mathematics

Article

A Trie Based Set Similarity Query Algorithm

Lianyin Jia [1,2], Junzhuo Tang [1], Mengjuan Li [3,*], Runxin Li [1], Jiaman Ding [1] and Yinong Chen [4]

[1] Faculty of Information Engineering and Automation, Kunming University of Science and Technology, Kunming 650500, China; lianyinjia@kust.edu.cn (L.J.); 20202204250@stu.kust.edu.cn (J.T.); rxli@kust.edu.cn (R.L.); jiamanding@kust.edu.cn (J.D.)
[2] Yunnan Key Laboratory of Artificial Intelligence, Kunming University of Science and Technology, Kunming 650500, China
[3] Library, Yunnan Normal University, Kunming 650500, China
[4] School of Computing and Augmented Intelligence, Arizona State University, Tempe, AZ 85287, USA; yinong@asu.edu
* Correspondence: lmjlykm@163.com; Tel.: +86-13518797292

Abstract: Set similarity query is a primitive for many applications, such as data integration, data cleaning, and gene sequence alignment. Most of the existing algorithms are inverted index based, they usually filter unqualified sets one by one and do not have sufficient support for duplicated sets, thus leading to low efficiency. To solve this problem, this paper designs T-starTrie, an efficient trie based index for set similarity query, which can naturally group sets with the same prefix into one node, and can filter all sets corresponding to the node at a time, thereby significantly improving the candidates generation efficiency. In this paper, we find that the set similarity query problem can be transformed into matching nodes of the first-layer (FMNodes) detecting problem on T-starTrie. Therefore, an efficient FLMNode detection algorithm is designed. Based on this, an efficient set similarity query algorithm, TT-SSQ, is implemented by developing a variety of filtering techniques. Experimental results show that TT-SSQ can be up to 3.10x faster than existing algorithms.

Keywords: set similarity query; T-starTrie; FMNodes; TT-SSQ

MSC: 68W32

check for updates

Citation: Jia, L.; Tang, J.; Li, M.; Li, R.; Ding, J.; Chen, Y. A Trie Based Set Similarity Query Algorithm. *Mathematics* **2023**, *11*, 229. https://doi.org/10.3390/math11010229

Academic Editor: Jakub Nalepa

Received: 16 December 2022
Revised: 28 December 2022
Accepted: 29 December 2022
Published: 2 January 2023

1. Introduction

There is a long stream of research on set similarity query (SSQ). Given a collection of sets and a query, SSQ aims to find all sets in a dataset that are similar to the given query. SSQ plays an important role in many applications. For example, SSQ can help identify multiple profiles of the same customer in data integration. In data cleaning, similar objects can be found using SSQ to eliminate inconsistencies and redundancies [1]. In bioinformatics, SSQ can be used to find similar genome sequences to help find treatments for diseases.

Comparing sets one by one is inefficient and index is key to efficient SSQ. Most existing SSQ and set similarity join (SSJ) algorithms are inverted index based and adopt a filter -validation framework [2–6]. In the filtering stage, a variety of filtering techniques, such as prefix filtering, length filtering, and position filtering, are used to generate candidates. In the validation phase, the candidates are validated to obtain the final results. However, these algorithms usually filter unqualified sets one by one and do not have sufficient support for duplicated sets, thus leading to a low efficiency.

Trie can effectively compress the prefixes between sets into a common path, so it can better support duplicated or a near duplicated set. ETI, a trie based index, is implemented in [7], which performs better than other inverted index based competitors. It is mainly designed for T-overlap query, which aims to find all sets having at least T common elements with a given query. Although it can be extended to support Jaccard and many other similarity predicates, the efficiency is not high due to a large index and insufficient filtering in ETI.

SSQ and SSJ are significantly different in the execution process, index construction, threshold support, and many other aspects [8,9]. While a lot of work is focused on SSJ [10–12], there are few research works on SSQ. This paper is a good supplement to the existing studies.

As the algorithms in ETI, our algorithm is also trie based. However, considering that a query with low threshold is of little significance in real applications, we design T-starTrie, a trie index created on the *t-prefix*s (the prefixes computed by threshold t^*) of all sets. As T-starTrie only indexes the prefix of each set, its space overhead is much smaller than ETI and prefix filtering can be effectively utilized. In addition, it can support any queries with query threshold t no less than t^*. We found that SSQ on T-starTrie can be transformed into matching nodes of the first layer (FLMNodes) detecting problem on T-starTrie. Therefore, an efficient FLMNode detection algorithm is designed. Based on T-starTrie, TT-SSQ, an efficient SSQ algorithm, is proposed by developing a variety of filtering techniques. Experimental results show that TT-SSQ significantly outperforms the existing algorithms.

The main contributions of this paper are as follows:

1. We designed T-starTrie, a trie based index created on *t-prefix*s, which can effectively support any queries with a threshold no less than t^*.
2. We transformed SSQ into FMNodes finding problem, and then proposed an efficient FMNode detection algorithm.
3. We design efficient node depth filtering (NDFiltering) and NIL length filtering (NLFiltering) and then the proposed efficient SSQ algorithm, TT-SSQ. Extensive experimental results show that TT-SSQ outperforms the existing algorithms significantly.

The rest of this paper is organized as follows: Section 2 gives a problem definition, data preprocessing and related works. Section 3 presents T-starTrie. The TT-SSQ algorithm is given in Section 4. Section 5 discusses the experiments and analyses. Finally, we conclude the full text.

2. Preliminary

2.1. Problem Definition

Given a universe U consisting of a collection of elements, which are comparable to each other and with the same type, a database D contains a collection of sets with each set S consisting of a number of elements in U. $|S|$ is denoted as the length of set S. Each set S has a unique set identifier (SID), and we refer to an SID as a set when there is no ambiguity. We assume S is not a multi-set, that is, there are no duplicated elements in S. Based on the above discussions, the definition of SSQ is given as follows:

Definition 1. *SSQ: Given a dataset D, a query Q and a threshold t, a SSQ returns all $S \in D$ that satisfies $sim(S, Q) \geq t$.*

The *sim* in Definition 1 represents a set similarity predicate. There are a variety of similarity predicates available, such as Jaccard, Dice, Cosine, T-overlap, etc. This paper uses Jaccard as the default similarity predicate. Given two sets S and Q, the Jaccard similarity between them is defined as $\text{Jaccard}(S, Q) = \frac{|S \cap Q|}{|S \cup Q|}$. The value range of Jaccard is $[0, 1]$.

Example 1. *Table 1 shows an example dataset. Given a query $Q = \{2, 5, 6, 7\}$ and a threshold $t = 0.6$, the SSQ returns $\{S_3, S_4\}$, while the Jaccard similarity between Q and all the other sets are smaller than t.*

Table 1. An example dataset.

SID	Set
S_1	{2, 3}
S_2	{1, 2, 3}
S_3	{5, 6, 7}
S_4	{2, 5, 6, 9}
S_5	{1, 3, 7, 8}
S_6	{1, 3, 7, 8}
S_7	{2, 5, 7, 8, 9}
S_8	{2, 3, 4, 5, 7, 8, 9}
S_9	{1, 3, 4, 5, 6, 7, 8, 9}

2.2. Data Preprocessing

Existing SSQ algorithms usually need some specific data preprocessing, mainly including sorting sets or elements in specific orders. Similar to the most previous algorithms, this paper adopts length ascending ordering as the default set ordering, as it can effectively exploit length filtering to generate candidates. For the element ordering in each set, there are three common representatives: the value ordering, the frequency ordering and the frequency-value ordering [8]. In this paper, we use the frequency-value ordering as our default element ordering, and it can be further divided into two kinds: the frequency value ascending ordering (FVA) and the frequency value descending ordering (FVD). Take FVA as an example, and the steps to sort D in FVA are as follows:

1. sorts the elements of U in frequency ascending order in D;
2. for each element e in each set S of D, replace e with its subscript in U;
3. sort the elements in each set S by their value in ascending order.

Example 2. *Given the example dataset in Table 1, the frequencies of elements from 1 to 9 are 4,5,6,2,5,3,6,5,4, respectively. After sorting these elements in frequency ascending order, we obtain the following mappings between each element in U and its subscript using FVA :* $\{4 \rightarrow 1, 6 \rightarrow 2, 1 \rightarrow 3, 9 \rightarrow 4, 2 \rightarrow 5, 5 \rightarrow 6, 8 \rightarrow 7, 3 \rightarrow 8, 7 \rightarrow 9\}$. *Following these mappings, the mapped FVA dataset is given in Table 2.*

Table 2. The mapped FVA dataset.

SID	Set
S_1	{5, 8}
S_2	{3, 5, 8}
S_3	{2, 6, 9}
S_4	{2, 4, 5, 6}
S_5	{3, 7, 8, 9}
S_6	{3, 7, 8, 9}
S_7	{4, 5, 6, 7, 9}
S_8	{1, 4, 5, 6, 7, 8, 9}
S_9	{1, 2, 3, 4, 6, 7, 8, 9}

2.3. Related Works

SSQ and SSJ are the two common set similarity processing methods. Most previous work has focused on SSJ. Given two datasets, SSJ identifies all similar set pairs, in which the first set comes from the first dataset and the second set comes from the other dataset [13]. AllPairs [3] preprocessed the datasets in length ordering and then combined prefix filtering and length filtering together to generate candidates. PPJoin [4] , viewed as a combination of prefix filtering and positional filtering, further imposed the frequency ascending ordering and position filtering to improve the efficiency of AllPairs. Considering the limited filtering ability of fixed prefixes, an adapted method was proposed in [5], which improves the filtering ability by increasing the prefix length and well supports SSQ and SSJ. Groupjoin proposed in [14] divided sets with the same length and prefix into one group, which can filter sets in a group simultaneously. Deng et al. [6] proposed a partition-based framework to enhance filtering capabilities. For more information, we refer reader the recent surveys [8,15] on SSQ and SSJ.

Currently, there are only a few works supporting SSQ, e.g., ETI [7] and Adapt [5]. In spite of this, it was also observed that the existing SSJ algorithms can also be considered as SSQ algorithms when the offline indexes are built in advance and RS joins are performed, so they can be compared with our algorithm as well.

In addition to the above works, there are some other studies on SSQ in specific scenarios. MinHash based algorithms were studied in [16] to obtain the approximate results of SSQ, which is different to our work aiming to obtain exact results. McCauley et al. [17] studied SSQ under skew distributed data. Zhang et al. [18] studied the knn SSQ algorithm, which returns the first k results with the maximum Jaccard similarity to the query. Zhang et al. [19] proposed a learning based method to divide a dataset into groups to improve the efficiency of SSQ. Vernica et al. [20] researched parallel SSJ algorithms on Mapreduce.

3. T-starTrie

3.1. Inverted Index

As an index extensively used in SSQ and SSJ, the inverted index is composed of two parts: a directory and inverted lists. The directory consists of all the elements in U. Each element corresponds to an inverted list containing all SIDs whose set contains that element. The inverted index created for the dataset in Table 1 is shown in Figure 1.

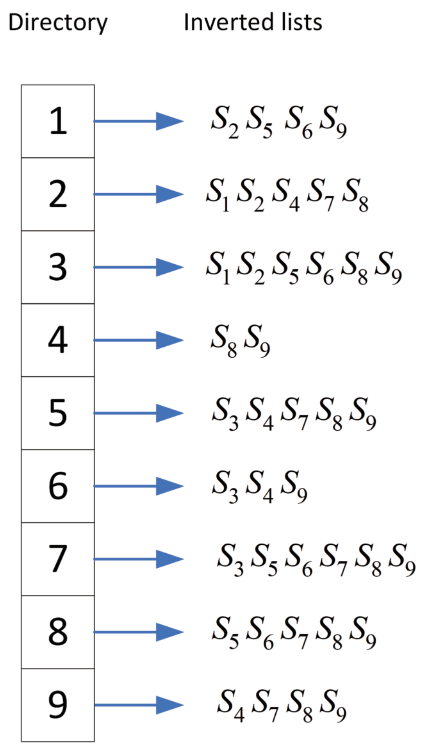

Directory
Inverted lists
1 → $S_2 S_5 S_6 S_9$
2 → $S_1 S_2 S_4 S_7 S_8$
3 → $S_1 S_2 S_5 S_6 S_8 S_9$
4 → $S_8 S_9$
5 → $S_3 S_4 S_7 S_8 S_9$
6 → $S_3 S_4 S_9$
7 → $S_3 S_5 S_6 S_7 S_8 S_9$
8 → $S_5 S_6 S_7 S_8 S_9$
9 → $S_4 S_7 S_8 S_9$

Figure 1. The inverted index.

As shown in Figure 1, the SID of each set is inserted into the inverted lists corresponding to all elements in that set, resulting in redundant storage and query time overhead.

For example, S_5 and S_6 are exactly the same. However, they are inserted into all the inverted lists of elements 1, 3, 7 and 8 and need to be processed dependently in these four inverted lists, deteriorating the performance.

3.2. T-starTrie

In this paper, we use trie to tackle this issue. Trie, also known as prefix tree, is an ordered tree structure. The common prefixes between two sets are mapped to a common path in trie. Each node in trie represents the collection of sets whose prefixes can map to this node. For these sets, we can process them in a group manner. Therefore, trie can support duplicated or nearly duplicated sets better than inverted indexing. The sample trie created for the dataset in Table 1 is shown in Figure 2.

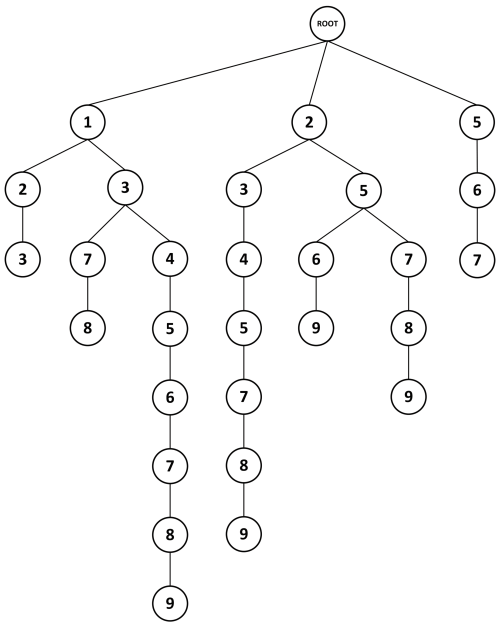
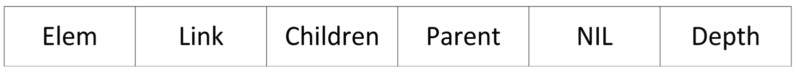


Figure 2. The sample trie created for the dataset in Table 1.

Considering that the simple trie in Figure 2 is difficult to support SSQ, we expand the trie node by adding more properties. The structure of the expanded node is shown in Figure 3.



Elem	Link	Children	Parent	NIL	Depth

Figure 3. The expanded node structure.

In Figure 3, for a Node N, the corresponding descriptions of each attribute are as follows:

1. N.Elem: the element corresponding to N;
2. N.Link: the pointer pointing to the next node with element N.Elem;
3. N.Children: the pointer pointing to the first child of N;
4. N.Parent: the pointer pointing to to the parent node of N;
5. N.NIL: the node inverted list containing all the SIDs whose prefix maps to N;
6. N.Depth: the depth of N in trie. The depth of the root is 0.

For a large dataset, creating a complete trie will lead to a large space overhead as shown in Figure 2. Considering that a very small similarity threshold has little significance in the real world scenarios, to reduce the space overhead, in this paper, we construct a novel trie based index: T-StarTrie.

T-StarTrie is made up of two parts: a directory and a trie. The directory is the same as that of an inverted index. Each entry in the directory contains an element e and a link pointing to the first node N with N.Elem $=e$, hereafter we called the link for element e as e.Link for ease of description. The trie in T-StarTrie is created for a specific threshold t^* (e.g., 0.6). That is, only the first $|S| - \lceil t^* \times |S| \rceil + 1$ elements of each set S are indexed in T-StarTrie. For convenience, we denote the prefix of S corresponding to threshold t as *S.t-prefix*.

As T-starTrie indexes t^**-prefix* of each set, if *Q.t-prefix* and *S.*t^**-prefix* have no common elements, Q and S cannot be similar. Thus, T-Startrie can support any query with threshold greater than t^*. Given the above description, the T-starTrie built on the dataset in Table 1 is shown in Figure 4. For ease of description, a node ID (colored in red) is virtually added to the right of each node, and we call the node with ID i as node i hereafter.

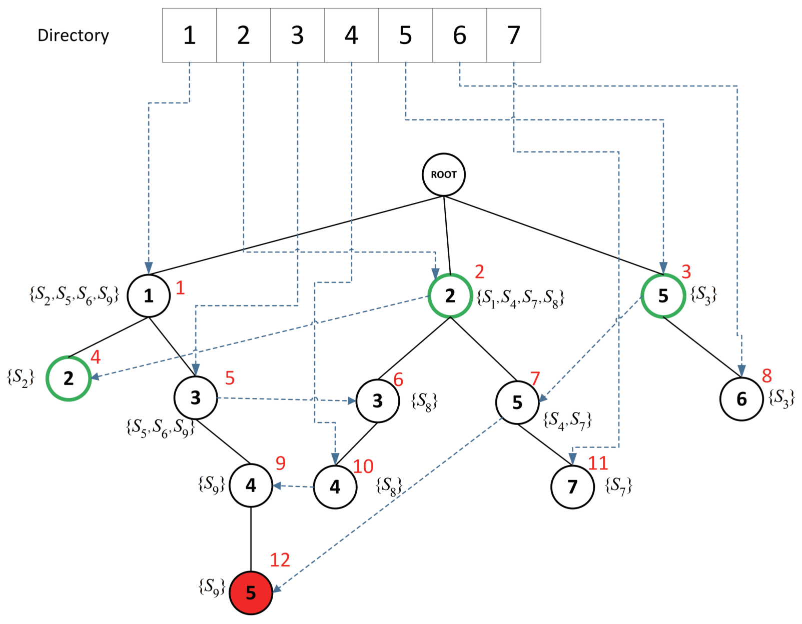


Figure 4. T-starTrie.

Example 3. *In Figure 4, for* $S_2 = \{1,2,3\}$ *and t = 0.6,* $S_2.t^*$*-prefix is {1,2}, so only elements 1 and 2 in* S_2 *are indexed in T-StarTrie, corresponding to node 1 and node 4 in T-StarTrie, respectively. The NIL of node 1 is {*S_2, S_5, S_6, S_9*} as these four sets have the same prefix {1}. In addition, node 4 has the same Elem with node 2, so these two nodes are linked through the Link pointer. According to the above description, the corresponding index construction algorithm is shown in Algorithm 1.*

Updating T-starTrie is relatively straightforward. However, it is not easy to update T-starTrie while keeping FVA, as inserting new sets into T-starTrie may change the underlying frequency distribution. Fortunately, we do not need to keep FVA all the time, and a T-starTrie with approximate FVA has little impact on the query performance. Thus, we can regulate FVA periodically when the server is not busy.

Algorithm 1 CreateT-starTrie

```
Input: D, a database with each set sorted according to a certain element ordering
       t*, a threshold to create trie
Output: R, the root of the created T-starTrie
1: $R \leftarrow \emptyset$
2: create a directory using all distinct elements in the $t^*$-prefix for all sets in $D$
3: for each set $S \in D$ do
4:     P ← R
5:     for each element $e$ in the $t^*$-prefix of $S$ do
6:         find the node $N$ whose Elem is $e$ in $P$.Children
7:         if $N$ is null then
8:             $N$ ←create a new node with Elem $e$
9:             $N$.NIL ← {$S$.SID}
10:            insert $N$ into $P$.Children
11:            insert $N$ into $e$.link
12:        else
13:            $N$.NIL ← $N$.NIL ∪ {$S$.SID}
14:        end if
15:    end for
16:    P ← N
17: end for
```

4. Similarity Query Algorithm

4.1. Filtering Techniques

Filtering techniques are key to efficient SSQ algorithms. In this paper, we develop two following efficient filtering techniques for TT-SSQ algorithm: node depth filtering (NDFiltering) and NIL length filtering (NLFiltering). Firstly, we have NDFiltering lemma as shown in Lemma 1.

Lemma 1. *Given a query Q, a threshold t and a node N, if N.Depth* $> \lfloor |Q|/t \rfloor - \lceil \lfloor |Q|/t \rfloor \times t \rceil + 1$, *then all sets in N.NIL cannot be similar to Q.*

Proof. According to the length filtering, the maximum length of a set S that meets Jaccard similarity with Q is $\lfloor |Q|/t \rfloor$, and the maximum length of $S.t^*$*-prefix* is $\lfloor |Q|/t \rfloor - \lceil \lfloor |Q|/t \rfloor \times t \rceil + 1$. If N.Depth> $S.t^*$*-prefix* , the length of all sets in N.NIL must be greater than $\lfloor |Q|/t \rfloor$. Therefore, all sets in N.NIL cannot be similar to Q. Thus, Lemma 1 holds.

According to NDFiltering, any nodes with depth greater than $\lfloor |Q|/t \rfloor - \lceil \lfloor |Q|/t \rfloor \times t \rceil + 1$ can be safely pruned. For the nodes passing the NDFiltering, NLFiltering can be further applied. That is, given a query Q, a threshold t and a node N, if a set $S \in N$.NIL is similar with Q, the length of S must be within $[|Q| * t, |Q|/t]$.

NLFiltering is the application of length filtering on NILs. As we keep all sets in length ascending order, the SIDs in a NIL also keep the same order. As a result, we can locate the qualified SID range in an NIL by binary search, thus filtering sets whose length cannot meet the requirements. □

Example 4. *In Figure 4, given a query* $Q = \{2, 5, 6, 7\}$ *and t = 0.6, the Q.t-prefix is {2, 5}. When querying in T-starTrie, although the ELEM of node 12 (the red node in Figure 4) is 5, this node can be filtered as its depth is* $4 > \lfloor |4|/0.6 \rfloor - \lceil \lfloor |4|/0.6 \rfloor \times 0.6 \rceil + 1 = 3$. *For another example, the depth of node 2 is 1, which satisfies NDFiltering. However, according to NLFiltering, the length bound of for Q is [3, 6], so* S_1 *and* S_8 *in the NIL of this node can be filtered out.*

Based on NDFiltering and NLFiltering, we present our TT-SSQ algorithm.

4.2. TT-SSQ

Before introducing TT-SQL, the following definitions are given:

Definition 2. *Matching Node (MNode): a node N is called a MNode if N.Elem appears in Q.t-prefix .*

Definition 3. *Matching node of the first layer(FMNode): an MNode with no other MNodes in the path from this node to the root.*

Based on the definition of FMNode, we have Lemma 2 as follows:

Lemma 2. *The union of the NILs of all FMNodes is the superset of SSQ.*

Proof. Firstly, we know from prefix filtering that if two sets are similar, they must have a common element in their prefixes. Thus, the union of the NILs of all MNodes contains all sets similar to Q. Secondly, the NIL of any non-FMNode is a subset of the NIL of FMNodes, so Lemma 2 holds.

According to Lemma 2, our problem can be transformed into obtaining FLMNodes on T-starTrie. We observed that a node N should meet one of the following three conditions to be a FMNode:

1. N.Elem equals the first element in Q;
2. N.Depth is 1;
3. N does not satisfy the 1st and the 2nd condition, but satisfies Definition 3.

□

Identifying an FMNode N satisfying the 1st and the 2nd condition is rather easy. For the 3rd condition, we can iteratively check each node N^* from the parent of N to the root in a bottom-up manner to see whether N^*.Elem is in *Q.t-prefix*. In most cases, we do not need to check all nodes from N to the root; instead, we can make an early termination when N^*.Elem is in *Q.t-prefix* or N^*.Elem overpasses the first element of Q, here "overpasses" means "is smaller than" for FVA, and "is greater than" for FVD. The algorithm for checking whether N is a FMNode, IsFMNode, is shown in Algorithm 2.

Algorithm 2 IsFMNode

```
Input: $Q$={$e_1$, $e_2$, ... , $e_m$}, a query
       $N$, a node
Output: true if $N$ is a FMNode, else otherwise
1: if $N$.Elem=$e_1$ then
2:     return true
3: end if
4: if $N$.Depth=1 then
5:     return true
6: end if
7: for each node $N^*$ from the parent of $N$ to the root do
8:     if $N^*$.Elem in $Q.t\text{-}prefix$ then
9:         return false
10:    else $N^*$.Elem overpasses $e_1$
11:        return false
12:    end if
13: end for
14: return true
```

Based on the above discussion, the main steps of TT-SSQ are described as follows:

1. For each node N in the Link of each element in *Q.t-prefix*;
2. If N does not pass NDFiltering, skip this node;
3. If N is a FMNode, perform NLFiltering on its NIL to obtain the candidates;

The complete TT-SSQ algorithm is shown in Algorithm 3:

Algorithm 3 TT-SSQ

```
Input: $Q$={$e_1$,$e_2$,...,$e_m$}, a query
       $t$, a query threshold
Output: $R$, all sets having a similarity at least $t$ with $Q$
1:  $R \leftarrow \emptyset$
2:  for each element $e$ in $Q.t$-prefix do
3:      for each node $N$ in $e$.Link do
4:          if $N$.Depth $\leq \lfloor |Q|/t \rfloor - \lceil \lfloor |Q|/t \rfloor \times t \rceil + 1$ then
5:              if IsFMNode(Q,N) then
6:                  C $\leftarrow$ the SIDs in $N$.NIL whose set length between $[|Q|t, |Q|/t]$
7:                  for each c in C do
8:                      if verify($Q$, $c$, $t$) then
9:                          R $\leftarrow$ R $\cup$ {c}
10:                     end if
11:                 end for
12:             end if
13:         end if
14:     end for
15: end for
```

Example 5. *Given a query* $Q = \{2, 5, 6, 9\}$ *and t=0.6, the Q.t-prefix is* $\{2, 5\}$. *Since 2 is the first element in* Q.t-prefix, *the nodes with green bold circle in Figure 4 (node 2 and node 4) are FMNodes. For nodes with Elem 5, only node 3 is a FMNode that passed NDFiltering. Therefore, nodes 2, 3 and 4 are FMNodes and are further filtered by NLFiltering to obtain the final candidates* {S_1,S_2,S_3,S_4,S_7,S_8} .

5. General Setup

5.1. Experiment Setup

All experiments were run on a Ubuntu machine with Intel(R) Core(TM) i7-7700 CPU @ 3.60 GHz with 32 GB memory and GCC is used as the default compiler.

5.2. Datasets

We carry out experiments on two real datasets. The first one is MSWEB [21] in the UCI KDD archive. Each set in MSWEB represents the virtual areas of the Microsoft portal visited in one session. MSWEB contains 32,711 sets with 294 distinct elements. The minimum, maximum, and average lengths recorded are 1, 35, and 3.01, respectively.

The second dataset we used is KOSARAK [22]. The dataset contains anonymous click-stream data of a Hungarian online news portal. KOSARAK contains 990k records with 41,269 different elements. The minimum, maximum, and average lengths recorded are 1, 2498, and 8.10, respectively.

The detailed statistics on the two datasets are shown in Table 3.

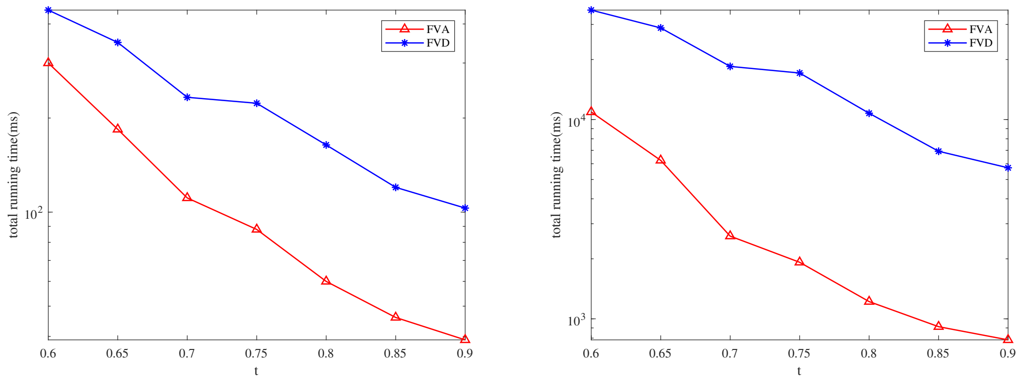
Table 3. The statistics of the two datasets.

Dateset	Cardinality	Avglen	Element	MaxLen	MinLen
KOSARAK	990,001	8.10	41269	2498	1
MSWEB	32,711	3.01	294	35	1

We randomly select 10,000 sets from each dataset as queries, and report the total running time for each algorithm mentioned below. Note that all results in the following figures are with an exponential scale. Unless otherwise stated, we assume $t = t^*$, that is, we use the creation threshold as the default query threshold.

5.3. The Effects of Different Orderings

To investigate the effects of different orderings on query performance, TT-SSQ was executed on FVA and FVD ordered datasets, respectively. The corresponding results on MSWEB and KOSARAK are shown in Figure 5a,b, respectively.



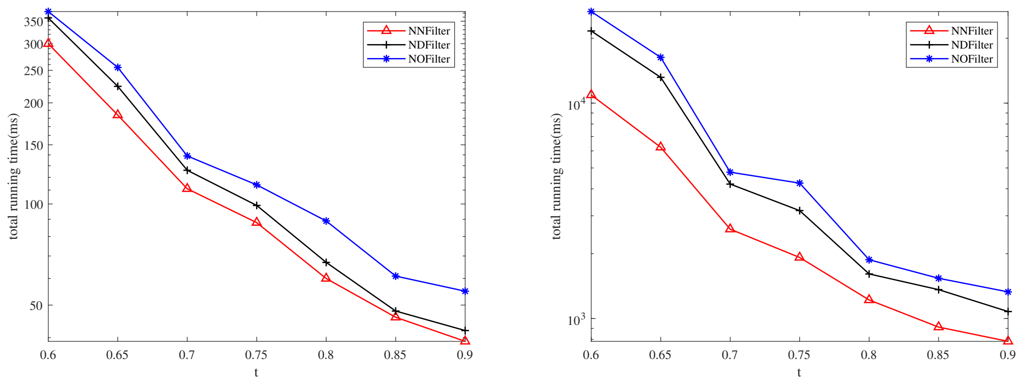
(**a**) MSWEB (**b**) KOSARAK

Figure 5. The results for different orderings.

As can be seen from Figure 5, the query efficiency of FVA is significantly higher than that of FVD on both datasets. Although more common prefixes can be compressed and more compact T-starTries can be obtained by using FVD, the number of candidates can be significantly reduced as each element corresponds to much fewer MNodes in T-starTrie for FVA. Taking KOSARAK as an example, when t = 0.7, the number of candidates corresponding to FVA and FVD are 46,774,970 and 984,889,127, respectively. Clearly, the much lesser candidates for FVA outweigh the more compact T-starTrie for FVD. In the following description, FVA is used as default dataset preprocessing method.

5.4. The Effects of Different Filtering Policies

To investigate the query performance of TT-SSQ under different filtering strategies, three filtering strategies were designed, namely no filtering (NOFiltering), NDFiltering only, and NDFiltering + NLFiltering(NNFiltering). The corresponding query results on MSWEB and KOSARAK are shown in Figure 6a,b, respectively.



(**a**) MSWEB (**b**) KOSARAK

Figure 6. The results for different filtering policies.

As can be seen from Figure 6, on both datasets, the effectiveness relationships among the three filtering strategies are: NOFiltering<NDFiltering<NNFiltering. On KOSARAK

when $t = 0.7$, the number of candidates under these three strategies are 57,425,680, 57,265,130 and 46,774,970, respectively. Therefore, combining NDFiltering and NLFiltering together can effectively reduce the number of candidate sets and improve the query efficiency. Unless otherwise specified, NNFiltering is used as the default filtering strategy in the following description.

5.5. The Support of Different Query Thresholds

Although T-starTrie created for threshold t^* can effectively support queries with any threshold $t \geq t^*$, for a T-starTrie created with a smaller t^*, there will inevitably introduce excessive overheads when query threshold t is large due to the additional index accesses. Therefore, we set $t^* = 0.6, 0.7, 0.8$ and t, respectively, to investigate the effects of query time over t^*. Here, $t^* = t$ represents the idea state that means we always execute queries with the creation threshold t^*. The results for supporting different query thresholds are shown in Figure 7a,b, respectively.

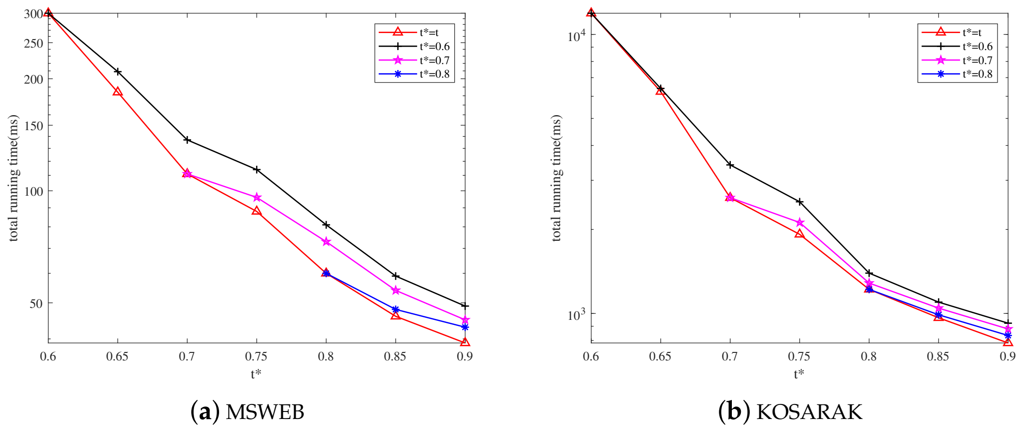


Figure 7. The results for supporting different query thresholds.

As it can be seen from Figure 7, the running times gradually decrease and reach the idea state ($t^* = t$) with the increase of t^*. When t is 0.9, the total running times on KOSARAK are 923, 880, 834, and 784 ms for $t^* = 0.6$, 0.7, 0.8, and 0.9, respectively. For $t^* = 0.6$, 0.7, and 0.8, the query times increased are 17.8%, 12.2%, and 6.4%, respectively, compared with the idea state $t^* = 0.9$. These increases are relatively small, indicating that T-starTrie created with t^* can effectively support query threshold $t \geq t^*$.

5.6. Compared with the State-of-the-Art Algorithms

To illustrate the efficiency of TT-SSQ, we compare it with five state-of-the-art algorithms: Allpairs [3], PPjoin [4], Adaptjoin [5], ETI [7], and Groupjoin [14]. Allpairs, PPjoin, and Groupjoin are originally SSJ algorithms, and we change them to SSQ algorithms according to the description in Section 2.3. For each algorithm, we add a suffix "-SSQ" to indicate that it is a SSQ algorithm. The comparisons among different algorithms are shown in Figure 8a,b, respectively.

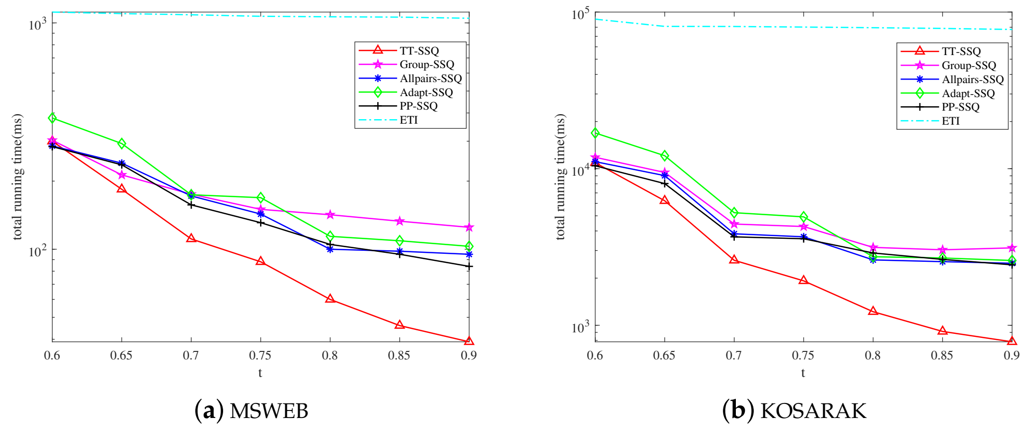


Figure 8. The comparisons among different algorithms.

As can be seen from Figure 8, the total running times of all algorithms decrease significantly with the increase of t. Relatively speaking, TT-SSQ is much more efficient than the other algorithms except for t = 0.6. The advantages of TT-SSQ gradually increase as t increases, indicating that TT-SSQ is especially suitable for higher thresholds. When t = 0.9, TT-SSQ only needs 784 ms to execute 10,000 queries on KOSARAK, while PP-SSQ, the fastest among all other algorithms, needs 2432 ms. The operation efficiency of TT-SSQ is 3.10x faster compared with PP-SSQ, the main reason lies in the fact that T-starTrie can compress the common prefix of different sets into a single path, so we can filter nodes in a group manner, avoiding the deficiency of filtering sets one by one.

6. Conclusions

In this paper, T-starTrie, a novel trie-based index is designed, and TT-SSQ is proposed based on T-starTrie. T-starTrie is time and space efficient by only indexing the t^**-prefix* of each set. We transform the SSQ problem into the problem of finding FMNodes in T-starTrie and propose two effective filtering methods: NDFiltering and NLFiltering. By exploiting NDFiltering and NLFiltering, TT-SSQ can filter a large number of sets in a group manner, which has a high efficiency. Experimental results show that TT-SSQ significantly outperforms the other algorithms, especially for high query thresholds. Next, we will further optimize our index and algorithm. In addition, applying this work to some real world applications is also an interesting direction.

Author Contributions: Conceptualization, L.J. and M.L.; methodology, L.J. and J.T.; software, L.J. and J.T.; validation, R.L. and M.L.; writing, L.J. and J.T.; resources, Y.C., J.D. and R.L.; review, J.D., Y.C. and M.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Natural Science Foundation of China under Grant Nos. (62262035, 62262034, 62062046, and 62162038).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Chaudhuri, S.; Ganti, V.; Kaushik, R. A Primitive Operator for Similarity Joins in Data Cleaning. In Proceedings of the 22nd International Conference on Data Engineering (ICDE'06), Atlanta, GA, USA, 3–7 April 2006.
2. Rong, C.; Lu, W.; Wang, X.;Du, X.; Chen, Y.; Tung, A.K. Efficient and Scalable Processing of String Similarity Join. *IEEE Trans. Knowl. Data Eng.* **2013**, *25*, 2217–2230. [CrossRef]
3. Bayardo, R.J.; Ma, Y.; Srikant, R. Scaling up all pairs similarity search. In Proceedings of the 16th International Conference on World Wide Web, Banff, AB, Canada, 8–12 May 2007.

4. Xiao, C.; Wang, W.; Lin, X.; Yu, J.X. Efficient similarity joins for near duplicate detection. In Proceedings of the 17th International Conference on World Wide Web, Beijing, China, 21–25 April 2008.
5. Wang, J.; Li, G.; Feng, J. Can we beat the prefix filtering?: An adaptive framework for similarity join and search. In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, Scottsdale, AZ, USA, 20–24 May 2012.
6. Dong, D.; Li, G.; He, W.; Feng, J. An efficient partition based method for exact set similarity joins. *VLDB Endow.* **2015**, *9*, 360–371. [CrossRef]
7. Jia, L.; Xi, J.; Li, M.; Liu, Y.; Miao, D. ETI: An efficient index for set similarity queries. *Front. Comput. Sci.* **2015**, *6*, 700–712. [CrossRef]
8. Jia, L.; Zhang, L.; Yu, G.; You, J.; Ding, J.; Li, M. A Survey on Set Similarity Search and Join. *Int. J. Perform. Eng.* **2018**, *14*, 245. [CrossRef]
9. Zhang, Y.; Li, X.; Wang, J.; Zhang, Y.; Xing, C.; Yuan, X. An Efficient Framework for Exact Set Similarity Search Using Tree Structure Indexes. In Proceedings of the IEEE 33rd International Conference on Data Engineering (ICDE), San Diego, CA, USA, 19–22 April 2017; pp. 759–770.
10. Bellas, C.; Gounaris, A. HySet: A hybrid framework for exact set similarity join using a GPU. *Parallel Comput.* **2021**, *104*, 102790. [CrossRef]
11. Yang, J.; Zhang, W.; Wang, X.; Zhang, Y.; Lin, X. Distributed Streaming Set Similarity Join. In Proceedings of the 2020 IEEE 36th International Conference on Data Engineering (ICDE), Dallas, TX, USA, 20–24 April 2020.
12. Ma, Y.; Zhang, R.; Cui, Z.; Lin, C. Projection Based Large Scale High-Dimensional Data Similarity Join Using MapReduce Framework. *IEEE Access* **2020**, *8*, 121665–121677. [CrossRef]
13. Arasu, A.; Ganti, V.; Kaushik, R. Efficient exact set-similarity joins. In Proceedings of the 2006 Very Large Data Bases Conference (VLDBC), Seoul, Republic of Korea, 12–15 September 2006.
14. Bouros, P.; Ge, S. Mamoulis. Spatio-textual similarity joins. *VLDB Endow.* **2012**, *6*, 1–12. [CrossRef]
15. Mann, W.; Augsten, N.; Bouros, P. An Empirical Evaluation of Set Similarity Join Techniques. *VLDB Endow.* **2016**, *9*, 636–647. [CrossRef]
16. Yu, C.; Nutanong, S.; Li, H.; Cong, W.; Yuan, X. A Generic Method for Accelerating LSH-Based Similarity Join. In Proceedings of the 2017 IEEE International Conference on Data Engineering, San Diego, CA, USA, 19–22 April 2017.
17. McCauley, S.; Mikkelsen, J.W.; Pagh, R. Set Similarity Search for Skewed Data. In Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, 10–15 June 2018.
18. Zhang, Y.; Wu, J.; Wang, J.; Xing, C. A Transformation-Based Framework for KNN Set Similarity Search. *IEEE Trans. Knowl. Data Eng.* **2020**, *32*, 409–423. [CrossRef]
19. Li, Y.; Yu, X.; Koudas, N. LES3: Learning-based exact set similarity search. *VLDB Endow.* **2021**, *14*, 2073–2086. [CrossRef]
20. Vernica, R.; Carey, M.J.; Li, C. Efficient parallel set-similarity joins using MapReduce. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, Indianapolis, IN, USA, 6–10 June 2010.
21. Bay, S.D.; Kibler, D.F.; Pazzani, M.J.; Smyth, P. The UCI KDD archive of large data sets for data mining research and experimentation. *SIGKDD Explor.* **2000**, *2*, 81–85. [CrossRef]
22. Yang, J.; Zhang, W.; Yang, S.; Zhang, Y.; Lin, X. TT-Join: Efficient Set Containment Join. In Proceedings of the 2017 IEEE 33rd International Conference on Data Engineering, San Diego, CA, USA, 19–22 April 2017.