Efficient Query Autocompletion for Abbreviated Queries

Sheng HU†, Chuan XIAO††, and Yoshiharu ISHIKAWA†

† Graduate School of Information Science, Nagoya University
Furo-cho, Chikusa-ku, Nagoya, 464–8601 Japan
†† Institute for Advanced Research, Nagoya University
Furo-cho, Chikusa-ku, Nagoya, 464–8601 Japan
E-mail: †hu@db.ss.is.nagoya-u.ac.jp, ††chuanx@nagoya-u.ac.jp, †††ishikawa@is.nagoya-u.ac.jp

Abstract In modern search engines applications, query autocompletion is a useful feature which can save many keystrokes from inputting the entire query. As the autocompletion features of search engines have become more and more popular, many variations of query autocompletion appear in various scenarios, e.g. error-tolerant autocompletion and common phrase autocompletion. In this paper, we propose a novel autocompletion paradigm called *Query Autocompletion for Abbreviated Queries*, and propose a prefix-network based index to solve this issue efficiently. We conduct some experiments with classical trie-based method using a large dataset. Experiments show that our method outperformed the classical method.

Key words Query autocompletion, prefix-network, text database search.

1. Introduction

Query autocompletion has become a useful feature and been studied for a long time since the appearance of search engines. As this feature can save many key strokes from inputting the entire query, it has been considered a standard function of search engines. Besides traditional text search engines, query autocompletion also has wide application domains including common shells, integrated development environments (IDEs) and special text abbreviation search. Also, as query autocompletion usually needs to response the incoming query in a very short time and suffer from heavy query throughput, query autocompletion on modern databases need to be more efficient to improve the user's experience. An autocompletion query often returns all the objects begins with the textual prefix. More useful scenarios are, users want to search for the most related objects, and then this issue changes into the autocompletion top-k query, which returns objects after calculating and ranking objects according to their textual popularities.

Autocompletion has become a standard feature these years. This important feature can return the search results when user is typing the partial keywords letter by letter and accordingly pop a pull-down list to show the most intended items. After that user just need choose the correct entry to save the tedious inputting time.

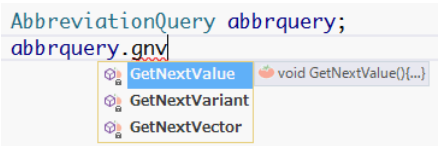

Figure 1 A Motivating Example

In this paper, we propose a novel query autocompletion features so that users may input queries like acronyms or abbreviation instead of letter by letter. We give an motivating example in Fig. 1. For example, in IDE, users may type "GNV" to get the function "GetNextValue()". We call this kind of problem *Abbreviated Query Autocompletion*. This problem is quite challenging because currently there's no approach to solve it very well. The most similar approach is to model it using common subsequence techniques, but we find that common subsequence model cannot solve it efficiently because we will not consider every subsequence like "NV" or "GV" can get the function "GetNextValue()". To model this novel query more suitably, we propose a novel problem definition to define "Abbreviated Query Autocompletion".

Additionally, it is also very hard to answer Abbreviated Query efficiently using existing techniques. One approach is to conduct the search on a prefix trie, but for every character, it needs do the exhaustive search throughout current subtree recursively. This method will result in extremely expensive computation when a query contains too many characters.

To deal with this problem, we propose a prefix-network index structure called PNI. It transforms a trie into a prefix-network by merging trie nodes which share the same special characters. It also attaches a unique bit array as an indicator on each network node to do efficient pruning when traverse the network. In search phase, the given query traverse the prefix-network from the root node and check the bit array indicator to decide if stop traversing. We also devise an efficient result fetching algorithm to fetch the results by doing interval intersection. Besides that, we also utilize the network's nodes subsumption properties to speed up the result fetching time and duplicate-removal time.

In this paper we mainly study the autocompletion of abbreviation

query and then extend our index to support processing top-k query.

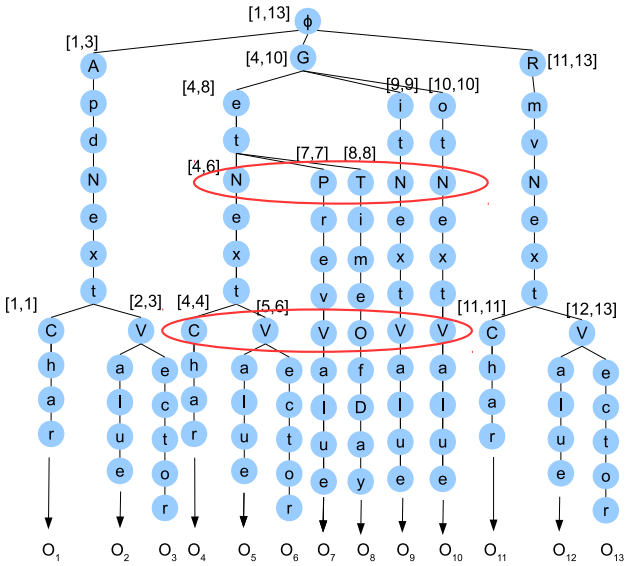

Figure 2 Merge Uppercase Node

Our main contributions are summarized as follows.

（1） Propose a novel query autocompletion feature Abbreviation Query and give the problem definition.

（2） Propose a novel prefix-network index PNI to deal with the Abbreviation Query efficiently.

（3） Propose an efficient results fetching method which can fetch the results by intersecting intervals.

（4） Devise a method to speedup the result fetching time and duplicate-removal time based on network nodes subsumption relationships.

（5） A simple experiment has be conducted to compare with existing method and the results show our method outperforms the classical one.

The rest of our paper is organized as follows. In section 2., we show some related works. In section 3., we will give the formal problem definition. In section 4., we will give the fundamental index structure called PNI and show our efficient search and result fetching algorithm. In section 5., we use a simple experiment to verify that our algorithm outperforms the classical method.

2. Related Work

2.1 Trie-based Autocompletion

In order to obtain the strings with the same prefix efficiently, trie-based index becomes very popular and is adopted as the most efficient way to implement autocompletion among the current research works. Figure. 2 shows a standard trie built based on data in Table 1.

2.1.1 Trie-based Text Search

Efficient string prefix search based on trie index was first proposed in Baeza-Yates et al.'s work [1] in 1996. In his work, he also proposed a patricia tree approach to save the index space. After that, community of information retrieval begins to focus on autocompletion using trie index. A large amount of studies [2], [11], [12] have been done to improve the autocompletion queries' qualities in the past decades. Besides that, some works [8], [9] focus on reducing the index size by various compression techniques in order to fit the index into server's main memory. The basic index structure used in these works is the classical trie structure showed in Fig. **??**. However, these works mainly focus on improving the autocompletion results qualities by ranking and calculate relevances carefully instead of focusing on search efficiency.

2.1.2 Error-Tolerant Query Autocompletion

Error-Tolerant Autocompletion (ETA) [6] studied the problem that when a user issues a partial query which contains some typos, the search engine can still identify the intended query and give the correct answers. META [6] put forward by Deng et al. is considered the state-of-the-art work currently. In META, they proposed a matching-based framework, which computes the answers based on matching characters between queries and data efficiently. Besides that, some works [3], [4], [10], [13], [14] also solve the ETA problem using various techniques.

2.2 Tolerant Retrieval and Wildcard Query

Tolerant Retrieval problem [5] comes from information retrieval community and has been studied for many years. The query model of this problem called wildcard query. This means a query such as n*g*y*, and seeks documents containing any term that includes all the three characters in sequence, e.g, nagoya, nergy. The * symbol can be replaced by any characters. This kind of query is especially useful when users are uncertain about how to spell query term, or want to seek documents containing any of the term. A classical approach to solve this problem efficiently is to use a special index called permuterm index [7]. It attaches a special symbol $ after each string and them calculate variants of each string by rotating the string by one step. And then index all these variants into a tree-like index to do efficient search.

3. Problem Definition

Below we'll give our data model and query model.

3.1 Data Model

Suppose D is a textual database, which consists of a lot of objects $\{O_1, O_2, ..., O_i, ..., O_n\}$. Each object is defined as a $tuple = \{O.id, O.string, O.staticscore\}$, where $O.id$ is the object id which is a unique number, $O.string$ is a single string which represents object's name, and $O.staticscore$ is the object's popularity.

In our running example, we use a textual database showed in Table. 1 for easier demonstration.

Table 1 Textual Database D

ObjectID	String	Popularity
O_1	ApdNextChar	0.4
O_2	ApdNextValue	0.9
O_3	ApdNextVector	0.9
O_4	GetNextChar	0.7
O_5	GetNextValue	1.0
O_6	GetNextVector	0.5
O_7	GetPrevValue	0.4
O_8	GetTimeOfDay	0.1
O_9	GitNextValue	0.1
O_{10}	GotNextValue	0.1
O_{11}	RmvNextChar	1.0
O_{12}	RmvNextValue	0.3
O_{13}	RmvNextVector	0.3

3.1.1 Token-based Data Preprocessing

We'll use a tokenizer to do the data preprocessing in order to build our index more conveniently during subsequent steps. We use the tokenizer to process every single string and identify every single word in this string and change every word's first letter into uppercase and then concatenate them together again.

[Example 1] For a string "getnextvalue", we first identify every single word "get", "next", "value", transform them into "Get", "Next", "Value" and at last concatenate them as "GetNextValue".

3.1.2 Intra-word Neighborhood Relationship

[Definition 1] (Intra-word Neighborhood Relationship) In a single word from a string, every adjacent character has the intra-word neighborhood relationship. When two characters a and b has the intra-word neighborhood relationship, we denote it as $a|b$.

[Example 2] In a string "GetNextValue", we take the word *Get* and we have $G|e$, $e|t$. Similarly, in *Next* we have $N|e$, $e|x$, $x|t$, in *Value*, we have $V|a$, $a|l$, $l|u$, $u|e$.

3.1.3 Inter-word Neighborhood Relationship

[Definition 2] (Inter-word neighborhood Relationship) In two adjacent single words from a string, previous word's every character has the intra-word neighborhood relationship with the last word's uppercase letter. When two characters a and b has the intra-word neighborhood relationship, we denote it as $a||b$.

[Example 3] In a string "GetNextValue", we take the word *Get* and *Next*, so we have $G||N$, $e||N$, $t||N$. Similarly, in *Next* and *Value* we have $N||V$, $e||V$, $x||V$, $t||V$.

3.2 Match an Abbreviated Query

Given a abbreviated query Q, and a data string s, when Q and s satisfy the following three conditions, s matches Q. When s matches Q, we denote it as $Q \preceq s$.

(1) Q is a subsequence of s.

(2) Q's first character is identical with s's first character.

(3) Every two adjacent characters in Q has intra-word neighborhood relationship *or* inter-word neighborhood relationship in s.

Note that we consider uppercase and lowercase is non-sensitive in the match between Q and s.

[Example 4] Given a query Q = *gnv*. String s = "Gnv Corp." matches Q because it satisfies first two conditions and also satisfies the third condition that *gnv* in s has inter-word neighborhood relationship which means $g||n$, $n||v$. While string t = "GetMyNvidia" doesn't match match Q, because although it also satisfies the first two conditions, the three letters *gnv* appearing in t doesn't have any of the two relationships.

3.3 Abbreviated Query

Autocompletion abbreviated query runs as follows: When user issues a query, with every single stroke of the query string, the string typed in so far will be sent as the input of the index. After that, our index return objects that match the query.

3.4 Top-k Abbreviated Query

Top-k query runs as follows: When user issues a query, with every single stroke of the query string, the string typed in so far will be sent as the input of the index. And then, our index will return the most relevant k objects that match the query according to textual popularity.

3.5 Problem Statement

3.5.1 Abbreviated Query

Based on above specification, we give the abbreviated query autocompletion problem formulation.

[Definition 3] (abbreviated query autocompletion) Given a query Q, a textual database $D = \{O_1, O_2, ..., O_n\}$,each object consists of one part $O.str$, it returns all the objects $R \subseteq D$ such that each object in R satisfies $Q \preceq O.str$.

3.5.2 Top-k Abbreviated Query

We also give the definition of Top-k Query.

[Definition 4] (top-k abbreviated query autocompletion) Given a query Q, a textual database $D = \{O_1, O_2, ..., O_n\}$, each object consists of two parts $O.str$ and $O.staticscore$, it returns the top-k objects $R \subseteq D$ such that each object $O \in R$ satisfied $Q \preceq O.str$, sorted by the textual static score.

4. Prefix-network Index

4.1 Index Structure

We introduce our index structure from the classical trie index showed in Fig. 2. Considering fetching the object results conveniently, we attach an object id interval $[a, b]$ on each trie node. If we traverse the trie and find a trie node, we can directly use the object id range $[a, b]$ to collect the results by random access instead of traversing the subtree recursively.

4.1.1 Merge Uppercase Node

Although the query match is non-sensitive to uppercase character, we can deem uppercase character as a beginning of a single word and do the search efficiently. Here, for convenience, we call the node attached with uppercase character *Uppercase Node*. As we have highlighted using the red circle in Fig. 2, we decide to merge

the uppercase nodes if they share the same uppercase lowest common ancestor. For example, in Fig. 2, the upside red circle highlights five uppercase nodes N, P, T, N, N, and then merge the three N node into one node. This will disorganize the original trie order and regenerate a new subtree. Then we can find that downside red circle highlights 6 uppercase nodes C, V, V, O, V, V, but the second V node doesn't share the same uppercase lowest common ancestor with the others so we just merge the other 3 V nodes. Note that we also merge the object id intervals to keep the intervals updated. After merge all the qualified uppercase nodes, we obtain a prefix-network described in Fig. 3.

4.1.2 Bit Array Indicator

To deal with string matched by satisfying inter-word neighborhood relationship, we attach a bit array indicator on each network node to judge if any uppercase nodes appear under the current network node and help us to locate such uppercase nodes efficiently. In Fig. 3, we use a character list in orange to represent what uppercase nodes will appear under current network node, but use bit array in implementation to save space. In Fig. 3, when we locate on node G in the first layer, the character list N, P, T shows that there are 3 uppercase nodes under current G node. Then we locate on G's child node i, the character list N shows that there's only one upper case node N under this i node because we only have one string *GitNextValue* in this dataset.

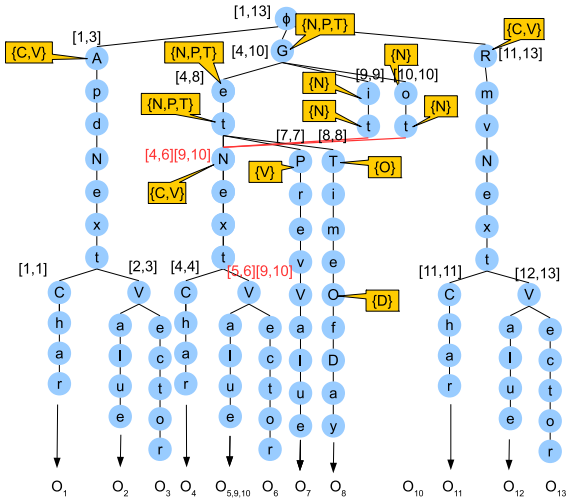

Figure 3 Prefix-network Index with Bit Array Indicator

4.1.3 Build the Index

Below we give the algorithm of building the prefix-network index.

- Algorithm 1 shows a whole procedure of building the prefix-network index. Line 2–3 shows that we insert the whole dataset iteratively, but note that we have sorted the dataset strings in alphabetical order in advance to make it convenient for the interval update. In line 4 we complete calculating the bit array indicator recursively.

Algorithm 1: BUILDPREFIXNETWORKINDEX(T, D)

Input: T is a network node, D is a textual dataset

1 $T \leftarrow RootNode$
2 **for** *each string* $s \in D$ **do**
3 INSERTSTRING(T, s)
4 TRAVERSEBITINDICATORSHORTCUT(T)
5 TRAVERSESUBSUMEINDICATORSHORTCUT(T)

Algorithm 2: INSERTSTRING(s, T)

Input: s is a single string with a unique integer id, T is a network node

1 $node \leftarrow T$
2 UPDATENODEINTERVALANDSCORE(*node*,*s.id*,*s.score*)
3 **for** *each character char in s* **do**
4 **if** *char is UpperCase* **then**
5 node.BitIndicator.add(char)
6 $FoundOrNot \leftarrow node.shortcut.find(char)$
7 **if** $FoundOrNot = NotFound$ **then**
8 $node.shortcut[char] \leftarrow NewNode$
9 $node \leftarrow node.shortcut[char]$
10 **else**
11 $FoundOrNot \leftarrow node.children.find(char)$
12 **if** $FoundOrNot = NotFound$ **then**
13 $NewNode.shortcut \leftarrow node.shortcut$
14 $node.children[char] \leftarrow NewNode$
15 $node \leftarrow node.children[char]$
16 UPDATENODEINTERVALANDSCORE(*node*,*s.id*,*s.score*)

Algorithm 3: UPDATENODEINTERVALANDSCORE(T, $s.id$, $s.score$)

Input: T is a network node, $s.id$ is a string integer id

1 $node \leftarrow T$
2 **if** *node.minid = -1* **then**
3 $node.minid \leftarrow s.id$
4 **if** *node.maxid = -1* **then**
5 $node.maxid \leftarrow s.id$
6 node.interval.pushback($IntervalTriplet < s.id, s.id, s.score >$)
7 **if** $s.id > maxid$ **then**
8 **if** $sid = maxid+1$ **then**
9 node.interval.lastPair.upperbound++
10 **if** $s.score > node.interval.lastPair.score$ **then**
11 node.interval.lastPair.score ← s.score
12 **else**
13 node.interval.pushback($IntervalTriplet < s.id, s.id, s.score >$)
14 $maxid \leftarrow s.id$
15 $node.maxscore \leftarrow MAX(node.maxscore, s.score)$

- Algorithm 2 shows how to insert a string object into our index. In line 2 we first update the root node interval. As uppercase character is special so when a uppercase character comes in, we begins to build the bit array indicator on each parent node of uppercase node in line 5. In line 6, we first create a map called *shortcut* only on uppercase node as an implicit "shortcut" to directly link to the below uppercase node for efficient search. Note that lowercase node only store a pointer as a copy of its nearest or lowest ancestor uppercase node. Line 8 shows that if we fail to find an existing node, we will insert a node. Then, in line 9 we update current node to keep the loop going. In line 11 we begin to insert the non-special lowercase node using a map called "children". In line 14 if we fail to find an existing child node, we will insert a new one. And in line 15 we update the node to keep the loop going. Finally, in line 16 we update the interval for every node we have accessed.

- Algorithm 3 shows how to update each node's string id interval when inserting strings. In line 2 and line 4 we set the interval minimum and maximum value to both -1 to represent the interval is empty. In line 6 shows that if we encounter an empty interval we can insert the string id into the interval immediately. Line 7–13 shows that if current string id is adjacent with current maximum id, we can simply plus 1 in the last the interval's upper bound, otherwise we will insert a new standalone interval into the interval vector. Last, we update the maximum id in line 14.

- Algorithm 4 and algorithm 5 show how to complete calculating each node's bit array indicator recursively. In algorithm 5 line 6, we do union operation between a parent node and its each child to finish the bit array calculation.

- Algorithm TraverseSubsumeIndicatorShortcut in algo 1 is similar with Algo 4, we omit it in details due to the paper space limit.

Algorithm 4: TraverseBitIndicatorShortcut(T)

```
Input: T is a network node
1 node ← T
2 if node.shortcut = NULL then
3     return
4 for each subnode in node.shortcut do
5     TraverseBitIndicatorShortCut(subnode)
6     TraverseBitIndicatorChildren(subnode)
```

Algorithm 5: TraverseBitIndicatorChildren(T)

```
Input: T is a network node
1 node ← T
2 if node.children = NULL then
3     return
4 for each subnode in node.children do
5     TraverseBitIndicatorChildren(subnode)
6     node.BitIndicator | = subnode.BitIndicator
```

4.2 Search with Abbreviation Query

Given an abbreviation query, we will traverse the prefix-network index from the root node first. We try to match each character in the query with each character in current network node's children and shortcut. If it successfully matches, we call this kind of node *Active Node*. When the search traverses the network and access a node, it will see the bit array indicator to judge if the short cut really has the query character. If the short cut really has the query character, we can use the short cut to efficiently set the corresponding node active. The active nodes in our prefix-network index will continue expanding until a query ends or the query fails to find at least one match character in its children or short cut nodes. After the query ends, if there are nodes remaining active, we will fetch the string results using these active nodes. As each active node has a path from the root and every nodes it had accessed to current node, we can utilize the intervals store in these nodes to obtain the results. Note that we only need some of these intervals, more precisely, we only need those nodes who is the most far from its lowest ancestors. Then we can intersect these nodes' intervals to obtain the final intervals. We give detailed algorithm in Algo 6. We also devise a merge-sort-like algorithm IntersectInterval to efficiently intersect the interval segments, we omit the details of IntersectInterval due to the paper space limit. After that, we can simply fetch those results strings by random access. Figure. 4 gives an example of abbreviated query search.

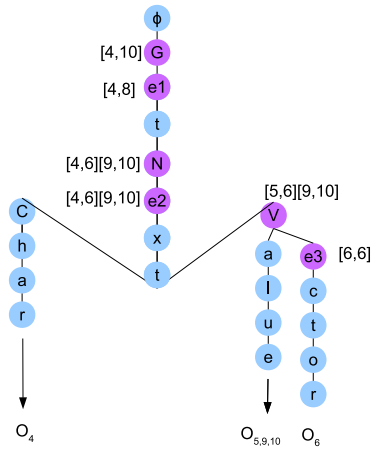

Figure 4 Search with Abbreviation Query

［Example 5］ See Fig. 4, given a abbreviation query "geneve". So after we traverse the prefix-network index, we can obtain an active node $e3$ under the node V. We use color purple to represent those node it has gone through. The path it goes through is $G - e1 - N - e2 - V - e3$. Their intervals are:

- G : [4, 10]
- $e1$: [4, 8]
- N : [4, 6][9, 10]
- $e2$: [4, 6][9, 10]
- V : [5, 6][9, 10]
- $e3$: [6, 6]

, respectively. As node $e1$ is absolutely subsumed by its parent G, we don't need to intersect node G's interval. Similarly, we only need to intersect nodes $e1$, $e2$, $e3$'s 3 intervals which the result is $[4,8] \cap [4,6][9,10] \cap [6,6] = [6,6]$. [6,6] only contains O_6 which is *GetNextVector* and it's the only match of abbreviation query "geneve".

4.2.1 Subsumption Relationship-based Intersection Optimization

We propose a technique which optimizes the intersection cost by utilizing the subsumption relationship in this prefix-work. We show the algorithm below. We also show the basic idea with an illustrative example.

［Example 6］ Consider the path $G - e1 - N - e2 - V - e3$ in example 5. Notice that the node $e2$'s in this path interval is [4,6][9,10], and the interval of node V is [5,6][9,10], and the interval of node $e3$ is [6,6]. As the node $e3$ has parent-child relationship with node V, naturally the interval of node $e3 \subseteqq V$. However we observe that there's also node $V \subseteqq e2$. So we can deduce that the node $e3 \subseteqq e2$. This means when we calculate the intersection of $e1$ $e2$ $e3$ in example 5, it is not necessary to intersect node $e2$'s interval anymore. As a result, we only need to calculate $[4,8] \cap [6,6]$.

For any uppercase node whose interval is completely subsumed by its ancestor node, we mark "subsumed" on that ancestor node's link to this uppercase node. Actually, we only need to mark those uppercase nodes who appear in the bit array indicator because only those who show up in the array indicator can be searched and become active. We do this by invoking TRAVERSESUBSUMEINDICATOR showed in Algo. 1.

4.2.2 Subsumption Relationship-based Duplicate-removal Optimization

We can also utilize the subsumption relationship to help to remove those possible duplicates. We also show the basic idea with an illustrative example.

［Example 7］ See Fig. 3, given an abbreviation query "get". After we traverse the prefix-network index, we can obtain an active node t from path $G - e - t$ and an active node T from path $G - e - T$. The second active node is obtained because when search goes through $G - e$, node e's bit array indicator shows that there's a T below so the node T also become active. We observe that these active nodes' intervals are [4,8] and [8,8] respectively. As we need to fetch all these active nodes results, we do $[4,8] \cup [8,8] = [4,8]$. As node $T \subseteqq$ node t, the interval of node T becomes completely duplicate in this UNION operation. To solve this problem, we need to check the subsumption relationship especially when a single active nodes expands into two active nodes, in this example, we will check the node t and T subsumption relationship when active node e "splits" into these two active nodes. We can find that node $T \subseteqq$ node t, and if query ends here, we will abandon active node T to do the duplicate removal.

About the "active node split", we give the definition below.

［Definition 5］（Active Node Split） Given an active node n, when query's next character s comes in, n finds the character s matched both in its children and shortcut map, we call this active node n split into two active nodes.

Algorithm 6: SEARCHWITHABBREVIATIONQUERY(s, T)

Input: s is an abbreviation query, T is the root network node
Output: $NodeSet$ is the final active nodeset

```
1  NodeSet ← NULL
2  bNodeSet ← NULL
3  NodeSet.push(T)
4  for each character char in s do
5      for each node in NodeSet do
6          ChildrenFoundOrNot ← node.children.find(char)
7          ShortcutFoundOrNot ← node.shortcut.find(char)
8          BitIndicatorFoundOrNot ← node.bitIndicator.find(char)
9          SubIndicatorFoundOrNot ←
             node.subsumeIndicator.find(char)
10         if ChildrenFoundOrNot = NotFound AND
             BitIndicatorFoundOrNot = NotFound then
11             continue
12         else
13             if ChildrenFoundOrNot = Found then
14                 Found.history ← node.history
15                 bNodeSet.push(node.children[char])
16             if BitIndicatorFoundOrNot = SCFound AND
               ShortcutFoundOrNot = Found then
17                 SCFound.history ← node.history
18                 if SubIndicatorFoundOrNot = NotSubsume then
19                     midIntersect ← node.IntersectIntervals
20                     (node.interval, SCFound.history)
21                     if midIntersect is EMPTY then
22                         node.history.clear()
23                     else
24                         SCFound.history ← midIntersect
25                 bNodeSet.push(node.shortcut[char])
26             node.history.clear()
27     NodeSet ← NULL
28     for each node in bNodeSet do
29         NodeSet.push(node)
30 for each acnode in nodeSet do
31     finalIntersect ←
         activenode.IntersectInterval(ac.interval, acnode.history)
32     acnode.history ← finaIntersect
33 return nodeSet
```

Whenever an active node split into two active nodes, we begin to trace these two nodes and check if they have subsumption relationship, if they have and query ends here, we can abandon the subsumed node to remove the duplicate. If they have subsumption relationship but the query doesn't end, we will trace these two nodes' expansion as query's next character comes in, if these two nodes keep active until query ends and:

- These two nodes only expand through children map.
- Upside node always subsumes the downside node's lowest uppercase ancestor.

We can remove the downside node safely when query ends.

We give our search algorithm and a running example below.

4.3 Search with Top-*k* Abbreviation Query

Algorithm 7: FETCHTOPKRESULTSWITHACTIVENODESET(*NodeSet*)

Input: *NodeSet* is an active node set
Output: *Results* is the final topk results

```
1  i ← 0
2  TopkPriorityQueue ← NULL
3  for each active node node in NodeSet do
4      TopkPriorityQueue.push(node)
5  while i < k AND TopkPriorityQueue is not empty do
6      topElem ← TopkPriorityQueue.pop()
7      if topElem is a node then
8          Expand topElem into the interval segments
9          for each interval segment seg in topElem do
10             TopkPriorityQueue.push(seg)
11     if topElem is a interval segment then
12         for each stringobject locate in topElem do
13             TopkPriorityQueue.push(stringobject)
14     if topElem is a stringobject then
15         Results.push(topElem)
16         i++
17 return Results
```

Top-*k* query processing is quite different from the non-top-*k* query and efficiently fetch the top-*k* is even an challenging work. To fetch the most relevant top-*k* data string, we need to rank the results by their static score. In our index, we store a upper bound static score for every node's every interval segment (e.g. [4,6]:1.0 ;[9,10]:0.1.) in the prefix-network and use the early termination technique to fetch the results efficiently. Also, as showed in alg. 3, we store an overall upper bound score on each node (e.g. for node *N*, we store a maximum score 1.0).That means, we will fetch the results from the highest static score node's highest interval segment which is considered the most relevant. We use a priority queue to rank the top-*k* results and if any node's upper bound score, or any node's any interval segment score is lower than the *k*-th score, we can easily prune this node or remaining interval segments to avoid unnecessary operations.

Algorithm 7 shows the details of top-*k* abbreviation query search.

5. Experiment

5.1 Experimental Setting

In this section, we will give our experimental results to show the performance and compare it with other works showed below.

（1） Classical trie. Classical trie is used in text retrieval and efficient autocompletion these years. We donote it as CT for simplicity.

（2） PNI. With the novel index structure and fetching method we proposed, we will give the comparison between the other work, and we denote it as PNI for simplicity in our experiment.

The experiments were implemented using a PC with Intel Xeon CPU E5620 (2.40GHz), 32GB memory, Ubuntu14. All the algorithm are implemented in C++ and are run as the in-memory index.

5.1.1 Datasets

We use a large dataset and show the statistics below. Dataset

Table 2 Statistics of the Dataset

Dataset	DBLP
number of objects	9316195
Dataset Size	135MB
Max Text Length	12
Average Text Length	9.40

DBLP contains bibliography records in Computer Science, and we extract 9.3 million worldwide author names from the records.

5.1.2 Queryset

We generate queries by randomly selecting 1000 strings and generate the abbreviation query for each string. We ignore all the spaces in multikeyword and contecate them into one single string. We remove meaningless symbols and only remain the alphabetic letters. In our experiment, we mainly measure:

（1） Abbreviation Query Response Time. It represents the whole time to execute a abbreviation query and fetch all the qualified results.

（2） Top-*k* Abbreviation Query Response Time. It represents the whole time to execute a top-*k* abbreviation query and fetch the most relevant *k* qualified results.

5.2 Abbreviated Query Response Time

In Fig. 5 we show the abbreviated query response time using dataset DBLP. We vary the query string prefix length from 1 to 8 to see the runtime. When prefix length is longer, CT method becomes much slower while our PNI method becomes faster. This is because CT method costs too much on iteratively search the whole trie while our PNI almost has no time cost on search of index but

mainly on fetch results phase. When prefix length becomes longer, qualified strings will reduce drastically and our PNI method will become faster. We also observe that in the most common case, when prefix length equals 2 or 3, our PNI method is 100 times and 1000 times faster than the CT method respectively.

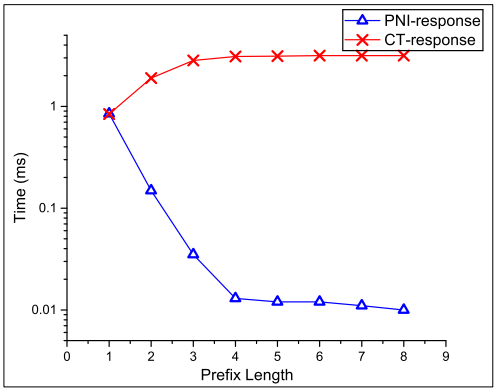

Figure 5 DBLP, Query Response Time

5.3 Top-*k* Abbreviated Query Response Time

In Fig. 6 we show the top-*k* abbreviated query response time. We set $k = 5$ and vary the query string prefix length from 1 to 8 to see the results. Similarly, when prefix length is longer, CT method becomes much slower while our PNI method becomes faster after a wave when prefix length equals 2. CT method still costs too much on trie traversal search while our PNI is fast at index search and also has decent performance of top-*k* results pruning. Additionally, we also record the index size of both methods. CT costs 5.3Gb while PNI costs 2.27 Gb when $k = 5$. CT also doesn't support a dynamic *k* value become it needs to materialize a top-*k* rank list on each node of the trie which will introduce prohibitive memory cost under a large *k* value. Compared with that, our PNI supports a dynamic *k* and the index size will keep constant when *k* changes.

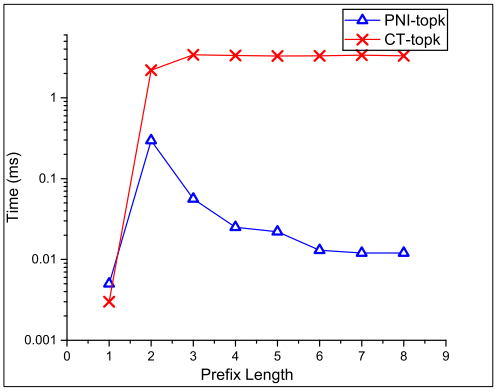

Figure 6 DBLP, Top-*k* Query Response Time

6. Conclusion

In this paper, we propose the definition of abbreviation query autocompletion. Then we design an efficient index structure along with a search algorithm and a top-*k* fetch algorithm. After that we give a simple evaluation using a large dataset.

Acknowledgments

This research is partly supported by the Grant-in-Aid for Scientific Research, Japan (16H01722, 26540043).

References

[1] R. A. Baeza-Yates and G. H. Gonnet. Fast text searching for regular expressions or automaton searching on tries. *Journal of the ACM (JACM)*, 43(6):915–936, 1996.

[2] Z. Bar-Yossef and N. Kraus. Context-sensitive query autocompletion. In *Proceedings of the 20th International Conference on World Wide Web*, pages 107–116. ACM, 2011.

[3] I. Cetindil, J. Esmaelnezhad, T. Kim, and C. Li. Efficient instant-fuzzy search with proximity ranking. In *2014 IEEE 30th International Conference on Data Engineering*, pages 328–339. IEEE, 2014.

[4] S. Chaudhuri and R. Kaushik. Extending autocompletion to tolerate errors. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 707–718. ACM, 2009.

[5] P. R. Christopher D. Manning and H. Schutze. *Introduction to Information Retrieval*. Cambridge University Press, 2009.

[6] D. Deng, G. Li, H. Wen, H. Jagadish, and J. Feng. Meta: An efficient matching-based method for error-tolerant autocompletion. *Proceedings of the VLDB Endowment*, 9(10), 2016.

[7] P. Ferragina and R. Venturini. The compressed permuterm index. *ACM Transactions on Algorithms (TALG)*, 7(1):10, 2010.

[8] R. Grossi and G. Ottaviano. Fast compressed tries through path decompositions. *Journal of Experimental Algorithmics (JEA)*, 19:3–4, 2015.

[9] B.-J. P. Hsu and G. Ottaviano. Space-efficient data structures for top-k completion. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 583–594. ACM, 2013.

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

[11] M. Shokouhi and K. Radinsky. Time-sensitive query autocompletion. In *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 601–610. ACM, 2012.

[12] S. Whiting and J. M. Jose. Recent and robust query auto-completion. In *Proceedings of the 23rd International Conference on World Wide Web*, pages 971–982. ACM, 2014.

[13] C. Xiao, J. Qin, W. Wang, Y. Ishikawa, K. Tsuda, and K. Sadakane. Efficient error-tolerant query autocompletion. *Proceedings of the VLDB Endowment*, 6(6):373–384, 2013.

[14] X. Zhou, J. Qin, C. Xiao, W. Wang, X. Lin, and Y. Ishikawa. Beva: An efficient query processing algorithm for error-tolerant autocompletion. *ACM Transactions on Database Systems (TODS)*, 41(1):5, 2016.