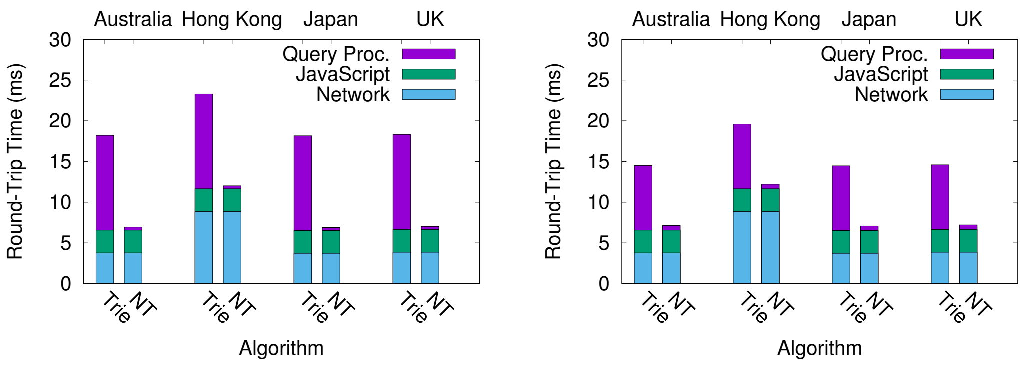
FIGURE 3.9: Scalability.

3.11.3 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 3.9(c) – 3.9(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.

3.11.4 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



(a) PINYIN (b) ALLIE

FIGURE 3.10: Round-trip Time.

over 1,000 queries are plotted in Figures 3.10(a) – 3.10(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.

3.11.5 Evaluation of Index

Table 3.9 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 3.9: 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

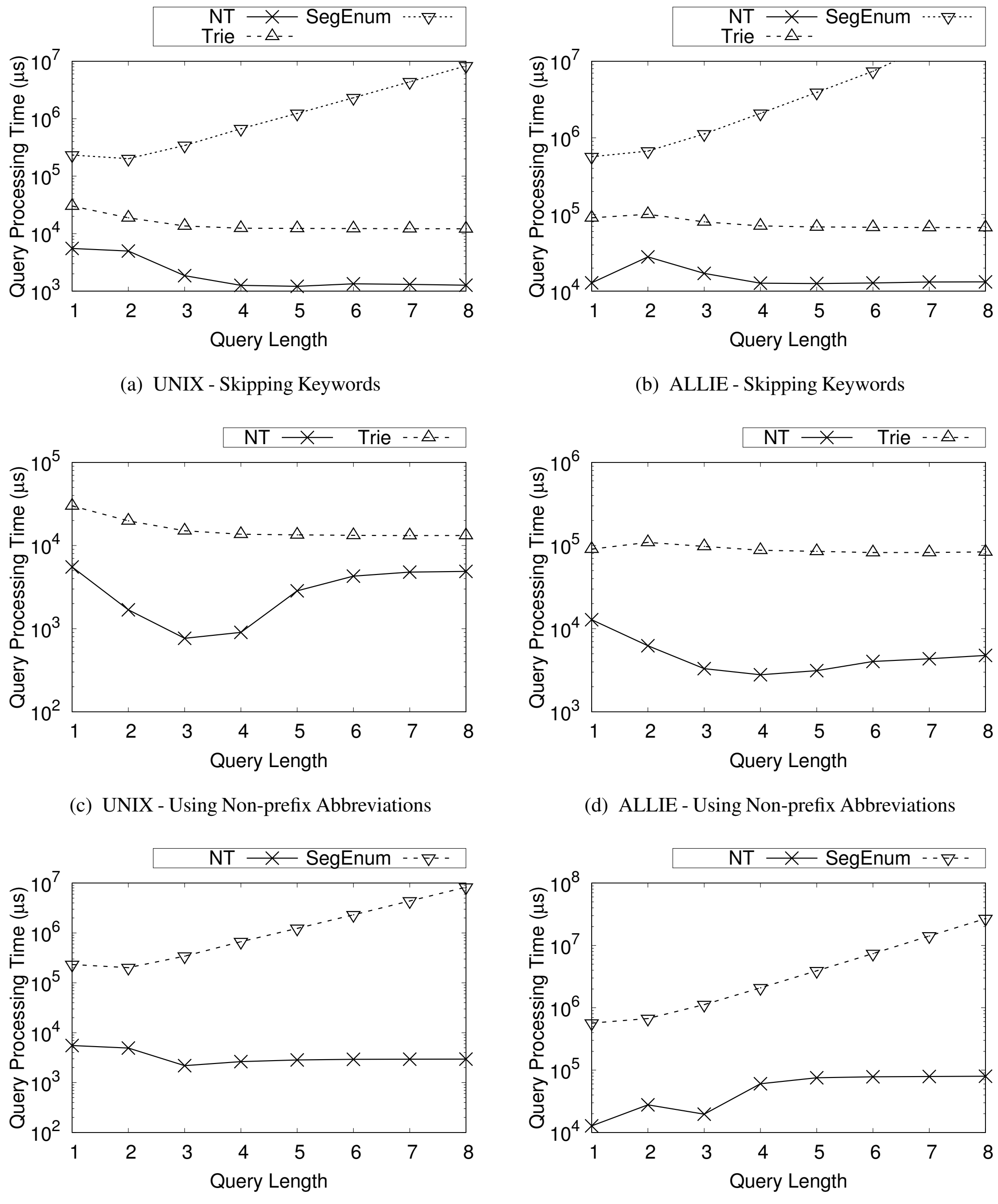
3.11.6 Extensions

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 3.7.1, 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 3.11(a) – 3.11(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 3.11: Query processing time for extensions.

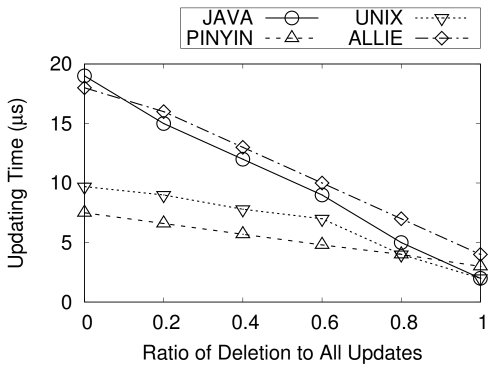


FIGURE 3.12: Update processing time.

3.11.7 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 3.6.4 for the single index setting (i.e., no auxiliary index). Figure 3.12 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.

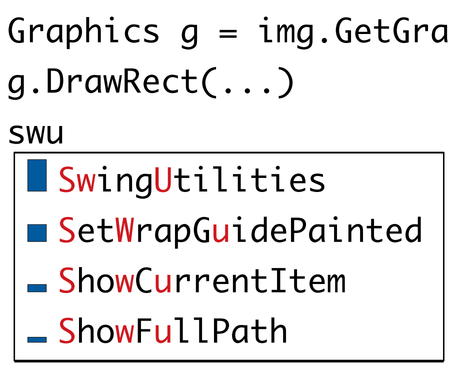
Chapter 4

Scope-aware Code Completion with Discriminative Modeling

4.1 Introduction

Code completion is a very useful feature for programmers, especially beginners, when they input long API names in integrated development environments (IDEs). It aims to help formulate accurate predictions for users' intended input APIs to save keystrokes and avoid typographical errors. This feature has become popular in prevalent IDEs for the following three reasons: First, according to the receiver object type, code completion can provide meaningful API method calls appearing in this object's definition, hence avoiding low-level incorrect API invocations. Second, if developers are not familiar with the APIs that should be called in their current context, code completion is able to present all possible completions in a pop-up window, providing an overall view and documentation to help beginners to learn programming patterns. Third, with code completion, developers are encouraged to use longer and more descriptive method names to improve code readability. We show an example of code completion in Example 4.1.

Example 4.1. *Suppose a user wants to input an API name "SwingUtilities". He types the first few characters such as "**swin**", and then the system automatically suggests "**Swin**gUtilities" and "**Swin**gWorker".*



```
Graphics g = img.GetGraphics(...)
g.DrawRect(...)
swu
```

```
SwingUtilities
SetWrapGuidePainted
ShowCurrentItem
ShowFullPath
```

FIGURE 4.1: Code completion for acronym-like input.

We call the above problem setting *code completion for prefix-like input*. It receives a prefix-like input and returns a candidate API if the method name begins with the given prefix. Such setting is adopted in [116]. However, there is a major drawback of such a problem setting which makes it unpractical in some cases: when the candidate API set becomes larger, completion becomes less effective, especially when some prefixes are found to be shared by many API names. E.g., more than a hundred methods in JButton, a class of Java, begin with the prefix “get”. To narrow down the candidate list, a user has to additionally type a longer prefix which significantly compromises the benefit of code completion.

To solve this problem, we need to find the intended API name by requiring a short input from the user to reduce the typing efforts. In this work, we adopt an acronym-like input paradigm instead of the prefix-like one. An illustrating example is shown below.

Example 4.2. *In Fig. 4.1, suppose the user types in characters such as “**swu**”. Then the system adopting acronym-like matching paradigm suggests “**Sw**ing**U**tilities”, “**S**et**W**rapG**u**idePainted”, “**S**ho**w**C**u**rrentItem” and “**S**ho**w**F**u**llPath”*[1].

We call such a problem setting *code completion for acronym-like input*. It receives an acronym-like input and returns a candidate API if the method name contains all the characters of the input in a subsequence matching way[2].

[1]Our method supports completions for both package names such as “SwingUtility” and API names such as “showFullPath”. Without losing generality, we capitalize all the first characters of API names for the ease of illustration.

[2]In our practical setting without losing generality, we assume that the first characters of an input and an API candidate must be exactly matched.

```
Graphics g = img.GetGraphics(...)
g.SetColor(...)
g.DrawRect(...)
g.FillRect(...)
g.DrawString(...)
SwingUtilities.InvokeAndWait(...)
painter.SetWrapGuidePainted(...)
```

FIGURE 4.2: Scope for graphics utility.

```
VideoPlayer p;
p.SetCamera(...)
p.SetVideoSource(...)
p.SetAudioSource(...)
p.ShowFullPath(...)
p.ShowCurrentItem(...)
```

FIGURE 4.3: Scope for video utility.

Existing solutions to code completion focus on using neural language models [73, 117–120] or statistical language models [6–9] learned from a large code base by modeling it into a natural language processing problem. However, they fail to utilize the user's input to narrow down the candidate list by proper relevance ranking. We argue that the acronym-like input from users will remarkably improve the completion accuracy when the underlying input patterns are taken into consideration. Intuitively, an acronym-like input from a user will be definitely affected by some acronym patterns due to human typing behavior. Take an example in Fig. 4.1. A user's input "swut" has a higher probability to match "**Sw**ing**Ut**ilities" than to match "**S**ho**wFu**llPa**t**h", because the acronym for the latter is hardly to be "swut" in common practice. In order to take advantage of the underlying patterns of human typing behavior, a transformation model needs to be learned for the purpose of providing high-quality candidates. In this work, we use a machine learning technique to learn the patterns to obtain a transformation model.

The scope context information [6–9], which is described as the co-occurrences of APIs in a scope, is also found to be a helpful feature to improve the prediction accuracy. The reason is that there are many fixed API pattern flows in a scope for a specific utility. E.g., Fig. 4.2 shows a typical API invoking pattern flow for graphics processing, while Fig. 4.3 shows another case for video processing. Detecting such scope utility type can obviously improve the prediction accuracy when a new API invoking statement is inserted into the current scope.

To integrate different features, we propose a discriminative model which can assign different weights for each feature to achieve more accurate performance. This discriminative model consists of three features: 1) the API usage counts collected from the training corpus to reflect the popularity of each API, 2) the transformation probability that a user's input is transformed into the intended API, and 3) the scope context information represented by co-occurrence counts of APIs, for which a transformation model is learned by logistic regression. The above features are combined linearly in the discriminative model for the overall prediction, and their weights are learned by a support vector machine (SVM).

In addition to the prediction accuracy, we address the efficiency challenges when computing the top-k completions using our discriminative model. Specifically, we develop a candidate ranker framework to firstly generate the most possible k candidates and then use the ranker to re-rank the top-k results. We use a trie index to efficiently generate the possible candidates and then use inverted lists located on the trie's leaf nodes to store the scope context information for fast co-occurrence lookups.

Experiments are conducted with a large-scale training dataset collected from GitHub and a test set which covers 12 popular Java projects. The results demonstrate the effectiveness of our approach: it outperforms the baseline methods by up to 7.3% on top-1 accuracy. The experiments on efficiency show that our approach is faster than the baseline methods by up to 31 times.

To the best of our knowledge, this is the first work that focuses on improving ranking performance on the problem of code completion by utilizing the user's input. We also note that our method is orthogonal to the existing code suggestion methods [6–9] because it can independently work as a standalone module after language model-based code suggestions.

Our main contributions are summarized as follows.

- We propose a novel method for code completion using acronym-like input.
- We propose a discriminative model that combines API counts, transformation probability, and scope context information for accurate code completion.
- We develop an efficient algorithm to compute top-k candidates from a trie index.

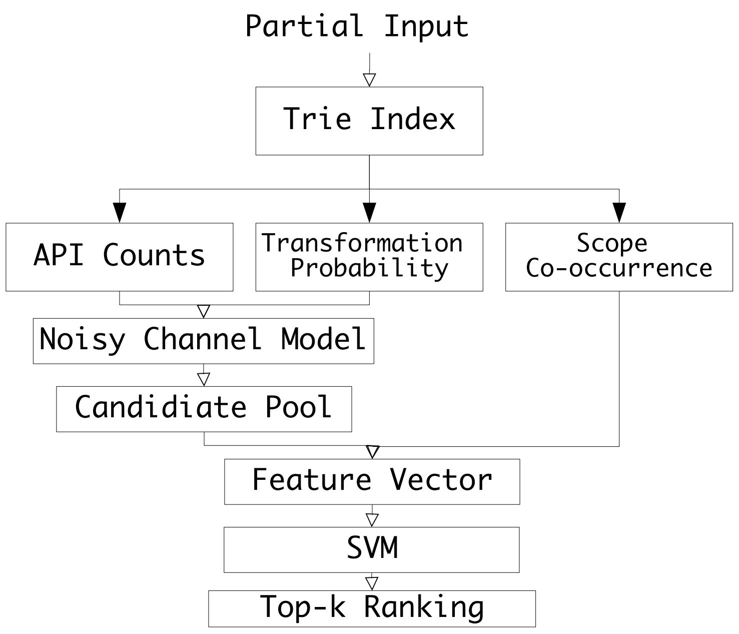


FIGURE 4.4: Overview of the process.

- Extensive experiments show the performance of both effectiveness and efficiency.

The rest of our chapter is organized as follows: Section 4.2 presents the details of our discriminative model. Section 4.3 shows the index structure and candidate ranker framework. Section 4.4 reports the experimental results. Section 4.5 surveys related work. Section 4.6 concludes the chapter.

4.2 A Ranker-based Model

In this section, we propose a discriminative model to rank the API candidates. The basic idea of our model is to combine three main features with a proper weighting for more accurate predictions. To account for efficiency challenges, we adopt a ranker-based model which consists of a candidate generator and an SVM-based ranker. First, the candidate generator uses a traditional noisy channel model to roughly pick up the most possible top-*k* API completions. Then, a carefully tuned SVM-based ranker will re-rank the top-*k* completions again and finally output a re-ranked completion list. An overview is showed in Fig. 4.4. Our process includes three steps:

1. API names are indexed using a trie, with corresponding scope context information collected from our large training corpus. Each scope is assigned a unique scope ID and each API

has a list of scope IDs such that this API appears in these scopes. Such lists of scope IDs are linked with the leaf nodes of our trie index for fast access.

2. An SVM is trained using three features, which are API usage counts collected from our code base, transformation probability and scope co-occurrence counts. The transformation probability describes how likely the input from the user is the completion candidate and is trained by a logistic regression model.
3. Search for API candidates from the trie by matching the user's input in a subsequence matching way. The previous s lines of context from current code position are taken as the scope context information. Then we use a traditional noisy channel model to roughly rank and output a top-k list as the candidate pool. Finally, we use the trained SVM to re-rank the top-k list to obtain an ultimate result list.

We first introduce a traditional noisy channel model and then give the definition of our discriminative model.

4.2.1 Noisy Channel Model

Noisy channel model is widely used in string transformation tasks, especially for spelling correction. Given an input $Q = q_1 \cdots q_{|Q|}$, we want to find the best transformed string $C = c_1 \cdots c_{|C|}$ among all candidates that match the input:

$$C* = \arg\max_{C} P(C|Q) \tag{4.1}$$

By applying Bayes' Rule and dropping the constant denominator, we have

$$C* = \arg\max_{C} P(Q|C)P(C) \tag{4.2}$$

where the transformation model $P(Q|C)$ models the transformation probability from C to Q, and the language model $P(C)$ models how likely C is the intended input. One problem with the noisy channel model is that there is no weighting for the two kinds of probabilities, and because they are often estimated from diverse sources, suboptimal performance might be incurred with regard to diversity of the sources [121, 122]. Moreover, noisy channel model

TABLE 4.1: Transformation model features *Sim*.

ID	Description
Sim_1	`number of consonant letter matches`
Sim_2	`number of vowel letter matches`
Sim_3	`number of capital letter matches`
Sim_4	`number of letter skips`
Sim_5	`number of skipping occurs`
Sim_6	`percentage of letter matches`

cannot utilize additional useful features (e.g., scope context information), and this becomes a severe limitation in practice.

As our subsequence matching paradigm can be seen as a generalized case for string transformation, we can use noisy channel model directly to model our problem.

The language model $P(C)$ can be trained by simply counting the frequency of the API names in the code base, in line with many previous works [9, 58, 123, 124].

The transformation probability $P(Q|C)$ is learned using a logistic regression model and the training details are the same with the work [5]. The logistic regression model is shown below:

$$\begin{aligned} P(Q|C) &= g(\beta_0 + \beta_1 \cdot Sim_1(Q,C) + \cdots + \beta_6 \cdot Sim_6(Q,C)) \\ where \quad g(z) &= \frac{1}{1+e^{-z}} \end{aligned} \tag{4.3}$$

where β_i represents the regression coefficients, $Sim_i(Q,C)$ is the similarity feature showed in Table 4.1 and $g(z)$ is the logistic function.

4.2.2 Discriminative Model

A discriminative model may overcome the shortcomings of noisy channel model by adding additional features and applying proper weightings. A general discriminative formulation of the problem is of the following form:

$$C* = \arg\max_{C}[\mathbf{w} \cdot \mathbf{F}(Q,C)] \tag{4.4}$$

where $\mathbf{F}(Q,C)$ is a vector of features and $\mathbf{w}$ is the model parameter which is a vector of weights. Compared with the noisy channel model, this discriminative formulation is more general. We can deem the noisy channel model as a special case of the discriminative form where only two features, the language model estimates and the transformation probability are used and uniform weightings are applied. In this work, the weightings $\mathbf{w}$ are derived by training an SVM showed in Section 4.2.4.

4.2.3 Scope-awareness

Scope context information has been proved to be very helpful in code suggestions, as it can describe which API methods are often invoked before the intended API method is called. We add the scope context variable in our discriminative model as the scope co-occurrence counts. It describes how often the candidate API appears with its previous API names in the same scope by collecting the statistics of the large training corpus. After adding in this feature, our discriminative model can be extended as:

$$\begin{aligned} C* = \arg\max_{C}[&w_0 + w_1 \cdot F_{lang}(Q,C) \\ &+ w_2 \cdot F_{trans}(Q,C) \\ &+ w_3 \cdot F_{scope}(Q,C)] \end{aligned} \tag{4.5}$$

where Q is the input, C is the candidate, $F_{lang}(Q,C)$ represents the unigram language model probability, which is calculated by a normalized usage counts, $F_{trans}(Q,C)$ represents the transformation probability from C to Q and $F_{scope}(Q,C)$ represents the scope co-occurrence counts of C and Q.

4.2.4 An SVM-based Ranker

As only one candidate API is relevant to code completion (such setting was also adopted by many previous studies [8, 117, 118]), which is different from the traditional document retrieval problem, we only need to consider the possibility of an API as "hit" or "not hit". This can be essentially handled by a classification model such as a support vector machine. The one with higher possibility to be classified as "hit" will be ranked higher, and vice versa. We do not employ RankSVM [125] because usually detailed human-judged ranking data such as clickthrough data or log is not available. Our training SVM examples are generated from the noisy channel model, whose detailed rankings do not really matter. Hence they cannot be used as pair-wise training data in RankSVM.

The settings of our SVM are similar with [126]. The feature vectors are passed to a support vector machine employing a simple radial basis function (RBF) kernel with $\gamma = 1$ after all feature values are normalized between [0,1]. We employed Joachim SVM^{light} [126, 127] implementation.

Our SVM model is trained on a human-crafted training set. Such a training set comprises $\langle$`API-candidate, feature-vector, class`$\rangle$ triples. The `class` has two values: +1 and −1. +1 will be assigned if `API-candidate` is the intended API, otherwise −1 will be assigned. To describe how likely a candidate API is a "hit", the trained SVM will output a value between −1 and +1 for each candidate it generates. Then we can use these values to rank these candidates. This works because the value output by the SVM represents the distance to the maximum-margin hyperplane [126].

Table 4.2 gives an example to illustrate the features of candidates for an input "swu" from the user. The scope context information is extracted from Fig. 4.2 and Fig. 4.3. Note that each feature is a normalized value between [0,1]. Suppose a user types an input "swu", then all the matched candidates are listed in Table 4.2 according to the subsequence matching paradigm. For example, "SwingUtilities" has an F_{lang} which is its usage counts in the code base, an F_{trans} which is $P(swu|SwingUtilities)$ value computed from the transformation model and an F_{scope} which is the scope co-occurrence value with its previous context "DrawRect" and "GetGraphics" showed in Fig. 4.1. These feature vectors are then passed into our trained SVM to give a final ranking list.

TABLE 4.2: Example candidates for input "swu".

Candidates	F_{lang}	F_{trans}	F_{scope}
`SwingUtilities`	0.6	0.7	0.5
`SetWrapGuidePainted`	0.2	0.2	0.1
`ShowCurrentItem`	0.2	0.1	0.1
`ShowFullPath`	0.1	0.1	0.1

4.3 Searching Algorithm

TABLE 4.3: Example dataset *S*.

ID	String	Popularity
API_1	`DrawRect`	0.6
API_2	`GetGraphics`	0.8
API_3	`SetAudioSource`	0.2
API_4	`SetCamera`	0.1
API_5	`SetColor`	0.6
API_6	`SetVideoSource`	0.1
API_7	`SetWrapGuidePainted`	0.2
API_8	`ShowCurrentItem`	0.2
API_9	`ShowFullPath`	0.1
API_{10}	`SwingUtilities`	0.6

In this section, we show the detailed algorithm and index implementation of the three steps showed in Section 4.2. There is a straightforward way to generate candidates: traverse the trie for all possible matching strings and then calculate all the probabilities for them and sort them in descending order. However, the number of all matching candidates might be prohibitive. Thus we adopt a ranker-based method which consists of a candidate generator and an SVM-based ranker. The candidate generator is responsible for only picking up the most possible top-*k* candidates and then SVM-based ranker will carefully rank these results for higher accuracy. Moreover, in the generator step, early termination techniques and a threshold-based algorithm are applied to efficiently compute top-*k* candidates.

4.3.1 Candidate Generator of a Trie

The basic index structure of the candidate generator is a trie built on top of all the API names. Trie is a tree structure where each path from the root to a leaf node corresponds to a API name. Given an input, we can find its corresponding node, and traverse all its leaf nodes to obtain the corresponding API candidates. An example API set with popularities (normalized usage counts) is given in Table 4.3 and the corresponding trie is showed in Fig. 4.5. The API "SwingUtilities" has a trie node 106. On each leaf node, we store the API usage counts in it and link it with an inverted list of the ordered scope IDs. We call such an inverted list a *scope inverted list*. With the ordered scope inverted list, we can look for co-occurrences by simply doing a list intersection operation efficiently.

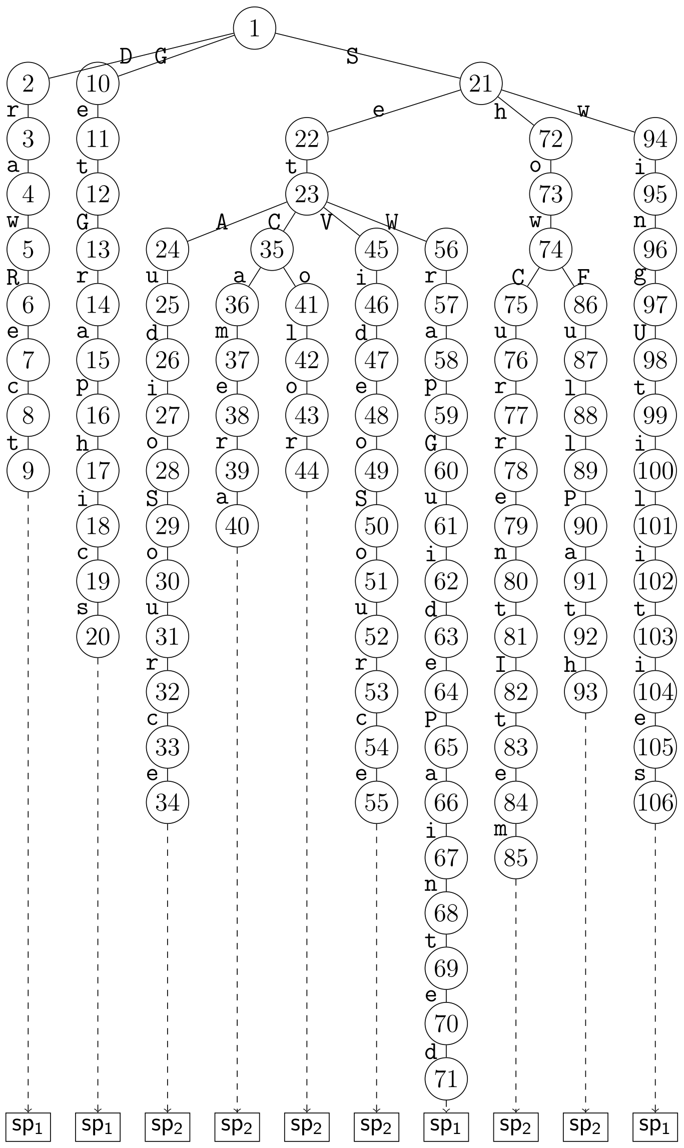


FIGURE 4.5: A trie with scope inverted lists.

Our search strategy is based on two assumptions: First, the first characters of an input and an API candidate must be exactly matched. E.g., it is very unlikely that a user will input "sw" to look for "Ba**sh****W**rite". Second, the user will not skip too many keywords in an acronym-like input. Such an assumption is made after the observation of the collected acronyms from Amazon Mechanical Turk. Thus, we only match the nodes whose distances to the current one are within next two keywords in the trie. E.g., "snt" can match **S**howCurre**nt**Item but cannot match **S**etWrapGuidePai**nt**ed, because in the trie, the next two keywords for `S` are `CurrentItem` and `WrapGuide`. Thus the keyword `Painted` cannot be matched.

The algorithm is illustrated in Algorithm 12. It takes as input the user's input, the scope

Algorithm 12: Generator-Trie (q, s, T)

```
Input   : $q$ is the user input, $s$ is the scope context, $T$ is a trie built on $S$.
Output  : $\{API_i\}$, such that $API_i \in S$ and $q$ is a subsequence of $API_i$.
1  $A \leftarrow \{$ the root of $T\}$ ;                                  /* node set */
2  foreach character $q[i]$ do
3      $A' \leftarrow \emptyset$;
4      foreach $n \in A$ do
5          if $q[i]$ is the first character in $q$ AND $n$ has a child n' through $q[i]$ then
6              $A' \leftarrow A' \cup \{n'\}$ ;                         /* first character */
7              continue;
8          if $n$ has any descendant $n'$ through character $q[i]$ within a distance threshold $\tau$ then
9              $A' \leftarrow A' \cup \{n'\}$;
10     $A \leftarrow A'$;
11 $R \leftarrow \emptyset$;
12 foreach $n \in A$ do
13     $R \leftarrow R \cup$ API candidates stored in the subtree rooted at $n$;
14 foreach API $\in R$ do
15     API.co-occurrence $\leftarrow$ API.scope-list Intersects s.scope-list;
16 return SVM-Ranking$(R)$;
```

context and the trie. The first characters of user input and candidate API are exactly matched (Line 5). Then it begins to traverse the trie to match each character of the input for each node's descendant nodes iteratively (Lines 1 – 10). This ensures it searches for all the API names in a subsequence matching way. Line 8 makes sure that the distance between the current node and next matching node will not be too long. We set the distance threshold τ as the distance to the end of next two keywords w.r.t the specific API names. After fetching all the matched API candidates, it intersects each of the candidates with the context API(s) to calculate co-occurrences, respectively (Lines 11 – 15). Especially when the scope context s consists of multiple lines of APIs, we take the union of the scope-lists for all APIs in context s without removing repeated scope IDs as s.scope-list. Finally, all the generated candidates will be sent to SVM for further ranking (Line 16).

Take an example in Fig. 4.1, for an input "swu" and its scope context "DrawRect". One node that matches "swu" is node 98. One node that matches "DrawRect" is node 9. Node 9 is already a leaf node and node 98 has an only descendant leaf node 106. Both nodes have scope inverted lists of $\langle \mathsf{sp}_1 \rangle$ and $\langle \mathsf{sp}_1 \rangle$, respectively. After intersecting these two lists, we obtain the co-occurrence scope and the co-occurence count for "swu" and "DrawRect" is $\langle \mathsf{sp}_1 \rangle$ and

1, respectively.

4.3.2 Efficient Ranking Algorithm

We observe that with the increase of the scope context lines, the co-occurrence computation will need prohibitive intersection operations for the scope inverted lists of all the possible candidates. This cost can be unbearable for a real-time code completion system. To solve this issue, we first select the most possible candidates by using a noisy channel model showed in Section 4.2. Recall that this model needs to compute a product of $P(C) \cdot P(Q|C)$. $P(C)$ is stored at each leaf node in the trie. Hence we can materialize the maximum $P(C)$ at each intermediate node for a threshold algorithm (TA). $P(Q|C)$ is computed on-the-fly using a logistic function showed in Section 4.2. Note that as the logistic function is a monotonically increasing function, we can also obtain a maximum upper value of $P(Q|C)$ by only computing the longest prefix matched with the input. Moreover, as the input length of users mainly lies in the range of $[1,6]$ (see Table 4.5), we do not consider off-line probability pre-computation methods, which practically do not improve the overall runtime by much but will result in large memory consumption.

In detail, when a user issues an input q, we send the q and previous scope context line(s) s to the search algorithm. We first search for s to obtain the scope inverted lists of the API results. Then we search q according to a subsequence matching manner in the trie. For each matched path in the trie, we compute the transformation probability by the logistic function in Equation 4.3. Then we can use $P(C) \cdot P(Q|C)$ to compute a upper bound score UB. By using the UB, we can compute the rough top-k candidates efficiently by using a priority queue for early termination. After that, intersection operations are only done between scope context and top-k candidates. Feature vectors will be sent to our SVM for a final ranking.

The detailed algorithm is showed in Algorithm 13. To apply the pruning techniques, we simply use it to replace Lines 11 – 16 in Algorithm 12. A priority queue is initialized for early termination (Line 1). Then for each node, it iterates through the corresponding API candidates of each node (Line 5), compute the $P(API) \cdot P(q|API)$ as an overall score by $score(API, q)$ (Line 6). If the number of candidates in the priority queue is less than k or the candidate's score is greater than the k-th temporary API, we insert this API candidate into the queue (Lines 6

Algorithm 13: TopK-Pruning (q, A, k)

```
$R \leftarrow \emptyset$;                                  /* a priority queue of size $k$ */
foreach $n \in A$ do
    if $n$.UB $\leq R[k].score$ then
        continue;
    foreach API as n's descendant leaf do
        if $|R| < k$ or $score(API, q) > R[k].score$ then
            $R.insert(API)$;
foreach API $\in R$ do
    API.co-occurrence $\leftarrow$ API.scope-list Intersects s.scope-list;
return SVM-Ranking($R$);
```

– 7). If the node's upper bound UB is less than the score of k-th temporary API, we skip this node for pruning (Lines 3 – 4). Eventually, we only need to do the intersection operations between k API candidates and the scope context, then send the k candidates to our SVM for further ranking (Lines 8 – 10).

The time complexity is $O(k|s| + \mathscr{N} logk)$, reducing from the naïve one's $O(\mathscr{O}|s| + \mathscr{O} log \mathscr{O})$, where k is the candidate pool size, $|s|$ is the context size, i.e., the number of context lines, $\mathscr{N}$ is the number of candidates scanned until the process terminated and $\mathscr{O}$ is the number of unique API names.

We show an example for the top-2 ranking process according to Table 4.2. Given the scope context "DrawRect" and an input "swu", we first search for "swu" in the trie and find node 61, 76, 87, 98. The maximum usage counts for them are 0.6, 0.2, 0.2, 0.1. The transformation probability is computed as $P(swu|SwingU) = 0.7$, $P(swu|SetWrapGu) = 0.2$, $P(swu|ShowCu) = 0.1$ and $P(swu|ShowFu) = 0.1$. Finally the noisy channel model probability is computed as $\mathsf{UB}_{\mathsf{SwingU}} = 0.6 \times 0.7 = 0.42$, $\mathsf{UB}_{\mathsf{SetWrapGu}} = 0.2 \times 0.2 = 0.04$, $\mathsf{UB}_{\mathsf{ShowCu}} = 0.2 \times 0.1 = 0.02$, $\mathsf{UB}_{\mathsf{ShowFu}} = 0.1 \times 0.1 = 0.01$. We only keep the top-2 API names, "SwingUtilities" and "SetWrapGuidePainted" remain. The intersections between scope inverted lists are done for both API names to compute co-occurrences. After that, both API names are sent to our SVM for the final ranking.

TABLE 4.4: Dataset statistics.

Dataset	LOCs	Files	Total Projects
Java Corpus	10,753,168	86,158	1,000
Java Test	2,788,955	11,371	12

4.4 Experiments

We conducted extensive experiments to evaluate both effectiveness and efficiency of our proposed method against baseline methods. In this section, we report the experimental results and analyses.

4.4.1 Experiment Setup

Two datasets are collected for model training and testing tasks.

- **Java Corpus**[3] is a large-scale code base collected from all Java projects on GitHub. We sampled the large corpus into the 1,000 projects **Java Corpus** to more clearly show the impact of training set size, in line with the previous works [8, 128]. We use this dataset as a training set in our evaluations. We use the API usage counts and scope context extracted from this code base.
- **Java Test** is a dataset used in [110] collected from 12 popular Java projects. We use this dataset as a test set for evaluations.

Table 4.4 shows the statistics of the two datasets, where LOCs is the line of codes, Files is the number of files and Total Projects is the number of projects included.

We only use APIs in Java Development Kit (JDK) as the dictionary to build the trie index. In total 17,116 APIs appearing in JDK 8 are collected to build the trie.

We use the code base of **Java Corpus** as our corpus and use Eclipse's Java parser to parse the code for scope extractions. We tried different scope granularity such as class scope, method scope and block scope. Finally we choose to use class scope for the best balance between efficiency and accuracy. API usage counts are also extracted from **Java Corpus**.

[3]http://groups.inf.ed.ac.uk/cup/javaGithub/

TABLE 4.5: Input length distribution.

length of input	1	2	3	4	5	6
average length of candidate API names	5.3	7.4	9.0	12.0	14.0	16.1
# of API names	17	62	259	354	139	52

To train the logistic regression model, we extract 5,000 API names, 4,000 from **Java Corpus** and 1,000 from **Java Test**. Then we use these API names to collect 5,000 acronym-like $\langle input, API \rangle$ pairs. The acronym-like input is collected from volunteers in Amazon Mechanical Turk, by telling them to intuitively give an input when they see an original API name. Our transformation model is trained on the complete training set using logistic regression model showed in Section 4.2 and well tuned by enough iterations.

Although the acronym-like input collected from Amazon Mechanical Turk has lengths ranging from 1 to 8, most of the input does not need to be fully typed to obtain its corresponding top-1 completions. E.g., the acronym-like input for "SwingUtility" we collected is "swut", but "SwingUtility" will be ranked as top-1 in our approach when "swu" is typed.

To quantitatively demonstrate that the completion is useful, we show in Table 4.5 the statistics about the input length distribution for how many characters are needed when the corresponding API completion is ranked as top-1 in our approach. In this table, **length of input** means the necessary characters for its completion to be ranked as top-1, **average length of candidate API names** means the average length of such API names. **# of API names** means how many API names, out of the 1,000 collected ones, will be ranked as top-1 for the given length of input. E.g., the first column means that a total of 17 API names will be ranked as top-1 when the first letter is input, and their average length is 5.3. It can be seen that the length of input is significantly shorter than the average length of API names.

To train the SVM model, for each API name in the training set, we find an appropriate scope in **Java Corpus** for the API to fit in and extract its previous API names located in the scope context. Similarly, for each API in the test set, we also find an appropriate scope in **Java Test** and extract its previous API names as context for prediction purpose. Then we generate the candidates' feature vectors as $\langle candidate, input, context \rangle$, where $candidate$ represents the candidate API name, $input$ is the user input and $context$ is the scope context API(s).

The following algorithms are compared.

- APIREC is a statistical model based approach in [128]. We carefully implement the method and use our own test set **Java Test** for evaluations.
- POP is the popularity-based sorting method used in [123].
- NCM is the noisy channel model method described in Section 4.2.
- SDM is our proposed scope-aware ranker-based method with discriminative modeling.

Note that APIREC does not utilize any input API name but provide suggestions when the user presses the "." button. In NCM and SDM, the candidate pool size is set to 50. That is, in SDM, the top-50 results are first selected and then passed to our SVM for further re-ranking. We have tried different values for the candidate pool size and 50 is proved to have a considerable processing time and does not lose any accuracy.

In addition, we extract the nearest *one* line prior to the current calling method as the scope context information in default for better performance unless explicitly stated otherwise. The reason is explained in Section 4.4.2.

The experiments were carried out on a PC with an Intel i5 2.6GHz Processor and 8GB RAM, running Ubuntu 14.04.3. The algorithms were implemented in C++ and in a main memory fashion.

4.4.2 Evaluation of Effectiveness

We adopt the same evaluation metrics used in existing studies [8, 117, 118]. We evaluate: (1) top-k accuracy, which indicates the fraction of times the correct API appears in the top-k candidates, where $k \in 1, 3, 5, 10$. (2) Mean Reciprocal Ranking (MRR), which is calculated as the average reciprocal of the correct API's rank in the top-k candidates. MRR can give an overall evaluation of the model. The closer to 1 the MRR value, the better the ranking accuracy.

We first evaluate the top-k accuracy with the baseline methods. Table 4.6 shows the results. The last column shows the comparison of MRR. As seen, SDM achieves higher accuracy than any other baseline method. Without utilizing the user's input, APIREC can only reply on predictions by statistical models, thus causing low accuracy compared with the other methods.

TABLE 4.6: Accuracy comparison.

Model	top-1	top-3	top-5	top-10	MRR
APIREC	29.6	43.4	58.2	70.2	0.407
POP	81.0	95.7	97.7	98.7	0.884
NCM	81.8	96.2	97.7	98.7	0.891
SDM	88.3	97.3	98.6	99.0	0.928

At top-1 accuracy, SDM has the largest improvements of 7.3%, 6.5%, 58.7%, over POP, NCM and APIREC, respectively. Along with the increase of k, the advantage becomes not that obvious because the correct API will be more easily included in a larger top-k list. Nonetheless, SDM still outperforms POP, NCM and APIREC on all the values of k. NCM is better than naïve POP approach, which suggests that the input from user is useful to predict the intended API names. However, the observed improvements over POP are very limited due to the lack of weights applied in NCM. We also observe that, SDM achieves the highest MRR of 0.928, meaning that on average in 10 cases, it can almost correctly rank the API on top of its list among 9 cases. The relative improvements on MRR is 3.3% and 1.9% over POP and NCM, respectively. These experimental results verified the significant improvements on accuracy of our proposed scope-aware discriminative model. We also observe that SDM has higher accuracy than APIREC on the column top-10, indicating that some rare API names which would hardly appear in statistical models can be retrieved in SDM by more strictly input matching.

Figure. 4.6 shows an overall comparison for top-k accuracy by varying k from 1 to 30. The top-20 and top-30 accuracy for APIREC is 74.2% and 76.4%, respectively. Hence we do not show APIREC anymore because the conclusions are almost the same as Table 4.6. We can observe that SDM outperforms other two baseline methods at any $k \in [1,30]$. The gap becomes close in [20,30] because for a large k, even the baseline method can include the correct candidate easily. Another valuable observation is that when $k = 30$, SDM is still higher than POP and NCM. This suggests that there are some API names especially with low frequency in the code base can never be retrieved by POP or NCM. As our SDM can consider the scope context and apply a proper weighting on it, these rare API names can be easily retrieved in our model.

Varying Context Line Size. Scope context size may have large impacts on the accuracy of our model. Here, we explain the scope context size as the *number of lines* prior to the current calling method within the same scope. The co-occurrence counts between each context line

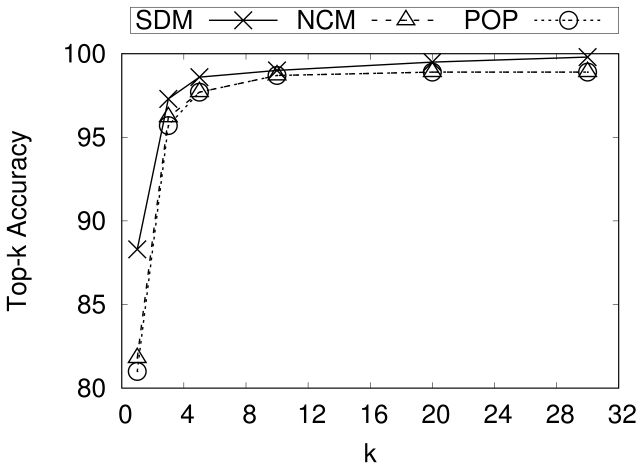


FIGURE 4.6: Top-*k* accuracy of different approaches.

TABLE 4.7: Accuracy with different context size.

Context Size	top-1	top-3	top-5	top-10	MRR
1	88.3	97.3	98.6	99.0	0.928
2	85.5	97.3	98.6	99.0	0.913
3	85.5	97.1	98.6	99.0	0.912
4	85.5	97.1	98.4	99.0	0.912
5	85.5	97.1	98.4	99.0	0.912

and current API candidate are summed as the feature *co-occurrence counts*. Table 4.7 shows the results by varying the scope context size from 1 to 5. Interestingly, we can observe that with the increase of the context size, the accuracy slightly drops, the same as the MRR. We examined the test examples and found that summing up co-occurrence counts of multiple previous lines might over-weigh the co-occurrence feature in our model thus leading to inaccurate predictions. This fact reminds us that excessive context information might be not beneficial to accuracy but lead to deteriorative performances. There might be a way to take use of the multiple lines of context information more properly but such techniques are beyond the scope of this work.

Varying Training Set Size. We also want to analyze the impacts on accuracy by varying the size of our training set. For our large-scale dataset **Java Corpus**, we randomly sample the 1,000 projects into two subsets, one with 300 projects and one with 100 projects, denoted by Train300 and Train100. Then we evaluate the accuracy by varying these three training sets. We show the results in Table 4.8. We can obviously see that Train1000 always outperforms Train100 and Train300. The largest improvements occur at top-1, are 4.8% and 2.6% over Train100 and Train300, respectively. Train300 is always better than Train100. This verified

TABLE 4.8: Accuracy of different training set size.

Dataset	top-1	top-3	top-5	top-10	MRR
Train100	83.5	96.1	97.6	98.7	0.898
Train300	85.7	96.4	98.1	98.7	0.911
Train1000	88.3	97.3	98.6	99.0	0.928

our intuition that a larger training set will contribute to better prediction accuracy.

4.4.3 Evaluation of Efficiency

A practical code completion system must be efficient enough to work in real-time to avoid interrupting a developer's flow of coding. Thus, for efficiency, we evaluate the overall processing times in Fig. 4.7. If not otherwise noted, the scope context size and training set will be set to 1 and Train1000 in default. In Fig. 4.7, we vary the input length and plot the average runtime of the code completion system as SDM. We also plot the results of the straightforward method mentioned in Section 4.3, denoted by SDM-NoNCM, meaning without the noisy channel model but sending all the matched API candidates to our SVM for ranking. For SDM, a larger k will cause better accuracy but longer processing time. By trying different k values, we choose $k = 50$ since it has the same accuracy with SDM-NoNCM but also runs efficiently. We can directly observe that SDM is much faster than SDM-NoNCM, because SDM uses a noisy channel model to drop hopeless candidates with extremely low probabilities to avoid prohibitive additional computations. The maximum speedup is 31 times, at length of 1. SDM-NoNCM is very slow due to the numerous matched API candidates which are needed to be passed to SVM given a short input while our SDM only keeps the most possible k API candidates for re-ranking. SDM is around 1 ms for all the lengths, and thus can be applied to Web settings, such as online IDEs. The times begin to decrease when we use a longer input because longer input is more selective thus less candidates are processed for both SDM and SDM-NoNCM.

Varying Context Line Size. The processing times by varying scope context line size are compared. Figure. 4.8 shows the results by varying the context size from 1 to 5. As seen, a large context size might incur considerable overhead that causes sensible system delays. Context-5 is almost 3–4 times slower than Context-1. This suggests that if multiple lines of context information are used, further optimization on processing might be required.

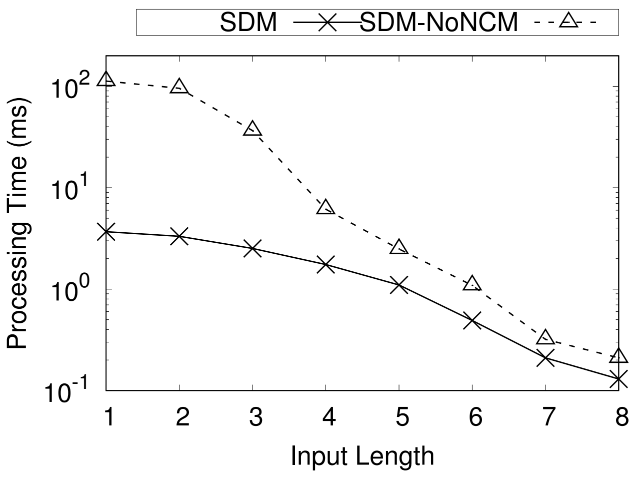


FIGURE 4.7: Processing time comparison.

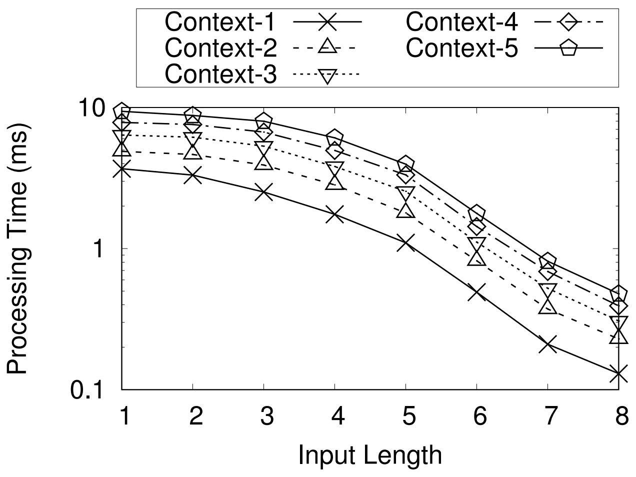


FIGURE 4.8: Processing time with different context size.

Varying Training Set Size. The processing times by varying training set size are evaluated. For our large-scale dataset **Java Corpus**, we show the results in Fig. 4.9. The evaluated training set is the same with that in Table 4.8. Train1000 is the slowest, 1.8 and 1.5 times slower than Train100 and Train300, at the length of 1 while its corpus size is 10 and 3.3 times larger than Train100 and Train300. Intuitively, the larger the training corpus, the slower the processing time. This is mainly because larger training corpus will contain more scope context information such that the scope inverted list will become longer and slower for lookup operations. Nonetheless, the growth rate on processing time is much lower than the corpus size, and thus a larger corpus might be always preferable for better accuracy performance.

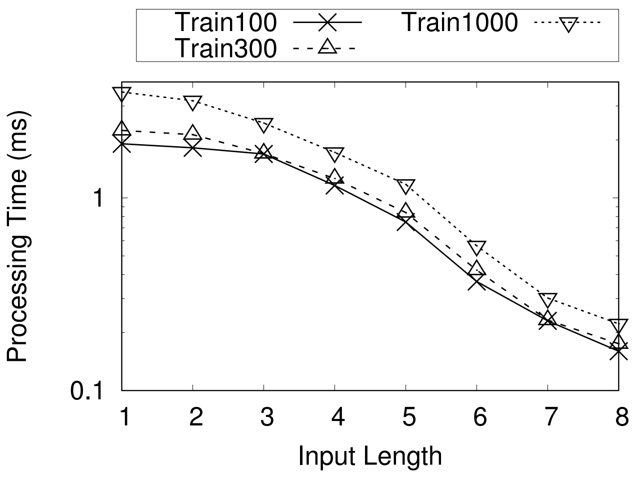


FIGURE 4.9: Processing time with different training set size.

4.5 Related Work

Code Completion. With the birth of text editors, the research on code completion has received much attention in the past several decades. In one early study, Willis *et al.* [92] proposed an approach to expand some abbreviations into a sentence to save input efforts. In the domain of programming IDEs, Little and Miller [110] proposed to translate a small number of unordered keywords provided by the user into a valid expression in order to reduce the need to remember syntax and API names in Java. After this work, Han *et al.* [5] used a hidden Markov Model learning from a code corpus to expand ordered abbreviated keywords into a valid code expression. Their work was followed by Pini *et al.* [94] to additionally deal with the keyword missing problem. They used Support Vector Machine (SVM) to create classifiers to judge whether a keyword is an abbreviation or not. A recent study by Sandnes [129] developed a system to predict word input by only using a simple longest common subsequence algorithm for practical use.

Code Suggestion. The code suggestion problem is to try using statistical language models [9] (LMs) to predict the next code line without any input from users. We refer readers to two latest studies [130, 131] about source code naturalness. Nguyen *et al.* [7] proposed to use a semantic model to capture the patterns of source code, by incorporating a local semantic n-gram model with a global n-gram topic model. Graph-based LMs are proposed in [6] and [8] to capture graph-based patterns from source code. After that, Savchenko and Vokkov [132] also propose a probabilistic model with n-gram models to calculate a sorted list of all possible functions. Scope and context information has been proved to greatly improve the accuracy for predictions in these studies [6–9]. Recent trends feature a boom by applying the Deep Learning Network (DNN) instead of LMs. However, While many studies [73, 117, 119, 120] have been developed to accommodate DNN in their code suggestion systems, a study [118] from Hellendoorn and Devanbu showed that carefully adapting n-gram models for source code can yield better performance than deep-learning models.

String Transformation. The string transformation problem is to map a source string s into another desirable form t. This problem has been extensively studied in the natural language processing community. A specific case for this problem is spelling correction. Okazaki *et al.* [133] proposed to use substring substitution rules as features in their discriminative models to generate transformed string candidates. Duan *et al.* [121] proposed a discriminative model based on latent structural SVM to model the alignment of words in the spelling correction process.

4.6 Conclusion

In this work, we have studied the problem of code completion using a scope-aware ranker-based discriminative ranking model. We use an acronym-like input setting to avoid the fatal drawback of existing code completion systems. To improve the accuracy, we utilize API usage counts, transformation probability and scope context information as the features to pass to our trained SVM as a discriminative model. To solve the efficiency challenge, we adopt a ranker-based model to use noisy channel model as a filter to eliminate hopeless candidates. We have examined our approach with a training corpus and a test set. The experimental results have shown that our proposed method outperforms the existing methods in terms of both effectiveness and efficiency.

Chapter 5

Real World Data Circulation

In this chapter, I illustrate the realized data circulation in my thesis in Section 5.1 and present my contributions to the society in Section 5.2.

5.1 Realized Data Circulation

First, I show an overview of the data circulation realized in my works in Fig. 5.1.

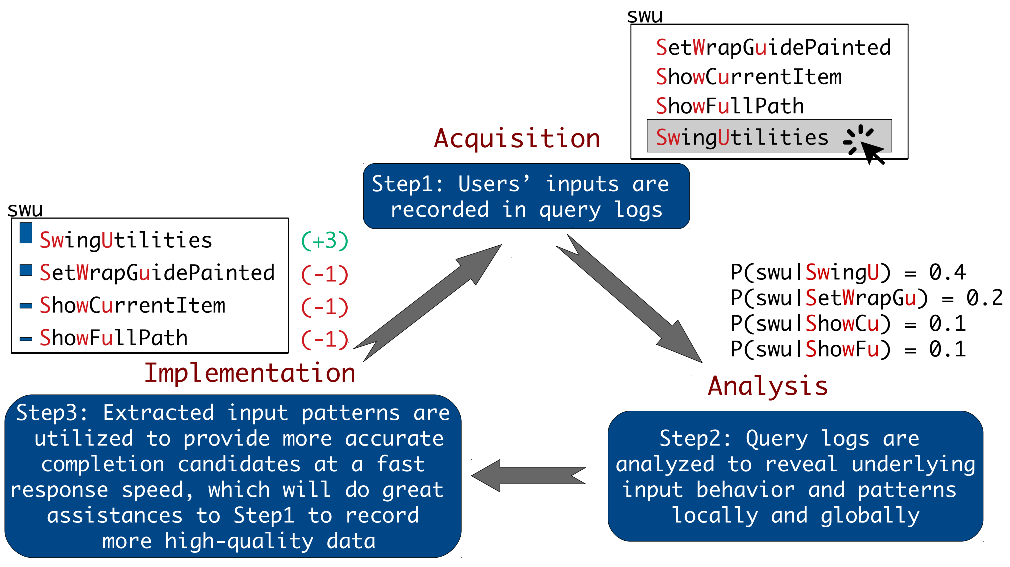

swu
SetWrapGuidePainted
ShowCurrentItem
ShowFullPath
SwingUtilities
Acquisition
Step1: Users' inputs are recorded in query logs
swu
SwingUtilities (+3)
SetWrapGuidePainted (-1)
ShowCurrentItem (-1)
ShowFullPath (-1)
P(swu|SwingU) = 0.4
P(swu|SetWrapGu) = 0.2
P(swu|ShowCu) = 0.1
P(swu|ShowFu) = 0.1
Implementation
Analysis
Step3: Extracted input patterns are utilized to provide more accurate completion candidates at a fast response speed, which will do great assistances to Step1 to record more high-quality data
Step2: Query logs are analyzed to reveal underlying input behavior and patterns locally and globally

FIGURE 5.1: Real world data circulation in autocompletion.

As Fig. 5.1 shows, I divide the real world data circulation into three steps:

- Step1: Acquisition
- Step2: Analysis
- Step3: Implementation

In the general sense, Acquisition means revealing the users' real intentions or desires from the digital footprints in existing services left by the users. Analysis indicates employing the cutting-edge data analysis techniques such as statistical models, machine learning and deep learning models to describe the underlying patterns from the observations of Acquisition. Analysis also includes utilizing novel database techniques such as indexing and query processing paradigm designs to handle scalable data volumes collected in Acquisition. A well-formulated data model will fit the users' desires properly and thus be able to provide accurate predications. A scalable and robust data model can handle simultaneous query requests to satisfy real-time requirements. In Implementation step, I apply the models and prototypes obtained in Analysis to specific applications. Parameters are adjusted to fit the problem settings in real-world scenarios. By inspirations and fancy ideas, innovative products and services are expected to be developed in this step.

In this thesis, I realized the data circulation in terms of autocompletion. Acquisition here represents the observations of entire users' activities that occur from the user's first keystroke in the search box to the last selection of an autocompletion candidate provided. The specific input of a user can reflect his input behavior and patterns. The cursor hovering time over a completion can be seemed as an implicit feedback. The final clickthroughed link can be considered highly related to the user intentions. All of such information is recorded into query logs for further analysis.

For Acquisition, I have used large datasets in all the three works presented in this thesis. For example:

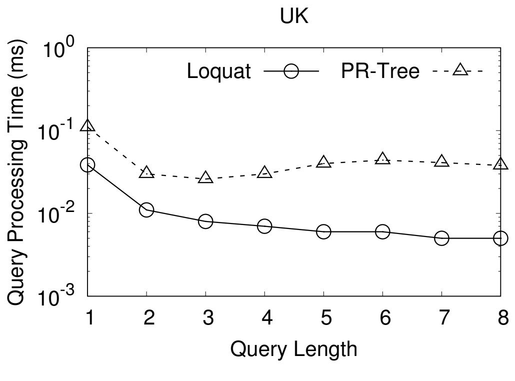
- FSQ is a dataset containing 1M POIs collected from real-world check-in data.
- PINYIN is the Chinese input method corpus compiled by collecting frequent words appearing in real-world communications. It contains 3.55M Chinese phrases.
- Java Corpus is a large-scale code base collected from the real-world code bases on GitHub. It contains 10M lines of codes.

Moreover, corresponding hand-crafted training sets are collected from Amazon Mechanical Turks. I manage to handle these “Big Data” to produce efficient and effective completions in my thesis. These “Big Data” are sufficient reflections and perfect representations of real-world data.

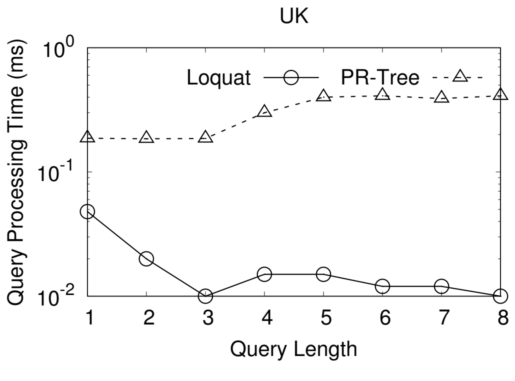
In Analysis step, I focus on revealing the input behavior patterns of users by employing machine learning and statistical modeling techniques for accurate predications. In Chapter 3, I take use of Gaussian Mixture Model (GMM) to evaluate the probability (density function) of a specific input pattern. In Chapter 4, I utilize the noisy channel model, logistic regression and Support Vector Machine (SVM) to accurately predict the user’s real intended completions. I consider the input patterns and usage counts as global features as well as the scope context information as a local feature. Thus I can explore the underlying input patterns both locally and globally. A significant observation is that common input patterns of human writing behavior can evidently improve the completion accuracy. E.g., users tend to preserve consonant letters more than vowel letters when input abbreviations. Moreover, to handle scalable query throughputs, I proposed novel indexing techniques in Chapter 2 and Chapter 3.

In Implementation step, I demonstrate superior performances using my prototype autocompletion systems. The work developed in Chapter 2 can be directly applied to many Location-based Service (LBS) applications such as Google Map, POI recommendation systems or trip route recommendation systems. My prototype system in Chapter 2 can handle a hundred thousand queries per second on a very large dataset containing 13M records. My works in Chapter 3 and Chapter 4 can be directly applied to online integrated development environments (IDEs), cloud input method editors (IMEs), desktop search systems, and search engines. Experiments in Chapter 3 show that my prototype system can save on average 21.6% keystrokes at most and achieve 121 times speedup at most than the runner-up method. Experiments in Chapter 4 show that my experimental system can achieve 6.5% improvement at top-1 accuracy and achieve 31 times speedup than the naïve method. Sufficient experiments results prove that my prototype systems are superior in terms of both effectiveness and efficiency, which can be easily adapted to innovative products and services.

To show the real-world applicability of my algorithms, I incorporate the algorithms in Chapter 2 into a real-world mapping application. This application is called Loquat (Location-aware query autocompletion). I use a proper dataset UK here. UK is a dataset containing POIs (e.g. banks and cinemas) in UK (www.pocketgpsworld.com). It contains 181,549 POIs in total and the geographical coordinates are within the territory of UK. I compare my system with the state-of-the-art method Prefix-Region Tree [4] (denoted as PR-Tree).

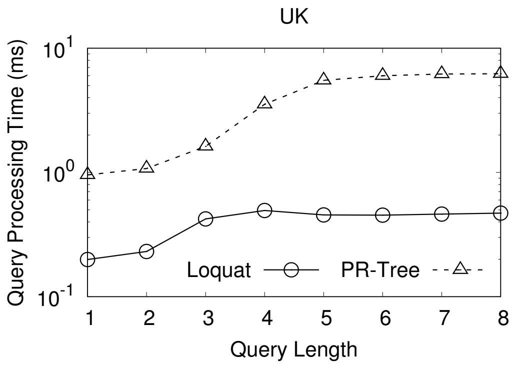


(a) Range Query - UK

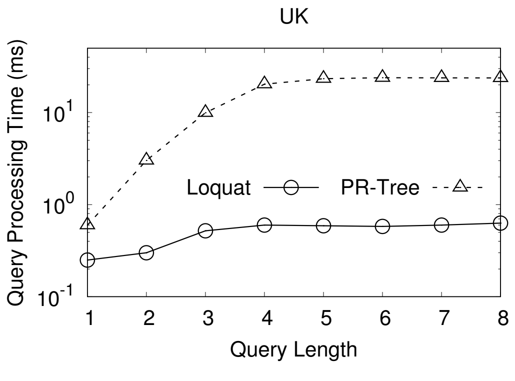


(b) Top-*k* Query - UK

FIGURE 5.2: Performance of range and Top-*k* query .



(a) Range Query - UK



(b) Top-*k* Query - UK

FIGURE 5.3: Performance with error-tolerant feature.

Figure 5.2(a) and Fig. 5.2(b) show the efficiency on range queries and top-*k* queries, respectively. Loquat is at most 10–15 times faster than PR-Tree. Figure 5.3(a) and Fig. 5.3(b) show the efficiency on range queries and top-*k* queries with error-tolerant feature, respectively. Same advantages are also observed that Loquat is at most 10–15 times faster than PR-Tree.

High-performance products and services in Implementation will also help to provide more high-quality data for the collection in Acquisition, which forms an occlusive circulation in terms of autocompletion.

Figure 5.1 gives an example to illustrate the data circulation of the work in Chapter 4. First begins with the Acquisition step. Suppose a user types an input “swu”, then all the matched candidates are listed in a naïve alphabetical order as `SetWarpGuidePainted`, `ShowCurrentItem`, `ShowFullPath`, `SwingUtilities`. After that, the user clicked `SwingUtilities` as his intended completion. Then Analysis step begins. Based on the observations in Acquisition, I train my machine learning model to let it have a high probability to match “swu” with “SwingU” but low probabilities with the others. Note that $P(Q|C)$ in Fig. 5.1 represents the probability of abbreviating C as Q if the user’s intended completion is C. Last is the Implementation step. After training my model, `SwingUtilities` will have a higher rank (move up by 3 positions) in the autocompletion candidates. Thus I can obtain more accurate predications by learning the user input patterns.

5.2 Contributions to the Society

Autocompletion can do great assistances in human knowledge explorations. The data circulation formed in my works can help better understand the search intentions of users, which can benefit the society from many aspects.

First, in traditional search systems under e-commerce settings, effective query autocompletion can directly lead to a purchase decision. Figure 5.4 shows an example of the autocompletion system of Amazon. A customer-personalized autocompletion system can stimulate consumer spending and save the time of users, which might boost the economic growth of a country as well as improve customer satisfaction.

Second, an effective and efficient autocompletion system for Location-based Service can promote the local tourism development. Figure 5.5 shows an example to autocomplete the query as “Restaurants near you”. Comparing with existing systems such as Google Maps which can only tolerate a few typos in its autocompletion, my work [12] can tolerate more typos in a dynamic way. This is extremely important when the location names are too long, e.g., “Gasselterboerveenschemond”, the name of a village located in Netherlands. A personalized autocompletion system can balance the user preferences and spatial distances to provide proper rankings. A scalable and robust autocompletion system can support large query throughputs

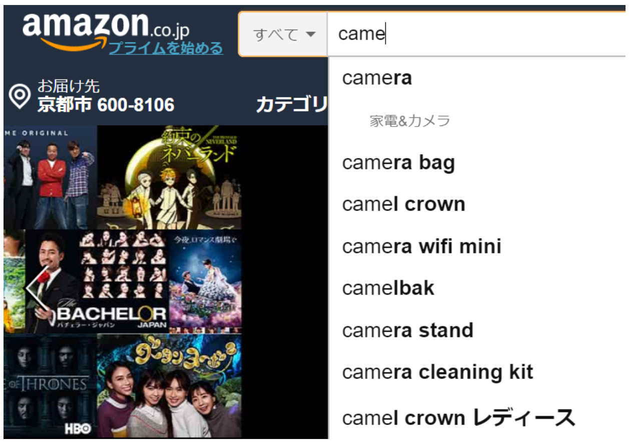
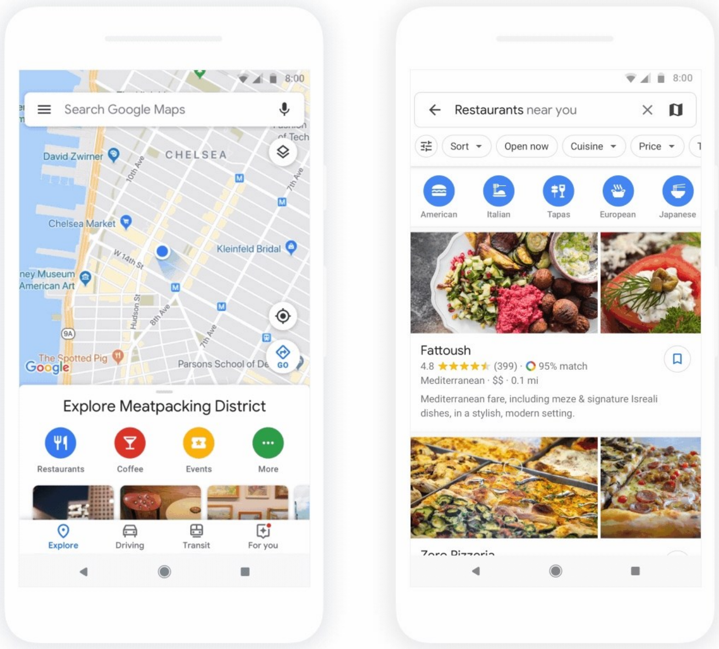


FIGURE 5.4: Autocompletion at Amazon.

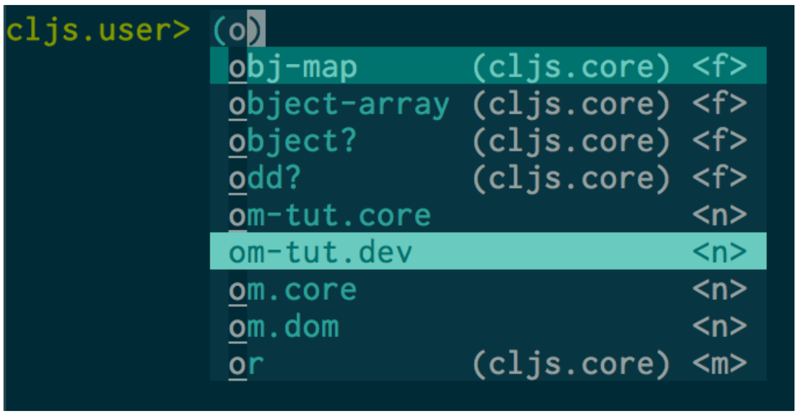


Mediterranean fare, including meze & signature Isreali dishes, in a stylish, modern setting.

FIGURE 5.5: Autocompletion on Goolge Maps.

and thus provide smooth and stable services. Therefore, it will help to increase the venues of restaurants, museums, hotels and tourist attractions.

Third, a quick-response online IDE editor with accurate API suggestions can help the programmers or designers to improve their productivities in their work. We show an example in Fig. 5.6. With the growing popularity of online text editors/IDEs (e.g., Overleaf and IBM Bluemix), the demand on efficiency and effectiveness is increasing. Comparing with these existing online IDEs, my works [13, 14] can support acronym-like input and thus save almost 30% keystrokes. Especially in team work, an effort-saving and well-performed code



```
cljs.user> (o)
           obj-map      (cljs.core) <f>
           object-array (cljs.core) <f>
           object?      (cljs.core) <f>
           odd?         (cljs.core) <f>
           om-tut.core              <n>
           om-tut.dev               <n>
           om.core                  <n>
           om.dom                   <n>
           or           (cljs.core) <m>
```

FIGURE 5.6: Autocompletion in online IDEs.

completion system will boost the coordination among multiple workers, which will definitely create more value for the whole society.

Chapter 6

Conclusions and Future Work

In this chapter, we conclude this thesis in Section 6.1 and present several interesting future directions in Section 6.2.

6.1 Conclusions

In this thesis, we studied autocompletion for online services in a wide variety of fields in recent decades. There are many practical challenges for online real-time autocompletion in the real world. In particular, efficiency challenges are the most important ones which will influence the response time of an autocompletion system directly. We focused our attention on the indexing techniques in various autocompletion paradigms to handle these efficiency challenges.

We studied three different autocompletion systems for online services. The first one is *location-aware QAC*, which searches for POIs whose locations are close to the query location and whose textual contents begin with the given string prefix. In the second work, we investigated *autocompletion for prefix-abbreviated input*. The keywords will be completed based on the assumption that users may abbreviate keywords by prefixes and do not have to explicitly separate them. Finally, our third work studied *code completion with acronym-like input*. The API candidates will be completed by incorporating the users' acronym-like input conventions and the APIs' scope context into a discriminative model.

For online autocompletion systems, a high query throughput is demanded because there is a large number of simultaneous queries submitted to the online service. Hence, we developed efficient algorithms for these three problems and conducted comprehensive performance studies using real datasets. We summarized each of our proposed solutions as follows.

6.1.1 Location-aware Autocompletion

In this work, we studied efficient algorithms and indexing techniques for location-aware query autocompletion. We search for matched POIs if they are close to the querying spatial location and their textual descriptions begin with the given query prefix. Under this setting, we managed to answer range and top-k queries at a large scale.

To avoid unnecessary computations, we proposed a method by which data objects are indexed in a trie with integrated spatial information. We utilize trie node to store the textual information and equip each trie node with an additional bit array to indicate the spatial partition by a quadtree. The proposed indexing techniques are very effective in pruning the hopeless results at an early stage.

Furthermore, we proposed several pruning algorithms at different levels for fast lookups. We calculate the upper bound score in subtree level and region level to help skip unnecessary node access. We also discussed how to extend our method to support the error-tolerant feature.

The experimental results show that the speedup of our method can be up to one to two orders of magnitude at range queries and 4 times faster at top-k queries. This demonstrates the efficiency of the proposed method and its superiority over existing methods. We also managed to index a dataset with 13 million points of interests in 32GB main memory on a commodity machine, which shows that our method can support scalable datasets.

Moreover, in Chapter 5, we demonstrate that our algorithms can work very well considering the dynamic behavior of users. In Chapter 5, we adopt a relatively small dataset containing 1000K POIs in UK. The overall runtime results can prove that our algorithms can preserve superior performances when users move their locations to totally strange cities.

6.1.2 Autocompletion for Prefix-Abbreviated Input

In this work, we proposed a new feature of query autocompeltion which takes prefix-abbreviated keywords as input. Such a new feature supports more application scenarios particularly for those in which users may not explicitly specify delimiters of keywords. We also consider the user's mental cost to make such abbreviations by themselves. In the applications of input method editor and medical term search, the mental cost is trivial because it is very easy to recall the whole words by short abbreviations. In the applications of IDEs and file search, we argue that users can first explore the completion results by inputting very short prefixes and then use the intermediate results to recall the keywords. Considering the naïve iterative search on a trie will incur prohibitive computational cost, we proposed a novel index called nested trie to support efficient query processing.

We first analyzed the inefficiencies of the naïve index, along with a few other possibilities. After that, we developed an efficient indexing and query processing method to deal with the new autocompletion feature. Based on our analysis, we proposed inner trie and outer trie structures to comprise a nested trie for fast search. We also developed efficient algorithms to merge the lists intervals located on the trie nodes. To further improve the performance, we developed list merge optimization techniques to accelerate the list merging computation.

We also devised a ranking method specific to the proposed autocompletion paradigm and an efficient top-*k* result fetching algorithm. As obviously some prefixes are preferred by users, we decided to learn the prefix-abbreviated patterns from the dataset we collected from Amazon Mechanical Turk. We described such patterns by utilizing a mixture of a finite number of Gaussian distributions with unknown parameters. After that, we developed our pruning algorithm based on the maximum score of intervals located on each nested trie node for fast query processing.

In the experiments, we compared our approach with several baseline methods in terms of MRR, Success Rate and query processing time. Experiments on real datasets showed the effectiveness of the new type of query autcompeltion and the superiority of the query processing method over alternative solutions in terms of efficiency.

6.1.3 Code Compeltion

In this work, we investigated the problem of code completion using a scope-aware discriminative ranking model. We adopted an acronym-like input setting to avoid the fatal drawback of existing code completion system. In other words, we take into consideration the users' input acronym-like input conventions and the APIs' scope context to provide more accurate completions.

To improve the accuracy, we utilized the API usage counts, transformation probability and scope context information as the features to pass to our trained SVM as a discriminative model. To solve the efficiency challenge, we adopt a ranker-based model to use noisy channel model as a filter to eliminate hopeless candidates.

In our experiments, we evaluated our approach with a training corpus and a test set. We compared our method with several baseline methods in terms of top-k accuracy and query processing time. Our method has an improvement of 6.5% over the runner-up at top-1 accuracy. For efficiency, our approach achieved a maximum speedup about 31 times than the baseline method. The experimental results have shown that our proposed method outperformed the existing methods in terms of both effectiveness and efficiency.

6.2 Future Work

In this thesis, we considered the autocompletion for online services and our proposed approaches are mainly aimed at improving efficiency. As one future direction, we can consider more on the improvements on perspectives of effectiveness. Furthermore, we can consider more well-structured objects such as entity completions other than plain texts.

Other interesting research directions include autocompletion on other types of data structures such as graphs and XML documents. Moreover, we can consider autocompletion on road network navigations when an input from a geographic information system is necessary.

In the following sections, we present specific future directions for each of our three works.

6.2.1 Location-aware Autcompletion

In the current work, the POIs are independent points with static popularity scores on the map. For future work, we plan to take use of the boosted LBSN (Location-based Social Network) to enhance the location-location relationship for semantic autocompeltion. We can take use of the rich properties from LBSN such as functional regions, time, user attributes and even descriptive landscape pictures. we also plan to utilize external resources such as knowledge graphs or corpuses to enrich the utilities of our methods. Moreover, we also consider multimedia techniques to combine landscape descriptions in our methods.

6.2.2 Autocompletion for Prefix-Abbreviated Input

Currently, we use a generative probabilistic model of GMM to rank our completions. In the future, we consider improving this ranking module with more complex ranking model by utilizing learning to rank techniques. We also plan to incorporate user profile information to personalize the abbreviated patterns for more accurate suggestions. Moreover, we consider adding the error-tolerant feature to make such a novel QAC paradigm more practical. Currently, we are going to deploy our algorithms as a practical service and submit it as a demonstration paper in the future.

6.2.3 Code Completion

In the future, we plan to improve the ranking model from the perspective of information retrieval. In this work, we adapt our ranking model to all the code bases including different projects. Thus, in the future, we can consider collecting the project topic information and programmer information for more customized code completion. Furthermore, we will study the naturalness and conventions underlying in different programming languages to provide more accurate completions on top of specific programming languages. We are going to develop our techniques as plug-ins in online IDEs in the future. Moreover, our acronym-like input paradigm can be extended as various applications to improve its usability in the real world. We can adapt the method to complete a sentence, a road network sequence, or equip it with a general speech recognition interface to improve productivities.

Bibliography

[1] Senjuti Basu Roy and Kaushik Chakrabarti. Location-aware type ahead search on spatial databases: semantics and efficiency. In *SIGMOD*, pages 361–372, 2011.

[2] Shengyue Ji and Chen Li. Location-based instant search. In *SSDBM*, pages 17–36, 2011.

[3] Yuxin Zheng, Zhifeng Bao, Lidan Shou, and Anthony K. H. Tung. INSPIRE: A framework for incremental spatial prefix query relaxation. *IEEE Trans. Knowl. Data Eng.*, 27(7):1949–1963, 2015.

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

[5] Sangmok Han, David R. Wallace, and Robert C. Miller. Code completion of multiple keywords from abbreviated input. *Autom. Softw. Eng.*, 18(3-4):363–398, 2011.

[6] Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 69–79, 2012.

[7] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. A statistical semantic language model for source code. In *SIGSOFT*, pages 532–542, 2013.

[8] Anh Tuan Nguyen and Tien N. Nguyen. Graph-based statistical language model for code. In *ICSE 2015, Florence, Italy, May 16-24, 2015*, pages 858–868, 2015.

[9] Muhammad Asaduzzaman, Chanchal Kumar Roy, Kevin A. Schneider, and Daqing Hou. CSCC: simple, efficient, context sensitive code completion. In *ICSME 2014, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 71–80, 2014.

[10] Fei Cai and Maarten de Rijke. A survey of query auto completion in information retrieval. *Foundations and Trends in Information Retrieval*, 10(4):273–363, 2016.

[11] Unni Krishnan, Alistair Moffat, and Justin Zobel. A taxonomy of query auto completion modes. In *ADCS*, pages 6:1–6:8, 2017.

[12] Sheng Hu, Chuan Xiao, and Yoshiharu Ishikawa. An efficient algorithm for location-aware query autocompletion. *IEICE Transactions on Information and Systems*, 101-D(1):181–192, 2018.

[13] Sheng Hu, Chuan Xiao, Jianbin Qin, Yoshiharu Ishikawa, and Qiang Ma. Autocompletion for prefix-abbreviated input. *ACM SIGMOD 2019, July 1-5, Amsterdam, Netherlands, to appear*, 2019.

[14] Sheng Hu, Chuan Xiao, and Yoshiharu Ishikawa. Scope-aware code completion with discriminative modeling. *IPSJ Journal of Information Processing, to appear*, 2019.

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

[16] Milad Shokouhi and Kira Radinsky. Time-sensitive query auto-completion. In *SIGIR*, pages 601–610, 2012.

[17] Milad Shokouhi. Learning to personalize query auto-completion. In *SIGIR*, pages 103–112, 2013.

[18] Cyrus Omar, YoungSeok Yoon, Thomas D. LaToza, and Brad A. Myers. Active code completion. In *ICSE*, pages 859–869, 2012.

[19] Surajit Chaudhuri and Raghav Kaushik. Extending autocompletion to tolerate errors. In *SIGMOD*, pages 707–718, 2009.

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

[21] Guoliang Li, Jiannan Wang, Chen Li, and Jianhua Feng. Supporting efficient top-k queries in type-ahead search. In *SIGIR*, pages 355–364, 2012.

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

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

[24] Korinna Grabski and Tobias Scheffer. Sentence completion. In *SIGIR*, pages 433–439, 2004.

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

[26] Ju Fan, Hao Wu, Guoliang Li, and Lizhu Zhou. Suggesting topic-based query terms as you type. In *APWeb*, pages 61–67, 2010.

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

[28] Alpa Jain and Gilad Mishne. Organizing query completions for web search. In *CIKM*, pages 1169–1178, 2010.

[29] Jyun-Yu Jiang, Yen-Yu Ke, Pao-Yu Chien, and Pu-Jen Cheng. Learning user reformulation behavior for query auto-completion. In *SIGIR*, pages 445–454, 2014.

[30] Yanen Li, Anlei Dong, Hongning Wang, Hongbo Deng, Yi Chang, and ChengXiang Zhai. A two-dimensional click model for query auto-completion. In *SIGIR*, pages 455–464, 2014.

[31] Bhaskar Mitra, Milad Shokouhi, Filip Radlinski, and Katja Hofmann. On user interactions with query auto-completion. In *SIGIR*, pages 1055–1058, 2014.

[32] Katja Hofmann, Bhaskar Mitra, Filip Radlinski, and Milad Shokouhi. An eye-tracking study of user interactions with query auto completion. In *CIKM*, pages 549–558, 2014.

[33] Liangda Li, Hongbo Deng, Anlei Dong, Yi Chang, Hongyuan Zha, and Ricardo A. Baeza-Yates. Analyzing user's sequential behavior in query auto-completion via markov processes. In *SIGIR*, pages 123–132, 2015.

[34] Aston Zhang, Amit Goyal, Weize Kong, Hongbo Deng, Anlei Dong, Yi Chang, Carl A. Gunter, and Jiawei Han. adaqac: Adaptive query auto-completion via implicit negative feedback. In *SIGIR*, pages 143–152, 2015.

[35] Jyun-Yu Jiang and Pu-Jen Cheng. Classifying user search intents for query auto-completion. In *ICTIR*, pages 49–58, 2016.

[36] Liangda Li, Hongbo Deng, Anlei Dong, Yi Chang, Ricardo A. Baeza-Yates, and Hongyuan Zha. Exploring query auto-completion and click logs for contextual-aware web search and query suggestion. In *WWW*, pages 539–548, 2017.

[37] Liangda Li, Hongbo Deng, Jianhui Chen, and Yi Chang. Learning parametric models for context-aware query auto-completion via hawkes processes. In *WSDM*, pages 131–139, 2017.

[38] Stewart Whiting and Joemon M. Jose. Recent and robust query auto-completion. In *WWW*, pages 971–982, 2014.

[39] Fei Cai, Shangsong Liang, and Maarten de Rijke. Prefix-adaptive and time-sensitive personalized query auto completion. *IEEE Trans. Knowl. Data Eng.*, 28(9):2452–2466, 2016.

[40] Fei Cai and Honghui Chen. Term-level semantic similarity helps time-aware term popularity based query completion. *Journal of Intelligent and Fuzzy Systems*, 32(6):3999–4008, 2017.

[41] Yingfei Wang, Hua Ouyang, Hongbo Deng, and Yi Chang. Learning online trends for interactive query auto-completion. *IEEE Trans. Knowl. Data Eng.*, 29(11):2442–2454, 2017.

[42] Fei Cai and Maarten de Rijke. Selectively personalizing query auto-completion. In *SIGIR*, pages 993–996, 2016.

[43] Fei Cai, Wanyu Chen, and Xinliang Ou. Learning search popularity for personalized query completion in information retrieval. *Journal of Intelligent and Fuzzy Systems*, 33(4):2427–2435, 2017.

[44] Fei Cai, Ridho Reinanda, and Maarten de Rijke. Diversifying query auto-completion. *ACM Trans. Inf. Syst.*, 34(4):25:1–25:33, 2016.

[45] Wanyu Chen, Fei Cai, Honghui Chen, and Maarten de Rijke. Personalized query suggestion diversification. In *SIGIR*, pages 817–820, 2017.

[46] Bhaskar Mitra and Nick Craswell. Query auto-completion for rare prefixes. In *CIKM*, pages 1755–1758, 2015.

[47] Fei Cai and Maarten de Rijke. Learning from homologous queries and semantically related terms for query auto completion. *Inf. Process. Manage.*, 52(4):628–643, 2016.

[48] Aston Zhang, Amit Goyal, Ricardo A. Baeza-Yates, Yi Chang, Jiawei Han, Carl A. Gunter, and Hongbo Deng. Towards mobile query auto-completion: An efficient mobile application-aware approach. In *WWW*, pages 579–590, 2016.

[49] Bo-June Paul Hsu and Giuseppe Ottaviano. Space-efficient data structures for top-*k* completion. In *WWW*, pages 583–594, 2013.

[50] Roberto Grossi and Giuseppe Ottaviano. Fast compressed tries through path decompositions. *ACM Journal of Experimental Algorithmics*, 19(1), 2014.

[51] Giovanni Di Santo, Richard McCreadie, Craig Macdonald, and Iadh Ounis. Comparing approaches for query autocompletion. In *SIGIR*, pages 775–778, 2015.

[52] Umut Ozertem, Olivier Chapelle, Pinar Donmez, and Emre Velipasaoglu. Learning to suggest: a machine learning framework for ranking query suggestions. In *SIGIR*, pages 25–34, 2012.

[53] Rodrygo L. T. Santos, Craig Macdonald, and Iadh Ounis. Learning to rank query suggestions for adhoc and diversity search. *Inf. Retr.*, 16(4):429–451, 2013.

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

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

[56] Xiaoling Zhou, Jianbin Qin, Chuan Xiao, Wei Wang, Xuemin Lin, and Yoshiharu Ishikawa. BEVA: an efficient query processing algorithm for error-tolerant autocompletion. *ACM Trans. Database Syst.*, 41(1):5:1–5:44, 2016.

[57] Dong Deng, Guoliang Li, He Wen, H. V. Jagadish, and Jianhua Feng. META: an efficient matching-based method for error-tolerant autocompletion. *PVLDB*, 9(10):828–839, 2016.

[58] Huizhong Duan and Bo-June Paul Hsu. Online spelling correction for query completion. In *WWW*, pages 117–126, 2011.

[59] Inci Cetindil, Jamshid Esmaelnezhad, Taewoo Kim, and Chen Li. Efficient instant-fuzzy search with proximity ranking. In *ICDE*, pages 328–339, 2014.

[60] Ian De Felipe, Vagelis Hristidis, and Naphtali Rishe. Keyword search on spatial databases. In *ICDE 2008*, pages 656–665, 2008.

[61] Ariel Cary, Ouri Wolfson, and Naphtali Rishe. Efficient and scalable method for processing top-k spatial boolean queries. In *Int'l. Conf. on Scientific and Statistical Database Management (SSDBM 2010)*, pages 87–95, 2010.

[62] Zhisheng Li, Ken C. K. Lee, Baihua Zheng, Wang-Chien Lee, Dik Lun Lee, and Xufa Wang. Ir-tree: An efficient index for geographic document search. *IEEE TKDE*, 23(4):585–599, 2011.

[63] Dingming Wu, Gao Cong, and Christian S. Jensen. A framework for efficient spatial web object retrieval. *VLDB J.*, 21(6):797–822, 2012.

[64] Dingming Wu, Man Lung Yiu, Gao Cong, and Christian S. Jensen. Joint top-k spatial keyword query processing. *IEEE TKDE*, 24(10):1889–1903, 2012.

[65] Subodh Vaid, Christopher B. Jones, Hideo Joho, and Mark Sanderson. Spatio-textual indexing for geographical search on the web. In *Int'l. Symp. on Spatial and Temporal Databases (SSTD 2005)*, pages 218–235, 2005.

[66] Ali Khodaei, Cyrus Shahabi, and Chen Li. Hybrid indexing and seamless ranking of spatial and textual features of web documents. In *DEXA 2010*, pages 450–466, 2010.

[67] Maria Christoforaki, Jinru He, Constantinos Dimopoulos, Alexander Markowetz, and Torsten Suel. Text vs. space: efficient geo-search query processing. In *ACM CIKM 2011*, pages 423–432, 2011.

[68] Lisi Chen, Gao Cong, Christian S. Jensen, and Dingming Wu. Spatial keyword query processing: An experimental evaluation. *PVLDB*, 6(3):217–228, 2013.

[69] Hiroshi Motoda and Kenichi Yoshida. Machine learning techniques to make computers easier to use. *Artif. Intell.*, 103(1-2):295–321, 1998.

[70] Haym Hirsh and Brian D. Davison. An adaptive UNIX command-line assistant. In *AGENTS*, pages 542–543, 1997.

[71] Xiaohua Zhou, Xiaohua Hu, Xiaodan Zhang, and Xiajiong Shen. A segment-based hidden markov model for real-setting pinyin-to-chinese conversion. In *CIKM*, pages 1027–1030, 2007.

[72] Yabin Zheng, Chen Li, and Maosong Sun. CHIME: an efficient error-tolerant chinese pinyin input method. In *IJCAI*, pages 2551–2556, 2011.

[73] Veselin Raychev, Martin T. Vechev, and Eran Yahav. Code completion with statistical language models. In *PLDI*, pages 419–428, 2014.

[74] Valeriy Savchenko and Alexander Volkov. Statistical approach to increase source code completion accuracy. In *PSI*, pages 352–363. Springer, 2017.

[75] Merlijn Sevenster, Rob C. van Ommering, and Yuechen Qian. Algorithmic and user study of an autocompletion algorithm on a large medical vocabulary. *Journal of Biomedical Informatics*, 45(1):107–119, 2012.

[76] Ji-Rong Wen, HongJiang Zhang, and Jian-Yun Nie. Query clustering using content words and user feedback. In *SIGIR*, pages 442–443, 2001.

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

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

[79] Huanhuan Cao, Daxin Jiang, Jian Pei, Qi He, Zhen Liao, Enhong Chen, and Hang Li. Context-aware query suggestion by mining click-through and session data. In *KDD*, pages 875–883, 2008.

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

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

[82] Huanhuan Cao, Daxin Jiang, Jian Pei, Enhong Chen, and Hang Li. Towards context-aware search by learning a very large variable length hidden markov model from search logs. In *WWW*, pages 191–200, 2009.

[83] Alessandro Sordoni, Yoshua Bengio, Hossein Vahabi, Christina Lioma, Jakob Grue Simonsen, and Jian-Yun Nie. A hierarchical recurrent encoder-decoder for generative context-aware query suggestion. In *CIKM*, pages 553–562, 2015.

[84] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*. Cambridge University Press, 2008.

[85] Paolo Ferragina and Rossano Venturini. The compressed permuterm index. *ACM Trans. Algorithms*, 7(1):10:1–10:21, 2010.

[86] H. V. Jagadish, Nick Koudas, and Divesh Srivastava. On effective multi-dimensional indexing for strings. In *SIGMOD*, pages 403–414, 2000.

[87] Paolo Ferragina, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Two-dimensional substring indexing. In *PODS*, 2001.

[88] Ricardo A. Baeza-Yates. Searching subsequences. *Theor. Comput. Sci.*, 78(2):363–376, 1991.

[89] Ricardo A. Baeza-Yates and Gaston H. Gonnet. Fast text searching for regular expressions or automaton searching on tries. *J. ACM*, 43(6):915–936, 1996.

[90] Gonzalo Navarro. Nr-grep: a fast and flexible pattern-matching tool. *Softw., Pract. Exper.*, 31(13):1265–1312, 2001.

[91] Xiaochun Yang, Tao Qiu, Bin Wang, Baihua Zheng, Yaoshu Wang, and Chen Li. Negative factor: Improving regular-expression matching in strings. *ACM Trans. Database Syst.*, 40(4):25:1–25:46, 2016.

[92] Tim Willis, Helen Pain, Shari Trewin, and Stephen Clark. Informing flexible abbreviation expansion for users with motor disabilities. In *ICCHP*, pages 251–258, 2002.

[93] Tim Willis, Helen Pain, and Shari Trewin. A probabilistic flexible abbreviation expansion system for users with motor disabilities. In *ICADW*, pages 4–4, 2005.

[94] Stefano Pini, Sangmok Han, and David R. Wallace. Text entry for mobile devices using ad-hoc abbreviation. In *AVI*, pages 181–188, 2010.

[95] Claudiu Tanase, Ivan Giangreco, Luca Rossetto, Heiko Schuldt, Omar Seddati, Stéphane Dupont, Ozan Can Altiok, and T. Metin Sezgin. Semantic sketch-based video retrieval with autocompletion. In *Companion Publication of the 21st International Conference on Intelligent User Interfaces, IUI 2016, Sonoma, CA, USA, March 7-10, 2016*, pages 97–101, 2016.

[96] Wenbo Tao, Dong Deng, and Michael Stonebraker. Approximate string joins with abbreviations. *PVLDB*, 11(1):53–65, 2017.

[97] Arvind Arasu, Surajit Chaudhuri, and Raghav Kaushik. Transformation-based framework for record matching. In *ICDE*, pages 40–49, 2008.

[98] Arvind Arasu, Surajit Chaudhuri, Kris Ganjam, and Raghav Kaushik. Incorporating string transformations in record matching. In *SIGMOD*, pages 1231–1234, 2008.

[99] Arvind Arasu, Surajit Chaudhuri, and Raghav Kaushik. Learning string transformations from examples. *PVLDB*, 2(1):514–525, 2009.

[100] Jiaheng Lu, Chunbin Lin, Wei Wang, Chen Li, and Xiaokui Xiao. Boosting the quality of approximate string matching by synonyms. *ACM Trans. Database Syst.*, 40(3):15:1–15:42, 2015.

[101] Pengfei Xu and Jiaheng Lu. Top-k string auto-completion with synonyms. In *DASFAA*, pages 202–218, 2017.

[102] Feifei Li, Bin Yao, Mingwang Tang, and Marios Hadjieleftheriou. Spatial approximate string search. *IEEE Trans. Knowl. Data Eng.*, 25(6):1394–1409, 2013.

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

[104] Allie Development Team. Allie RDF Data. `http://data.allie.dbcls.jp/index_en.html/`, 2018.

[105] 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.

[106] Michael Ian Shamos and Dan Hoey. Geometric intersection problems.

[107] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[108] 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.

[109] The Awesome Java Contributors. Awesome Java frameworks, libraries and software. `https://github.com/akullpp/awesome-java`, 2018.

[110] Greg Little and Robert C. Miller. Keyword programming in java. *Autom. Softw. Eng.*, 16(1):37–71, 2009.

[111] Sougou Labs. Sougou Pinyin Dictionary. `https://pinyin.sogou.com/dict/`, 2006.

[112] Sougou Labs. Sougou Pinyin Dictionary. `http://www.sogou.com/labs/resource/w.php`, 2006.

[113] Warren Toomey. The Unix Archive. `https://wiki.tuhs.org/doku.php?id=source:unix_archive`, 2018.

[114] Grant Jenks. Python WordSegment. `http://www.grantjenks.com/docs/wordsegment/`, 2018.

[115] Sami Virpioja, Peter Smit, and Stig-Arne Grönroos. Morfessor. `http://morfessor.readthedocs.io/`, 2018.

[116] Romain Robbes and Michele Lanza. How program history can improve code completion. In *ASE 2008, 15-19 September 2008, L'Aquila, Italy*, pages 317–326, 2008.

[117] Anh Tuan Nguyen, Trong Duc Nguyen, Hung Dang Phan, and Tien N. Nguyen. A deep neural network language model with contexts for source code. In *SANER 2018, Campobasso, Italy, March 20-23, 2018*, pages 323–334, 2018.

[118] Vincent J. Hellendoorn and Premkumar T. Devanbu. Are deep neural networks the best choice for modeling source code? In *ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 763–773, 2017.

[119] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. Code completion with neural attention and pointer networks. In *IJCAI 2018, July 13-19, 2018, Stockholm, Sweden.*, pages 4159–4165, 2018.

[120] Avishkar Bhoopchand, Tim Rocktäschel, Earl T. Barr, and Sebastian Riedel. Learning python code suggestion with a sparse pointer network. *CoRR*, abs/1611.08307, 2016.

[121] Huizhong Duan, Yanen Li, ChengXiang Zhai, and Dan Roth. A discriminative model for query spelling correction with latent structural SVM. In *EMNLP-CoNLL 2012, July 12-14, 2012, Jeju Island, Korea*, pages 1511–1521, 2012.

[122] Jianfeng Gao, Xiaolong Li, Daniel Micol, Chris Quirk, and Xu Sun. A large scale ranker-based system for search query spelling correction. In *COLING 2010, 23-27 August 2010, Beijing, China*, pages 358–366, 2010.

[123] Daqing Hou and David M. Pletcher. An evaluation of the strategies of sorting, filtering, and grouping API methods for code completion. In *ICSM 2011, Williamsburg, VA, USA, September 25-30, 2011*, pages 233–242, 2011.

[124] Ziqi Wang, Gu Xu, Hang Li, and Ming Zhang. A probabilistic approach to string transformation. *TKDE*, 26(5):1063–1075, 2014.

[125] Thorsten Joachims. Optimizing search engines using clickthrough data. In *ACM SIGKDD 2002, July 23-26, 2002, Edmonton, Alberta, Canada*, pages 133–142, 2002.

[126] Johannes Schaback and Fang Li. Multi-level feature extraction for spelling correction. In *IJCAI-2007 Workshop on Analytics for Noisy Unstructured Text Data*, pages 79–86, 2007.

[127] T. Joachims. Making large-scale SVM learning practical. In B. Schölkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods - Support Vector Learning*, chapter 11, pages 169–184. MIT Press, Cambridge, MA, 1999.

[128] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N. Nguyen, and Danny Dig. API code recommendation using statistical learning from fine-grained changes. In *FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 511–522, 2016.

[129] Frode Eika Sandnes. Reflective text entry: A simple low effort predictive input method based on flexible abbreviations. In *DSAI 2015, Sankt Augustin, Germany, June 10-12, 2015*, pages 105–112, 2015.

[130] Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles A. Sutton. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51(4):81:1–81:37, 2018.

[131] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles A. Sutton. Learning natural coding conventions. In *FSE-22, Hong Kong, China, November 16 - 22, 2014*, pages 281–293, 2014.

[132] Valeriy Savchenko and Alexander Volkov. Statistical approach to increase source code completion accuracy. In *PSI 2017, Moscow, Russia, June 27-29, 2017*, pages 352–363, 2017.

[133] Naoaki Okazaki, Yoshimasa Tsuruoka, Sophia Ananiadou, and Jun'ichi Tsujii. A discriminative candidate generator for string transformations. In *EMNLP 2008, 25-27 October 2008, Honolulu, Hawaii, USA*, pages 447–456, 2008.

List of Publications

Journal Papers

- Sheng Hu, Chuan Xiao, and Yoshiharu Ishikawa. “An Efficient Algorithm for Location-Aware Query Autocompletion”, *The Institute of Electronics, Information and Communication Engineers (IEICE) Transactions on Information and Systems*, Vol. E101-D, No. 1, pp. 181-192, January 2018. (IEICE Best Paper Award 2018)
- Sheng Hu, Chuan Xiao, and Yoshiharu Ishikawa. Scope-aware Code Completion with Discriminative Modeling. *Journal of Information Processing*, 2019. (accepted for publication)

International Conference/Workshop Papers

- Sheng Hu, Chuan Xiao, Jianbin Qin, and Yoshiharu Ishikawa, Qiang Ma. Query Autocompletion for Prefix-Abbreviated Input. *ACM SIGMOD International Conference on Management of Data*, 2019. (accepted for publication)

国内学会発表（査読なし）

- Sheng Hu, Chuan Xiao, Yoshiharu Ishikawa, “Efficient Autocompletion with Error Tolerance”, 第8回データ工学と情報マネジメントに関するフォーラム (DEIM 2016), 2016年3月.（学生プレゼンテーション賞受賞）
- 胡 晟, 肖 川, 石川 佳治,「略記問合せに対する効率的な問合せ自動補完」, 第9回データ工学と情報マネジメントに関するフォーラム (DEIM 2017), 2017年3月.
- 胡 晟, 肖 川, 石川 佳治,「識別モデルを用いたスコープを意識したコード補完」, 第11回データ工学と情報マネジメントに関するフォーラム (DEIM 2019), 2019年3月.（学生プレゼンテーション賞受賞）