Trie-based Similarity Search and Join

Jianbin Qin[†]   Xiaoling Zhou[†]   Wei Wang[†]   Chuan Xiao[‡]

[†]University of New South Wales, Australia

{jqin, xiaolingz, weiw}@cse.unsw.edu.au

[‡] Nagoya University, Japan

{chuanx}@itc.nagoya-u.ac.jp

ABSTRACT

Driven by the increasing demands from applications such as data cleansing, integration, and bioinformatics, approximate string matching queries have gain much attention recently. In this paper, we present the design and implementation of a trie-based system which supports both string similarity search and join based on our recent work [23].

1. INTRODUCTION

1.1 Background

With the increasing need to store textual data in modern applications, it poses a stringent demand to manage string data and support efficient query processing for them. An fundamental type of query is the *string similarity query*, which matches database strings that are similar but not identical. Reasons for using approximate rather than exact matching in queries are mainly due to the allowance of typographical errors, or existence of multiple conventions for the same entity such as names or addresses.

String similarity queries include string similarity search and string similarity join. Given a set of string objects R in database, a search string Q, a similarity function and a similarity threshold t, string similarity search finds all the strings in R that has similarity of at least t with Q. If we do string similarity search by treating every string object in another set S as Q, this problem is called string similarity join. More specifically, string similarity join finds every pair of strings that have similarity at least t where the two similar strings in one pair come from two different sets.

There are many applications of string similarity queries. For example, in data integration, similarity join of customer records can help identify multiple profiles of the same customers. In data cleansing, similar objects can be found and inconsistencies and redundancies can be eliminated using string similarity search [2]. In bioinformatics, similarity queries can be used to find similar genome sequences to help finding a cure for diseases [8, 13]. Similarity queries have also been used to perform various Web mining activities, including duplicate web page detection [1, 22].

Consequently, there has been much interest to solving the string similarity queries efficiently. However, this problem is much more challenging than the exact string matching problem. This is mainly because the introduction of complex similarity functions (such as edit distance) and the potentially substantial increase of result size. In addition, the solution is expected to be able to scale to millions or billions of objects as required by applications such as industrial search engines (Google or Bing). For them, there are millions of users querying the service at any given time, which results in millions of similarity queries concurrently to the search system, yet all expecting to receiving results in near real-time.

1.2 Existing Work

In terms of similarity functions that are used to measure the similarity of two objects, commonly used ones includes overlap, Jaccard, and cosine similarity functions, and the edit distance function. In the following, we briefly survey existing work on string similarity search and join problems with a given edit distance threshold, as they are widely used to captures typographical errors for text strings or documents. Readers are referred to recent surveys and tutorials for a more comprehensive treatment of the existing work [6, 17, 7, 10].

There are several major approaches to deal with string similarity queries with edit distance constraints. A prevalent approach is based on the *filter-and-verify* framework. For example, the gram-based [21] and chunk-based methods [12, 18, 18] transform strings into sets of grams or chunks first, and then leverage existing search or join methods for sets to answer the queries. Additional filtering can be performed to further improve the performance, for example, count filtering, positional filtering and length filtering [5], prefix filtering [2], location-based and content-based mismatch-filterings [21], error-estimation-based filtering [12], and vchunk number filtering [18]. Finally, similarity functions are computed for each pair in the candidate set to verify if they are true match. A major disadvantage of these methods is that they are inefficient when dealing with datasets with very short strings, as the selectivities of the grams or chunks are become rather poor.

Another approach is based on enumeration. Enumeration-based methods generate all the possible strings that are within similarity threshold t from given strings, hence transforming a similarity query to an exact match query. The naive method typically does not work in practice as the num-

ber of enumeration is prohibitively large. Several approaches based on deletion neighbourhood [14], and partitioning [19] have been proposed, which work extremely well for small edit distance thresholds. The disadvantage of these methods is that they become resource intensive to enumerate all the variants when strings are very long or the edit distance threshold is very large.

Yet another approach is based on tries [3]. It indexes the strings in the database in a trie and support edit similarity search by incrementally maintaining a set of candidate nodes, or *active nodes*, during the query processing. [16] introduces a trie-based method to support edit similarity joins efficiently with additional pruning techniques such as sub-trie pruning. Most recently, we have proposed two novel trie-based approximate prefix match methods based on pre-computation [20, 23].

1.3 Outline

In this paper, we present a trie-based system for supporting string similarity search and join efficiently over large scale datasets. We use the trie structure for indexing as it has the following advantages: (1) it typically occupies smaller amount of space than strings; (2) it captures the prefix-sharing property of strings and hence is able to share computation; (3) it can be easily extended to support updates; and (4) it does not need final verification.

In our proposed system, we integrate techniques such as subtrie-pruning [4], improved edit distance computation [23], and multi-threading, in order to engineer an efficient and versatile systems for both string similarity search and join.

In the subsequent sections, we first describes the design of the system, followed by the description of pruning techniques and implementation details.

2. DETAILED DESCRIPTION

In this section, we present the detailed description of the system. Firstly, we give the framework of the system. Then, we demonstrate the detailed design and implementation to deal with string similarity search and join, respectively.

2.1 Framework

As mentioned above, the system is based on the trie data structure. Figure 1 shows an example trie constructed for four data strings: {art, cab, map, mate}. At first, all the strings in the dataset are indexed into a trie. Duplicate strings are indexed into the same path in the trie but with different string ids stored in the leaf node of the path. Each node in the trie contains a range of the length of strings stored underneath this node. For example, in Figure 1, a range [3, 4] is stored in node n_8 as the minimum length of the descendant leaf nodes of n_8 is 3 and the maximum length is 4. Then, the index trie is probed incrementally to compute active node set using Algorithm 1 and Algorithm 2, respectively, to answer string similarity search and join queries.

2.2 String Similarity Search

Algorithm 1 demonstrates the process for string similarity search.[1] In order to find similar strings for a given query string, we first start from the root node of the trie, incrementally compute the active node set of each prefix of the

---

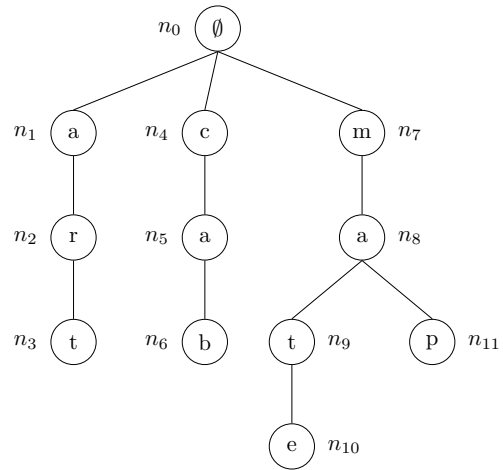[1]Detail of Function *computeActiveNode* can be found in [9].

Figure 1: Trie Index Example

query string [3]. More specifically, we compute the active node set of empty string first given the edit distance threshold, then for each prefix of length i of the query string (we call it Q_i, where $0 \leq i \leq |Q|$; latter being the length of the query string), we compute the active node set of Q_i according to the active node set of Q_{i-1}. After the computation of each active node set, several filtering methods such as length filtering are adopted to further reduce the size of active node set. These filtering methods are presented in the next section. Finally, after we get the active node set of the complete query string, we check add all the nodes in the active node set that is a leaf node to the final result set.

Algorithm 1: SimiSearch($\mathcal{I}, Q, \tau$)

> **Input** : $\mathcal{I}$: index trie which stores all the dataset strings
> Q : the query string
> τ : the given edit-distance threshold
> **Output**: $R : R = \{(s \in \mathcal{I}.strings, Q) | ed(s, Q) \leq \tau\}$

1 $r = \mathcal{I}.root$;
2 $\mathcal{A}_{Q_0} = \mathcal{A}_r = \{n | n \in \mathcal{I} \ \&\& \ |n| \leq \tau\}$;   /* get active node set of empty string */
3 **for** $i = 1$ *to* $|Q|$ **do**
4 $\quad$ Q_i = prefix of length i of Q;
5 $\quad$ $\mathcal{A}_{Q_i}$ = computeActiveNode(Q_i, pruning($\mathcal{A}_{Q_{i-1}}$));
6 **for each** *leaf node* $n_l \in \mathcal{A}_{Q_{|Q|}}$ **do**
7 $\quad$ $R = R \cup \{(n_l, Q)\}$;

2.3 String Similarity Join

A naive method to support join is to directly use the similarity search algorithm on each string in the data or query set. One immediate observation is that when searching data strings with similarity search algorithm, most data strings have shared prefixes. For example, string "abcdefg" and "abcdhijk" both share prefix "abcd". In this case, both strings will have to calculate the active node set of prefix "abcd" twice. Therefore, we can eliminate the redundant calculation through trie traversal.

For self-join, as all the data strings have been indexed into the trie, we can traverse the trie once to find all the

similar pairs as proposed in [4]. Algorithm 2 demonstrates similarity self-joins. Similar to its search counterpart, the trie is traversed from the root node to incrementally compute active node set of each trie node. In order to avoid repeated access of trie node, we traverse the trie in pre-order and store the active node set of traversed trie nodes into a stack, and pop the node out after all its descendant nodes are traversed. In this manner, we make sure each trie node is processed once and its active node set is computed once. During the traversal, if a leaf node is encountered, all the nodes in its active node set are checked. If anyone of the active nodes is a leaf node, it is added into result set with the current probing leaf node as a similar pair. An extra process is the filtering process. Same as the filtering process in search, after the calculation of each active node set, several filtering methods such as length filtering and single branch filtering are used to remove unnecessary active nodes so as to improve the time and space efficiencies.

For non-self-join, we just build a trie for each of the two datasets and traverse the one trie to search against the other trie.

Algorithm 2: SimiJoin($\mathcal{I}, n, \mathcal{A}, \tau$)

Input : $\mathcal{I}$: index trie which stores all the dataset strings
 n : current processing node.
 $\mathcal{A}$: current active node set.
 τ : the given edit-distance threshold
Output: $R : R = \{(s_i \in \mathcal{I}.strings, s_j \in \mathcal{I}.strings)|ed(s,Q) \leq \tau, i \neq j\}$

1 **if** $n = \mathcal{I}.root$ **then**
2 $\quad$ $\mathcal{A} = \mathcal{A}_r$; $\quad$ /* get active node set of empty string */
3 **if** n *is leaf node* **then**
4 $\quad$ **for each** *leaf node* $n_l \in \mathcal{A}$ **do**
5 $\quad\quad$ $R = R \cup \{(n_l, n)\}$;
6 **for each** *node* $n_i \in children(n)$ **do**
7 $\quad$ $\mathcal{A}_i = $ computeActiveNode(n_i, pruning($\mathcal{A}$));
8 $\quad$ $R = R \cup SimiJoin(\mathcal{I}, n_i, \mathcal{A}_i, \tau)$;

---

3. TECHNIQUES

In this section, we describe several techniques used in the system, including improved edit distance computation, length filtering used in similarity search and single branch filtering used for join.

3.1 Edit distance computation

The standard method to compute the edit distance between two strings D and Q (of length n and m, respectively) is the dynamic programming algorithm that fills in a matrix M of size $(n + 1) \cdot (m + 1)$. Each cell $M[i, j]$ records the edit distance between the length i and j prefixes of the input strings, respectively. The cell values can be computed in one pass in row-wise or column-wise order based on the following equation:

$$M[i,j] = \min(M[i-1,j-1] + \delta(D[i], Q[j]), \quad \text{(substitution)}$$
$$M[i-1,j] + 1, \quad \text{(insertion)}$$
$$M[i,j-1] + 1), \quad \text{(deletion)}$$
$$(1)$$

where $\delta(x, y) = 0$ if $x = y$, and 1 otherwise. The boundary conditions are $M[0, j] = j$ and $M[i, 0] = i$. The time complexity is $O(n \cdot m)$. In this paper, we use the convention of placing the query string vertically and the data string horizontally in the matrix, as shown in Figure 2(a).

Define a k-diagonal of the matrix as all the cells $M[i, j]$ such that $i - j = k$. To determine if D is within τ edit distance from Q, the *threshold edit distance algorithm* in [15] only needs to compute the k-diagonals of the matrix, where $k \in [-\tau, \tau]$, as shown in the green and yellow shaded area in Figure 2(a). The complexity is $O(\tau \cdot \min(n, m))$.

We can see that when the edit-distance value in the k-diagonal of the matrix is greater than the given threshold, its actual value is meaningless to us. Therefore, we can use a special character to replace all such values in the k-diagonal, for example Figure 2(b), which is transformed by replacing all the values in the k-diagonal in Figure 2(a) with "#". It is obvious that the computation of edit-distance can be stopped after values in the third row is computed (when length of query string = 2) as all the values in this row is greater than the threshold, thus this two strings cannot be similar strings. The method of incrementally computing active node set of each prefix of the query string is similar to compute edit distance values in k-diagonal of the matrix. Usually in the k-diagonal, the complexity of computing values in the $(j + 1)^{th}$ column/row from values in j^{th} column/row is $O(\tau)$. However, since we can precompute all the possible edit-distance transition state using the intuition in [11], the transition cost can be improved to $O(1)$ which enhances computation performance substantially.
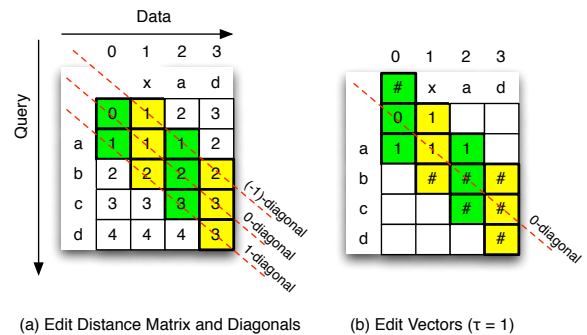

(a) Edit Distance Matrix and Diagonals $\qquad$ (b) Edit Vectors (τ = 1)

Figure 2: Edit Vector ($\tau = 1$)

3.2 Filtering methods

3.2.1 Length Filtering

Length filtering is first proposed by Gravano et. al. in [5] to deal with string similarity queries, but it is first used in trie-based method in [4]. The basic idea is that after computing one active node set, check all the active nodes in the set to see if there is any node whose descendant leaf nodes are all have length difference lager than τ with the query string; if so, these nodes should be removed from the active node set as they do not contain any leaf node which is within edit-distance τ with the query string.

3.2.2 Extended Length Filtering

There is another extended way to use the length filtering. When checking the minimum length difference between query string and all the strings stored underneath a particular active node, instead of using the original edit distance threshold τ, we can get a tighter bound by checking the current edit distance between current active node with the query string. More precisely, if the current edit distance between query string and the string represented by the current probing active node is τ_1, where τ_1 must be no larger than τ, then the new threshold should be changed to $\tau - \tau_1$, which means if all the strings stored underneath the current probing active node have length difference larger than $\tau - \tau_1$ with the query string, then this active node can be deleted from the active node set.

The above two filtering methods can be used in both approximate string search and join problem. By adopting these two filtering methods, the size of the active node set in each step can be reduced, and therefore, improves the search and join performance.

3.2.3 Single branch filtering

The single branch filtering method is proposed in [4]. The intuition of this method is that if two nodes in an active node set belong to the same trie branch (i.e. they have ancestor-descendant relationship), and they have exactly the same set of descendant leaf nodes, then one of them can be pruned as the including of this node will not introduce any new results (usually the one who is ancestor is removed).

3.3 Implementation details

3.3.1 Trie index

In the framework, we employ trie as the index structure. As we require the trie structure to efficiently support both traversal and random accesses, we use the STL map as the data structure to implement the trie node. With STL map in each trie node, the traversal of all children of one node requires $O(n)$ time while search one particular child node will only require $O(\log(n))$ time (n is the number of children).

3.3.2 Multi-thread Optimized Search and Joins

Most current hardware systems are equipped with multi-core CPUs. In the mean time, almost all the latest operating systems support multi-threaded programming. Although current techniques for String Similarity Search and Join are very efficient, it is clear that multi-thread based implementation has the potential to further improve the performance. As there is little published studies in this direction, we use the following simple strategies to implement the multi-threaded version of our system.

1. We support multi-threaded similarity search by a simple extension. We first construct a lock-protected query pool which contains all the unfinished queries. Then we setup several search threads and each search thread will independently acquire a query job and process its it and then output its results.

2. We use the following simple method to support multi-threaded similarity join. We first sort the data by alphabetic order. Then based on the data distribution, we partition the data into t (i.e., the number of threads) partitions logically. Within each partition, we assign each partition to one worker thread.

4. REFERENCES

[1] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, 2007.

[2] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, 2006.

[3] S. Chaudhuri and R. Kaushik. Extending autocompletion to tolerate errors. In *SIGMOD Conference*, 2009.

[4] J. Feng, J. Wang, and G. Li. Trie-join: a trie-based method for efficient string similarity joins. *VLDB J.*, 21(4):437–461, 2012.

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

[6] M. Hadjieleftheriou and C. Li. Efficient approximate search on string collections. *PVLDB*, 2(2):1660–1661, 2009.

[7] M. Hadjieleftheriou and D. Srivastava. Approximate string processing. *Foundations and Trends in Databases*, 2(4):267–402, 2011.

[8] T. Kahveci and A. K. Singh. Efficient index structures for string databases. In *VLDB*, pages 351–360, 2001.

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

[10] X. Lin and W. Wang. Set and string similarity queries: A survey. *Chinese Journal of Computers*, (10):1853–1862, 2011.

[11] W. J. Masek and M. Paterson. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.*, 20(1):18–31, 1980.

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

[13] D. Sokol, G. Benson, and J. Tojeira. Tandem repeats over the edit distance. *Bioinformatics*, 23(2):30–35, 2007.

[14] B. S. T. Bocek, E. Hunt. Fast Similarity Search in Large Dictionaries. Technical Report ifi-2007.02, Department of Informatics, University of Zurich, April 2007. http://fastss.csg.uzh.ch/.

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

[16] J. Wang, J. Feng, and G. Li. Trie-join: Efficient trie-based string similarity joins with edit. In *VLDB*, 2010.

[17] W. Wang. Similarity joins as stronger metric operations. *SIGSPATIAL Special*, 2:24–27, July 2010.

[18] W. Wang, J. Qin, C. Xiao, X. Lin, and H. T. Shen. VChunkJoin: An efficient algorithm for edit similarity joins. *IEEE Trans. Knowl. Data Eng.*, 2012.

[19] W. Wang, C. Xiao, X. Lin, and C. Zhang. Efficient approximate entity extraction with edit constraints. In *SIMGOD*, 2009.

[20] C. Xiao, J. Qin, W. Wang, Y. Ishikawa, K. Tsuda, and K. Sadakane. Efficient error-tolerant query autocompletion. *Proceedings of the VLDB Endowment*, 2013.

[21] C. Xiao, W. Wang, and X. Lin. Ed-Join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB*, 1(1):933–944, 2008.

[22] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW*, 2008.

[23] X. Zhou, J. Qin, C. Xiao, W. Wang, X. Lin, and Y. Ishikawa. LEVA: An efficient query processing algorithm for error tolerant autocompletion. Submitted for publication.