TT-Join: Efficient Set Containment Join

Jianye Yang[†], Wenjie Zhang[†], Shiyu Yang[†*], Ying Zhang[§], Xuemin Lin[‡†]
‡East China Normal University, China
[†]School of Computer Science and Engineering, University of New South Wales, Australia
[§]CAI and School of Software, University of Technology Sydney, Australia
{jianyey, zhangw, yangs, lxue}@cse.unsw.edu.au, ying.zhang@uts.edu.au

Abstract*—In this paper, we study the problem of set containment join. Given two collections $\mathcal{R}$ and $\mathcal{S}$ of records, the set containment join $\mathcal{R} \bowtie_{\subseteq} \mathcal{S}$ retrieves all record pairs $\{(r, s)\} \in \mathcal{R} \times \mathcal{S}$ such that $r \subseteq s$. This problem has been extensively studied in the literature and has many important applications in commercial and scientific fields. Recent research focuses on the in-memory set containment join algorithms, and several techniques have been developed following *intersection-oriented* or *union-oriented* computing paradigms. Nevertheless, we observe that two computing paradigms have their limits due to the nature of the intersection and union operators. Particularly, *intersection-oriented* method relies on the intersection of the relevant inverted lists built on the elements of $\mathcal{S}$. A nice property of the *intersection-oriented* method is that the join computation is verification free. However, the number of records explored during the join process may be large because there are multiple replicas for each record in $\mathcal{S}$. On the other hand, the *union-oriented* method generates a signature for each record in $\mathcal{R}$ and the candidate pairs are obtained by the union of the inverted lists of the relevant signatures. The candidate size of the *union-oriented* method is usually small because each record contributes only one replica in the index. Unfortunately, *union-oriented* method needs to verify the candidate pairs, which may be cost expensive especially when the join result size is large. As a matter of fact, the state-of-the-art *union-oriented* solution is not competitive compared to the *intersection-oriented* ones. In this paper, we propose a new *union-oriented* method, namely *TT-Join*, which not only enhances the advantage of the previous *union-oriented* methods but also integrates the goodness of *intersection-oriented* methods by imposing a variant of prefix tree structure. We conduct extensive experiments on 20 real-life datasets by comparing our method with 7 existing methods. The experiment results demonstrate that *TT-Join* significantly outperforms the existing algorithms on most of the datasets, and can achieve up to two orders of magnitude speedup.

I. Introduction

Set-valued attributes play an important role in modeling database systems ranging from commercial applications to scientific studies. For instance, a set-valued attribute may correspond to the profile of a person, the tags of a post, the links or domain information of a webpage, and the tokens or q-grams of a document. In the literature, there has been a variety of interest in the computation of set-valued attributes, including but not limited to set containment searches (e.g., [1], [2], [3], [4], [5], [6], [7]), set similarity joins (e.g., [8], [9], [10], [11], [12], [13], [14], [15]) and set containment join (e.g., [16], [17], [18], [19], [20], [21], [22], [23], [24]).

In this paper, we focus on the problem of *set containment join*. Given two collections $\mathcal{R}$ and $\mathcal{S}$ of records, each of which contains a set of elements, the set containment join, denoted by $\mathcal{R} \bowtie_{\subseteq} \mathcal{S}$, retrieves all pairs $\{(r, s)\}$ where $r \in \mathcal{R}$, $s \in \mathcal{S}$, and

[*]Corresponding author.

id	set
r_1	$\{e_1, e_2, e_3\}$
r_2	$\{e_1, e_2, e_4\}$
r_3	$\{e_1, e_3, e_4\}$
r_4	$\{e_2, e_5\}$

(a) $\mathcal{R}$ sets

id	set
s_1	$\{e_1, e_2, e_3, e_5\}$
s_2	$\{e_1, e_2, e_4\}$
s_3	$\{e_1, e_3, e_6\}$
s_4	$\{e_2, e_4, e_5\}$

(b) $\mathcal{S}$ sets

Fig. 1. A motivation example where e_i denotes a skill, $\mathcal{R}$ consists of four job advertisements with required skills, and $\mathcal{S}$ represents four job-seekers with their skills.

$r \subseteq s$. As a fundamental operation on massive collections of set values, the set containment join benefits many applications. For instance, companies may post a list of positions on an online job market website, and each of which contains a set of required skills. Let e_i denote a skill, Table 1(a) shows the skills required in four job advertisements in $\mathcal{R}$. A job-seeker, on the other hand, can submit her/his curriculum vitae to the website, which lists a set of her/his skills. Table 1(b) illustrates the skill records of four job-seekers in $\mathcal{S}$. Naturally, a company would like to consider a job-seeker if her/his skill set covers all required skills for a position. We call such a pair of job-seeker and position a containment match. By executing a set containment join on the positions and job-seekers, the website is able to identify all possible matches, i.e., $\mathcal{R} \bowtie_{\subseteq} \mathcal{S}$, and make recommendations.

An algorithmic challenge is how to perform the set containment join in an *efficient* way. A naive algorithm is to compare every pair of records from $\mathcal{R}$ and $\mathcal{S}$, thus bearing a prohibitively $\mathcal{O}(n_r n_s)$ time complexity where n_r and n_s denote the number of records in $\mathcal{R}$ and $\mathcal{S}$, respectively. In view of such high cost, the prevalent approach in the past is to develop disk-based algorithms [22], [23], [24], [16], [17] for this problem. We call these algorithms *union-oriented* methods because, as shown in Section III-B, the *union* is their core operator. In a high level, each record $r \in \mathcal{R}$ is assigned with a signature (e.g., bitmap), where an inverted index on $\mathcal{R}$ could also be built based on the signatures. For each $s \in S$, they generate all possible signatures by s or any of its subset. By computing the union of all the corresponding inverted lists, they obtain a set of candidate records within $\mathcal{R}$, each of which might be a subset of s. Then set containment join results are available after verifying the candidate pairs.

With the development of hardware and distributed computing infrastructure, a recent trend is to design efficient *in-memory* set containment join algorithms (e.g., [16], [18], [19], [20]). It is interesting that the state-of-the-art techniques follow a very different computing paradigm, namely *intersection-oriented* method, where details are introduced in Section III-A. In general, an inverted index is constructed based on *every* element of each record within $\mathcal{S}$. Then for a record $r \in \mathcal{R}$, we can identify records $s \in \mathcal{S}$ with $r \subseteq s$ by the *intersection* of the inverted lists built on $\mathcal{S}$ for all elements within r. Following this computing paradigm, instead of processing each record r within $\mathcal{R}$ individually, three variants of prefix tree structures are designed in [18], [19], [20] to share computation costs among records within $\mathcal{R}$.

Compared to *union-oriented* methods, *verification free* is the most judicious property of the *intersection-oriented* method, especially when the join result size is large. Our empirical study shows that the state-of-the-art in-memory *union-oriented* method [18], which is an extension of previous disk-based methods, has been significantly outperformed by the state-of-the-art in-memory *intersection-oriented* techniques. Nevertheless, we show that this benefit is off-set by the fact that *every* element in $s \in \mathcal{S}$ contributes to the inverted index due to the nature of *intersection* operator; that is, the ID of each record in $\mathcal{S}$ will be replicated in multiple inverted lists. This inevitably results in a large number of records visited in the join process, especially for the record $r \in \mathcal{R}$ with large size. With the same reason, we show that it is difficult for *intersection-oriented* method to exploit the skewness of the real-life data.

We are aware that there are several algorithms (e.g., [2], [13], [3]), which are devised for string similarity search, can also be utilized to handle set containment join with trivial modification. Algorithms in this category are called *adapted* methods, where the details are introduced in Section III-C.

In this paper, we re-visit and design a new *union-oriented* method, namely *TT-Join*, where an efficient set containment join algorithm is developed based on two different prefix trees built on $\mathcal{R}$ and $\mathcal{S}$, respectively. Through comprehensive cost analysis on simple *intersection-oriented* and *union-oriented* methods in Section IV-B, we show that the above two problems suffered by the *intersection-oriented* methods can be easily addressed by a new simple *union-oriented* method which uses the least frequent element as the signature. Not surprisingly, the new simple *union-oriented* method needs to verify candidates due to the inherent limit of *union-oriented* computing paradigm. Moreover, its pruning capability is limited by using only one element as the signature. To circumvent these limits, we propose a new prefix tree structure based on the k least frequent elements of the records within $\mathcal{R}$ such that we can (i) enhance the pruning power with a reasonable overhead, and (ii) integrate the *intersection* semantics to directly *validate* a significant number of join results without invoking the verification. To share the computational cost among records within $\mathcal{S}$, we also build a regular prefix tree on $\mathcal{S}$. Then we develop an efficient *TT-Join* algorithm to perform set containment join against two prefix trees.

Contributions. Our principle contributions are summarized as follows.

- We classify the existing solutions into two categories, namely *intersection-oriented* and *union-oriented* methods, based on the nature of their computing paradigms. Through comprehensive analysis on two simple *intersection-oriented* and *union-oriented* methods, we show the advantages and limits of the methods in each category.
- We propose a new *union-oriented* method, namely *TT-Join*. Particularly, we design a k least frequent elements based prefix tree structure, namely *kLFP-Tree*, to organize the records within $\mathcal{R}$. Together with a regular prefix tree constructed on records from $\mathcal{S}$, we develop an efficient set containment join algorithm.
- Our comprehensive experiments on 20 real-life set-valued data from various applications demonstrate the efficiency of our *TT-Join* algorithm. It is reported that *TT-Join* significantly outperforms the state-of-the-art algorithms on most of the datasets, and can achieve up to two orders of magnitude speedup.

Notation	Definition
$x, \mathcal{X}; r, \mathcal{R}; s, \mathcal{S}$	a record, a set of records
$e, \mathcal{E}$	an element, element domain
$\mathcal{R}(s)$	all records $r \in \mathcal{R}$ with $r \subseteq s$
$\mathcal{S}(r)$	all records $s \in \mathcal{S}$ with $r \subseteq s$
σ	signature of a record
$I_{\mathcal{R}}(\sigma)$	inverted list for signature σ in $\mathcal{R}$
$I_{\mathcal{S}}(e)$	inverted list for element e in $\mathcal{S}$
$T_{\mathcal{R}}, T_{\mathcal{S}}$	indexing tree on $\mathcal{R}$ / $\mathcal{S}$
v, w	a node in $T_{\mathcal{R}}$ / $T_{\mathcal{S}}$
$v.e, w.e$	record element in v / w
$v.set, w.set$	elements from root to v / w
$v.prefix, w.prefix$	elements from root to parent of v / w
$v.list, w.list$	records stop at v / w
$P(e)$	frequency distribution of elements
$\theta(l)$	distribution of record cardinality
$\lvert x\rvert_{avg}, \lvert r\rvert_{avg}, \lvert s\rvert_{avg}$	average size of records in $\mathcal{X}$, $\mathcal{R}$, $\mathcal{S}$
$\lvert x\rvert_{max}, \lvert r\rvert_{max}, \lvert s\rvert_{max}$	maximal size of records in $\mathcal{X}$, $\mathcal{R}$, $\mathcal{S}$

TABLE I. THE SUMMARY OF NOTATIONS

Road Map. The rest of the paper is organized as follows. Section II presents the preliminaries. Section III introduces the existing solutions. Our approach TT-Join is devised in Section IV. Extensive experiments are reported in Section V. Section VI concludes the paper.

II. PRELIMINARIES

In this section, we introduce basic concepts and definitions used in the paper. Table I summarizes the important mathematical notations used throughout this paper.

In this paper, each record x consists of a set of elements $\{e_1, e_2, \ldots, e_{|x|}\}$ from element domain $\mathcal{E}$. We use $\mathcal{X}$ to denote a relation with a set-valued attribute, i.e., a collection of records. By default, elements in a record are in decreasing order of their frequency in $\mathcal{X}$. Following the convention, we use $\mathcal{R}$ (resp. $\mathcal{S}$) to denote the left (resp. right) side relation (i.e., a collection of records) for the set containment join. Similar, we use r (resp. s) to denote a record within $\mathcal{R}$ (resp. $\mathcal{S}$).

Given two records r and s, we say r is contained by s, denoted by $r \subseteq s$, if all elements of r can be found in s. That is, for $\forall e \in r$, we have $e \in s$. In the paper, we also say r is a *subset* of s and s is a *superset* of r if $r \subseteq s$. For a record $r \in \mathcal{R}$, we use $\mathcal{S}(r)$ to denote all records $s \in \mathcal{S}$ with $r \subseteq s$. Similarly, $\mathcal{R}(s)$ denotes all records $r \in \mathcal{R}$ with $r \subseteq s$.

*Definition 1 (**Set Containment Join**):* Given two collections $\mathcal{R}$ and $\mathcal{S}$ of records, the set containment join between $\mathcal{R}$ and $\mathcal{S}$, denoted by $\mathcal{R} \bowtie_{\subseteq} \mathcal{S}$, is to find all pairs (r, s), such that $r \in \mathcal{R}$, $s \in \mathcal{S}$, and $r \subseteq s$. That is $\mathcal{R} \bowtie_{\subseteq} \mathcal{S} = \{(r, s) | r \in \mathcal{R}, s \in \mathcal{S}, \text{ and } r \subseteq s\}$.

Example 1: Consider the example in Fig. 1. The result of set containment join is as follows: $\mathcal{R} \bowtie_{\subseteq} \mathcal{S} = \{(r_1, s_1), (r_2, s_2), (r_4, s_1), (r_4, s_4)\}$.

III. EXISTING SOLUTIONS

A brute-force solution for set containment join is to enumerate and verify $|\mathcal{R}||\mathcal{S}|$ pairs of records, which is cost-prohibitive. To improve the efficiency of computation, many advanced algorithms are proposed in the literature. We classify them into two categories based on their computing paradigms, namely *intersection-oriented* methods [16], [17], [18], [19], [20] and *union-oriented* methods [21], [22], [23], [24], [18]. We also review several methods proposed for string similarity search [2], [13], [3].

A. ***Intersection-Oriented Methods***

Given two record collections $\mathcal{R}$ and $\mathcal{S}$, the key idea of *intersection-oriented* method is to build inverted index on $\mathcal{S}$ and then apply the **intersection** operator to calculate $\mathcal{R} \bowtie_{\subseteq} \mathcal{S}$. In this paper, we say these algorithms are $\mathcal{S}$-driven methods because their main index structures are built on $\mathcal{S}$.

Algorithm 1 illustrates a simple *intersection-oriented* method [16], namely **RI-Join**[*]. We use $I_{\mathcal{S}}(e)$ to denote the inverted list of an element e built on records in $\mathcal{S}$, which keeps IDs of the records containing the element e. Fig. 2 depicts the inverted index of $\mathcal{S}$ in the example of Fig. 1. Lines 1-2 build the inverted index of $\mathcal{S}$. Then for each record $r \in \mathcal{R}$, we can immediately identify $\mathcal{S}(r)$ (i.e., record $s \in \mathcal{S}$ with $r \subseteq s$) based on the intersection of the inverted lists for elements within r (Lines 4-6).

Algorithm 1: **RI-Join** ($\mathcal{R}$, $\mathcal{S}$)

```
  Output : $\mathcal{R} \bowtie_{\subseteq} \mathcal{S}$
1 for each record $s \in \mathcal{S}$ do
2     Update inverted list $I_{\mathcal{S}}(e)$ for every $e \in s$;
3 $J := \emptyset$;
4 for each $r \in \mathcal{R}$ do
5     $C := \bigcap_{e \in r} I_{\mathcal{S}}(e)$;
6     $J := J \cup \{(r, s)\}$ for every record $s \in C$;
7 return $J$
```

The dominant cost of Algorithm 1 is the intersection of the inverted lists (Line 5). We have

$$Cost(\mathcal{R} \bowtie_{\subseteq} \mathcal{S}) = \sum_{r \in \mathcal{R}} \sum_{e \in r} |I_{\mathcal{S}}(e)|. \tag{1}$$

Analysis. A nice property of the *intersection-oriented* approach is *verification free*. On the downside, a significant drawback is that we need to consider every element of a record for inverted index construction (Line 2). This may lead to long inverted lists and hence a large number of records accessed during the join process (Line 5).

Below are details of advanced *intersection-oriented* set containment join algorithms.

Algorithm PRETTI. Jampani *e*t. al [17] propose a method called PRETTI to improve the performance of *intersection-oriented* method. Instead of processing each individual record in $\mathcal{R}$, a prefix tree $T_{\mathcal{R}}$ is built on $\mathcal{R}$ to share the computational cost. We define a (regular) prefix tree as follows.

*Definition 2 (**Prefix Tree**):* Each node v in the tree (except root) is associated to an element in $\mathcal{E}$, denoted by $v.e$. We use $v.set$ to denote the set of elements associated with v and its ancestors. Similarly, we denote all elements in its ancestors by $v.prefix$ (i.e., $v.prefix := v.set \setminus v.e$). We also use a list, denoted by $v.list$, to keep the IDs of all records $\{x\}$ with $x = v.set$. Note that elements in each record follow a global order, and hence each record is assigned to a unique tree node.

Fig. 3 shows the prefix tree for the record set $\mathcal{R}$ in Fig. 1(a). By utilizing the prefix tree, we can share computation among records with the same prefix. For instance, the intersection for

[*]Algorithm 1 is named **RI-Join** in this paper since there is no index on $\mathcal{R}$ and an inverted index is built on $\mathcal{S}$.

Algorithm 2: **PRETTI($T_{\mathcal{R}}$, $I_{\mathcal{S}}$)**

```
   Input  : $T_{\mathcal{R}}$ : prefix tree on $\mathcal{R}$, $I_{\mathcal{S}}$ : inverted indexes on $\mathcal{S}$,
   Output : $\mathcal{R} \bowtie_{\subseteq} \mathcal{S}$
 1 for each child node $v$ of the root of $T_{\mathcal{R}}$ do
 2     processNode($v$, $I_{\mathcal{S}}(v.e)$, $J$);
 3 return $J$
 4 procedure processNode($v$, $list$, $J$)
 5 $list \leftarrow list \cap I_{\mathcal{S}}(v.e)$;
 6 for each record $r \in v.list$ do
 7     for each record $s \in list$ do
 8         $J \leftarrow J \cup \{(r, s)\}$;
 9 for each child node $v_i$ of node $v$ do
10     processNode($v_i$, $list$, $J$);
```

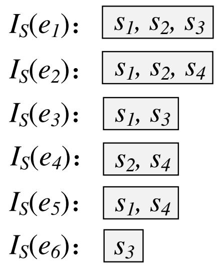

$I_{\mathcal{S}}(e_1)$: s_1, s_2, s_3
$I_{\mathcal{S}}(e_2)$: s_1, s_2, s_4
$I_{\mathcal{S}}(e_3)$: s_1, s_3
$I_{\mathcal{S}}(e_4)$: s_2, s_4
$I_{\mathcal{S}}(e_5)$: s_1, s_4
$I_{\mathcal{S}}(e_6)$: s_3

Fig. 2. Inverted index on $\mathcal{S}$

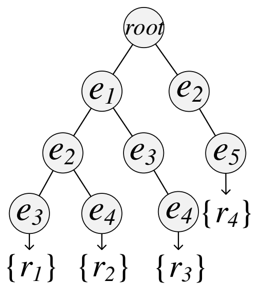


Fig. 3. Prefix tree on $\mathcal{R}$

inverted lists of $I_{\mathcal{S}}(e_1)$ and $I_{\mathcal{S}}(e_2)$ only needs to be performed once when we compute the superset of r_1 and r_2.

Algorithm 2 illustrates the details of PRETTI, which traverses the prefix tree on $\mathcal{R}$ in a depth-first manner. For each node v visited, we use $list$ to denote the intersection of the inverted lists of the elements in $v.prefix$, which is passed from its parent node. Based on the intersection of $list$ and the inverted list of the element $v.e$ $I_{\mathcal{S}}(v.e)$ (Line 5), we obtain the list of records in $\mathcal{S}$ each of which contains all elements in $v.set$. Lines 6-8 generate the join results regarding the node v. Then the join will continue through its child nodes where the updated $list$ will be passed (Lines 9-10).

Algorithm PRETTI+. To reduce the size of the prefix tree, Luo *e*t. al [18] introduce an extension of PRETTI, namely PRETTI+. In particular, PRETTI+ employs a compact prefix tree, called Patricia trie, to replace the prefix tree in PRETTI. This new prefix tree is the same as the previous one except that the nodes along a single path are merged into one node. The Patricia trie on the records set $\mathcal{R}$ in Fig. 1(a) is shown in Fig. 4. We omit the details of PRETTI+, which are the same as PRETTI except that we may need to merge inverted lists of multiple elements associated with a node.

Algorithm LIMIT. To avoid exploring many inverted lists for the large size records within $\mathcal{R}$, Bouros *e*t. al [19] propose a new algorithm, called LIMIT. Instead of building a complete prefix tree for $\mathcal{R}$, LIMIT only builds a prefix tree with limited height k; that is, only considers the prefix of record with a fixed length. Fig. 5 shows a limited prefix tree with $k = 2$ for records set $\mathcal{R}$ in Fig. 1(a).

Based on the limited prefix tree, LIMIT performs the join process following a two-phase procedure which involves *candidates generation* and *candidates verification*. In terms of algorithm implementation, LIMIT is basically the same as Algorithm 2 except the generation of join results (Lines 8 in Algorithm 2). Particularly, LIMIT handles this by considering two scenarios. If $|r| \leq k$, we output the record pair (r, s)

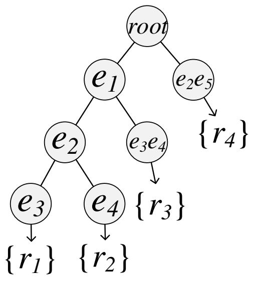


Fig. 4. Patrica trie on $\mathcal{R}$

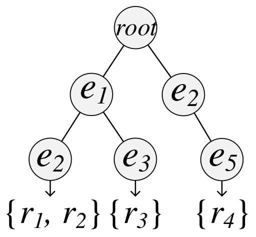


Fig. 5. Limited tree on $\mathcal{R}$

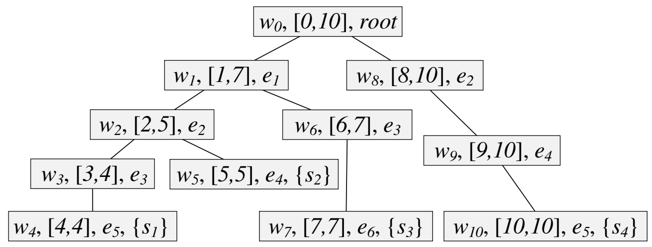


Fig. 6. Augmented prefix tree on $\mathcal{S}$

directly since the inverted lists of *all* elements in r participate in the intersection. Otherwise, we have to verify the record pair (r, s). Although we need to verify some candidate pairs in LIMIT due to the limited tree height, this cost is well paid off by significantly reducing the number of inverted lists involved in the intersection. As a matter of fact, our empirical study shows that LIMIT is the most promising *intersection-oriented* method on most of the datasets evaluated.

Algorithm PIEJoin. Recently, Kunkel *e*t. al [20] propose a two-tree based method, called PIEJoin, which aims to improve the performance of *intersection-oriented* method by exploiting advanced index technique on $\mathcal{S}$. PIEJoin builds two prefix trees $T_{\mathcal{R}}$ and $T_{\mathcal{S}}$ on relations $\mathcal{R}$ and $\mathcal{S}$, respectively, together with auxiliary structures on each tree node. In particular, for $T_{\mathcal{R}}$, each node is labeled with a preorder ID (e.g., $v_0, ..., v_9$ in Fig. 3), while for $T_{\mathcal{S}}$, there is a preorder interval on each node. Fig. 6 shows the augmented prefix tree $T_{\mathcal{S}}$ for $\mathcal{S}$ in Fig. 1(b).

The details of PIEJoin are illustrated in Algorithm 3, which traverses two prefix trees simultaneously. The search starts from the root of $T_{\mathcal{R}}$ and $T_{\mathcal{S}}$ (Line 1). On each tree node pair v and w, we check if there are some join pairs found (Line 4). In particular, if $v.list$ is not empty (Line 10), then we find all records in the subtree rooted at w and enumerate join pairs (Lines 11-14). After collecting results in current tree node pair, we go further by traversing the children of v. For each child v_i, we find the descendants of w such that the element contained in these nodes is $v_i.e$ (Line 6). We then recursively conduct the search process for each node pair v_i and w_j (Line 8).

Compared to the previous solutions, PIEJoin employs a tree structure on records in $\mathcal{S}$, instead of the inverted index. This alleviates the problem of the large size inverted lists for $\mathcal{S}$. However, some auxiliary structures have to be engaged to facilitate the node match at Line 6. Note that we need to find the matches within the whole subtree, which may be cost expensive. As reported in [20], the performance of PIEJoin is not competitive compared with LIMIT [19], which builds inverted index on $\mathcal{S}$, under most of the datasets evaluated.

B. Union-Oriented Methods

In general, all methods in this category use *signature-based* techniques. Let $\mathcal{L}$ denote the domain of the signature values,

Algorithm 3: **PIEJoin($T_{\mathcal{R}}$, $T_{\mathcal{S}}$)**

```
   Input  : $T_{\mathcal{R}}$ prefix tree on $\mathcal{R}$, $T_{\mathcal{S}}$ : prefix tree on $\mathcal{S}$
   Output : $\mathcal{R} \bowtie_{\subseteq} \mathcal{S}$
 1 search($T_{\mathcal{R}}.root$, $T_{\mathcal{S}}.root$, $J$);
 2 return $J$
 3 procedure search($v$, $w$, $J$)
 4 lookForOutput($v$, $w$, $J$);
 5 for each child node $v_i$ of node $v$ do
 6     $\mathcal{W} \leftarrow T_{\mathcal{S}}$.findNodes($w$, $v_i.e$);
 7     for each child node $w_j \in \mathcal{W}$ do
 8         search($v_i$, $w_j$, $J$);
 9 procedure lookForOutput($v$, $w$, $J$)
10 if $v.list \neq \emptyset$ then
11     $list \leftarrow T_{\mathcal{S}}$.getRecords($w$);
12     for each tuple $r \in v.list$ do
13         for each tuple $s \in list$ do
14             $J \leftarrow J \cup \{(r, s)\}$;
```

we use a hash function h to map a record x into a subset of signature values, denoted by $h(x)$, with $h(x) \subseteq \mathcal{L}$. For instance, in the partition-based containment join algorithm [22], a record x will be mapped into a number between 0 and $k-1$. These algorithms are also named $\mathcal{R}$-driven methods because the main index is built on records in $\mathcal{R}$.

Given two record collections $\mathcal{R}$ and $\mathcal{S}$, the key idea of *union-oriented* method is to generate **candidate records** within $\mathcal{R}$ for each record $s \in \mathcal{S}$ by the **union** of the inverted lists for the relevant signatures. Algorithm 4 illustrates a framework of simple *union-oriented* method. Lines 1-2 build inverted lists for possible signatures on $\mathcal{R}$. For each record $r \in \mathcal{R}$, Line 2 attaches its ID to the inverted list of the corresponding signature, denoted by $I_{\mathcal{R}}(\sigma)$. Then Lines 4-7 generate containment join result candidates based on the union of the inverted lists of the signatures. For a record $s \in \mathcal{S}$, we consider the inverted lists of the signatures generated by s or any of its subsets. Line 8 verifies the candidate pairs within J to remove the false positives. Note that in the implementation, we usually do not need to explicitly enumerate all subsets of s to generate M_s as shown at Line 5. Instead, M_s can be generated efficiently by exploiting the characteristics of the specific signatures used.

Algorithm 4: **A framework of simple *union-oriented* method($\mathcal{R}$, $\mathcal{S}$)**

```
   Output : $\mathcal{R} \bowtie_{\subseteq} \mathcal{S}$
 1 for each $r \in \mathcal{R}$ do
 2     $\sigma \leftarrow h(r)$; Update $I_{\mathcal{R}}(\sigma)$;
 3 $J := \emptyset$;
 4 for each $s \in \mathcal{S}$ do
 5     $M_s \leftarrow$ all possible signatures can be generated by $s$ or
       any of its subsets;
 6     $C \leftarrow \bigcup_{\sigma \in M_s} I_{\mathcal{R}}(\sigma)$;
 7     $J := J \cup \{(r, s)\}$ for every record $r \in C$;
 8 Verify candidate pairs within $J$;
 9 return $J$
```

The dominant cost of Algorithm 4 is the union of the inverted lists (Line 5) , denoted by C_{filter}, and the verification cost (Line 8), denoted by C_{vef}. We have

$$Cost(\mathcal{R} \bowtie_{\subseteq} \mathcal{S}) = C_{filter} + C_{vef} = \sum_{s \in \mathcal{S}} \sum_{\sigma \in M_s} |I_{\mathcal{R}}(\sigma)| + C_{vef}. \quad (2)$$

partition	$R_i \bowtie S_i$
0	$\{r_2\} \bowtie \{s_2, s_4\}$
1	$\{r_1\} \bowtie \{s_1, s_2, s_3, s_4\}$
2	$\{r_4\} \bowtie \{s_1, s_2, s_3, s_4\}$
3	$\{r_3\} \bowtie \{s_1, s_3\}$

Fig. 7. Partition-based method

Analysis. Compared with the *intersection-oriented* method in Algorithm 1, we need to verify the candidate pairs due to the nature of signature techniques, which usually brings false positives. Nevertheless, the advantage is that there is only one signature for each record. This leads to a smaller inverted index, and hence a smaller number of records explored during the join process (Line 6).

Below are details of the existing *union-oriented* algorithms classified based on their signature techniques.

Partition-based Techniques. In [22], a partition based method is proposed, where the signature of a record x is a number between 0 and $k-1$. For a given record $r \in \mathcal{R}$, the hash function h randomly selects an element e from r and maps e to an integer value between 0 and $k-1$ (Line 2 of Algorithm 4). We use this value as the signature of r. For a given record $s \in \mathcal{S}$, the signature subset M_s (Line 5 of Algorithm 4) consists of different signatures generated by all elements of s. Obviously, for two given collections $\mathcal{R}$ and $\mathcal{S}$, we can divide the candidate join pairs into k partitions. Fig. 7 shows the partitions for datasets in Fig. 1 where we assume that $k = 4$ and an element e_i is mapped into the value $(i \mod k)$. Several follow-up studies [23], [24] propose more sophisticated partitioning strategies (i.e., hash function h) to reduce the number of candidates in the partition pairs.

Bitmap-based Techniques. Helmer *e*t. al [21] use a bitmap as the signature of a record x, and a bitmap consists of fixed b bits. Given two bitmaps b_1 and b_2, we say $b_1 \subseteq b_2$ if every 1 bit in b_1 is also set to 1 in b_2. An important property of bitmap-based signature is that we have $h(x) \subseteq h(y)$ if $x \subseteq y$ for any two records x and y. This implies that, for a given record $s \in \mathcal{S}$ we can safely exclude a record $r \in \mathcal{R}$ from containment join if $h(r) \nsubseteq h(s)$. However, the task of enumerating all possible signatures by a record s or any of its subset (Line 5 of Algorithm 4) would be a bottleneck for the bitmap-based *union-oriented* method, since the the number of subsets of a signature is exponential to the bitmap length b. To avoid such a straightforward way of enumerating the subsets of a signature, Luo *e*t. al [18] recently propose a new algorithm, named **PTSJ**, based on a *trie-based* subsets enumeration method. In this method, the signatures of records in $\mathcal{R}$ are stored in a trie, where the leaf nodes store the record id in $\mathcal{R}$. Then, given a record $s \in \mathcal{S}$, we employ a breath-first search on the trie. For each tree node v, we store the bit values 0 and 1 in the left child and right child, respectively. If the corresponding bit value of $h(s)$ is 0, we only explore the left child. Otherwise, we will visit both children. Once we finish this traversal, the records in leaf nodes we accessed are the candidates.

Discussion. PTSJ algorithm proposed in [18] is the state-of-the-art in-memory *union-oriented* method which significantly enhances the previous solutions in this category by advanced signature enumeration method and careful bitmap length selection. Nevertheless, our empirical study shows that PTSJ is not competitive compared with other state-of-the-art *intersection-oriented* solutions. According to our analysis in Section IV-B2, PTSJ has two significant drawbacks: (i) does not utilize the data distribution; and (ii) needs to verify all candidate pairs obtained.

C. *Apply Set Similarity based Methods*

We are aware that existing set similarity search/join algorithms can be applied to support set containment join by setting specific thresholds. In this paper, we consider three representative works. It is worth mentioning that they are $\mathcal{S}$-driven methods in the sense that their main index structures are built on records from $\mathcal{S}$.

Li *e*t. al [2] propose an efficient list merging algorithm, named DivideSkip, to solve the generalized T-occurrence query problem. Given a query record Q, T-occurrence problem is to find the set of record IDs that appear at least T times on the inverted lists of the elements in Q, where the inverted lists are built on $\mathcal{S}$. By setting T to the size of Q, DivideSkip can be immediately employed to process set containment search. Using a nested loop, DivideSkip can also be extended to compute set containment join.

Wang *e*t. al [13] propose an adaptive framework for set similarity search, which adaptively selects the length of record prefix to build the inverted index. Since they apply the overlap similarity to handle different set similarity functions, by setting the overlap threshold T to the size of query Q, this framework can also be utilized to compute set containment join.

Agrawal *e*t. al [3] study the problem of error-tolerant set containment search. To boost the query performance, they propose an frequent element set based index structure that builds inverted index on careful chosen element set. By setting the error-tolerate threshold as 1, this index structure can also be applied to answer exact set containment query, and therefore, is also applicable to set containment join.

IV. Our Approach

In this section, we introduce a new in-memory set containment join algorithm, namely *TT-Join*, based on two tree structures constructed on $\mathcal{R}$ and $\mathcal{S}$, respectively.

A. *Motivation*

Our empirical study suggests that the existing competitive in-memory set containment join algorithms follow the *intersection-oriented* computing paradigm. However, to enjoy the nice property of verification free, we need to keep the ID of a record for each of its elements in the inverted index. This is an inherent limit of the *intersection-oriented* method which may lead to a large number of records explored during the join processing, especially when the number of inverted lists involved is large. Although an augmented prefix tree has been proposed in PIEJoin [20] to alleviate this issue, our empirical study suggests that the result is unsatisfactory due to the complicated data structure and expensive search cost incurred. Moreover, our analysis in Section IV-B2 also suggests that it is difficult for *intersection-oriented* methods to exploit the data distribution.

This motivates us to re-visit and design a new *union-oriented* approach. The drawbacks of existing *union-oriented* methods are two-fold: (i) the signature techniques used are data-independent, which cannot better exploit the distribution of the elements; (ii) they need to verify all candidate pairs. In this paper, we aim to design a new *union-oriented* method which not only enhances the nice property of *union-oriented* methods (i.e., small inverted list size) but also effectively addresses the above two issues.

In Section IV-B, we apply the ranked key [1] technique to use the least significant element as the signature of the record in the simple *union-oriented* algorithm (Algorithm 4). Through comprehensive cost analysis, we show that the performance of the new simple *union-oriented* method can significantly outperform that of simple *intersection-oriented* method (Algorithm 1) when data becomes skewed. It is rather intuitive to further enhance the filtering capacity by using k least frequent elements. We extend the inverted indexing of the new simple *union-oriented* method to accommodate the k least frequent elements based signature. Nevertheless, we show that a simple extension of inverted index is not promising due to the large overhead incurred.

This motivates us to impose a tree structure to accommodate the k least frequent elements based signatures. In Section IV-C, we build a prefix tree based on the k least frequent elements of the records in $\mathcal{R}$. By doing so, we can (i) further reduce the candidate size with a small overhead; (ii) naturally apply the intersection operator to validate a large number of candidates and hence reduce the verification cost. Together with a regular prefix tree constructed on $\mathcal{S}$, we develop an efficient set containment join algorithm, namely *TT-Join*.

B. *Inverted Index Based Method*

In this section, we introduce a simple *union-oriented* algorithm in Section IV-B1 which uses the least frequent element as the signature. Section IV-B2 conducts cost comparison between two simple *intersection-oriented* and *union-oriented* algorithms to reveal their inherent advantages and limits. Then Section IV-B3 investigates an extension of the inverted index to use k least frequent elements as the signature of a record such that the number of candidate pairs can be further reduced.

B1. Using the least frequent element (IS-Join)

As shown in Section III-B, different signature techniques are employed by the existing solutions to improve the performance of simple *union-oriented* method. However, none of them consider the distribution of the elements. To take advantage of the skewness of the real-life data, we apply the ranked key [1] technique to use the least significant element (i.e., least frequent element) as the signature of the record in the simple *union-oriented* algorithm (Algorithm 4). Our new simple *union-oriented* method, namely **IS-Join**[†], is immediate, based on two minor changes of Algorithm 4: (1) at Line 2, the hash function h simply returns the least frequent element as the signature; (2) at Line 5, M_s is the set of elements in s, i.e., considering $|s|$ signatures.

Algorithm correctness. For any result pair (r, s) (e.g., $r \subseteq s$), let σ be the signature of r (i.e., the least frequent element), we have $r \in I(\sigma)$. Since $r \subseteq s$, we have $\sigma \in s$ and hence $\sigma \in M_s$ at Line 5. It is immediate that $r \in C$ (Line 6). After verification at Line 8, *IS-Join* algorithm can identify the pair (r, s).

Next, we use the running example in Fig. 1 to show the advantage of our least frequent element based simple *union-oriented* method by comparing with the *RI-Join* (Algorithm 1).

***Example** 2:* The inverted index on $\mathcal{S}$ and the least frequent inverted index on $\mathcal{R}$ are shown in Fig. 2 and Fig. 8, respectively. According to Equation 1, we know that the cost for simple *intersection-oriented* method is 28, which is obtained

[†]The new simple *union-oriented* method is named **IS-Join** because an inverted index is built on $\mathcal{R}$ and there is no index on $\mathcal{S}$.

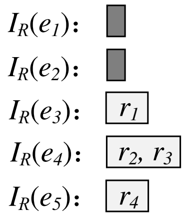


Fig. 8. Inverted index on $\mathcal{R}$

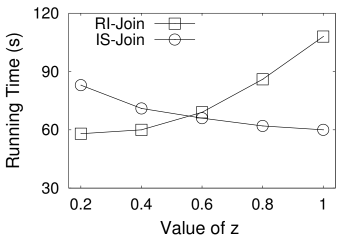


Fig. 9. Effect of data skewness

by summing up the size of all inverted list in $I_{\mathcal{S}}$ for each record. Similarly, we have that the candidate size of *union-oriented* method is 8 according to Equation 2, which means that the total cost is $8 \times \mathcal{T}_{vef}$, where $\mathcal{T}_{vef}$ is the cost to verify a candidate.

The above example shows the candidate set of our *union-oriented IS-Join* algorithm is much smaller compared to that of *intersection-oriented RI-Join* algorithm. When the verification cost of *IS-Join* algorithm is not expensive, it has a good chance to outperform *RI-Join* algorithm.

B2. Cost comparison

We now theoretically compare the expected costs of *RI-Join* (i.e., a simple *intersection-oriented* method in Algorithm 1) and the *IS-Join* algorithm (i.e., a simple *union-oriented* method in Algorithm 4 where the least significant element is used as the signature), denoted by C_{RI} and C_{IS}, respectively. We use $P(e)$ to denote the frequency distribution of an element $e \in \mathcal{X}$. Let $\theta(l)$ denote the probability that a record has l elements with $l \in [1, |x|_{max}]$ where $|x|_{max}$ is the maximum cardinality of a record in $\mathcal{X}$. In the cost analysis of this paper, we assume that $\mathcal{R}$ and $\mathcal{S}$ have the same distributions in terms of element frequency and record size. Moreover, we assume $|\mathcal{R}| = |\mathcal{S}| = n$, $|r|_{avg} = |s|_{avg} = m$, and the distributions are independent.

Estimating C_{RI}. Since each element of any record in $\mathcal{S}$ leads to one entry in the inverted lists $I_{\mathcal{S}}$, we know that the expected number of entries in the inverted index is $|\mathcal{S}| \times |s|_{avg}$ where $|s|_{avg} = \sum_{l=1}^{|s|_{max}} \theta(l) \times l$ is the average size of a record in $\mathcal{S}$. Therefore, the size of the inverted list $I_{\mathcal{S}}(e)$ can be estimated as follows:

$$|I_{\mathcal{S}}(e)| = P(e) \times |\mathcal{S}| \times |s|_{avg}. \tag{3}$$

According to Equation 1, we have

$$\begin{aligned} C_{RI} &= \sum_{r \in \mathcal{R}} \sum_{e \in r} |I_{\mathcal{S}}(e)| \\ &= |\mathcal{R}| \times |r|_{avg} \times \sum_{e \in \mathcal{E}} P(e)|I_{\mathcal{S}}(e)| \\ &= |\mathcal{R}| \times |r|_{avg} \times |S| \times |s|_{avg} \times \sum_{e \in \mathcal{E}} P(e)^2 \\ &= (nm)^2 \times \sum_{e \in \mathcal{E}} P(e)^2. \end{aligned} \tag{4}$$

Equation 4 shows that, when the number of records (n) and the average size of the records (m) are fixed, *RI-Join* will achieve its best performance when all elements have the same frequency because $\sum_{e \in \mathcal{E}} P(e) = 1$. This implies that the skewness of the frequency distribution will deteriorate the performance of this simple *intersection-oriented* method, while it is well known that many real-life data are skewed.

Estimating C_{IS}. We first estimate the size of inverted list $I_{\mathcal{R}}(e)$ for an element e. Given a record $r \in \mathcal{R}$, r is in $I_{\mathcal{R}}(e)$

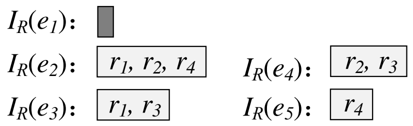


Fig. 10. 2 least frequent elements based inverted index on $\mathcal{R}$

if and only if $e \in r$ and there is no element $e' \in r$ with lower frequency than e. Thus, the probability that r within $I_{\mathcal{R}}(e)$, denoted by $P(r \in I_{\mathcal{R}}(e))$, is

$$P(r \in I_{\mathcal{R}}(e)) = P(e) \times F(e)^{|r|-1} = \sum_{l=1}^{|r|_{max}} \theta(l) \times l \times P(e) \times F(e)^{l-1}, \quad (5)$$

where $F(e) = \sum_{e' \prec e} P(e')$ is the cumulative probability before e where elements are ranked by frequency decreasing order; that is, $F(e)$ is the probability that a random chosen element has a higher frequency than e. Note that once an element e appears within the record r, it will serve as the signature with probability $F(e)^{|r|-1}$ due to the independent assumption. Thus, the expected size of list $I_{\mathcal{R}}(e)$ is as follows:

$$|I_{\mathcal{R}}(e)| = \sum_{r \in \mathcal{R}} P(r \in I_{\mathcal{R}}(e)) = |\mathcal{R}| \times \sum_{l=1}^{|r|_{max}} \theta(l) \times l \times P(e) \times F(e)^{l-1}. \quad (6)$$

According to Equation 2, we have

$$\begin{aligned} C_{IS} &= \sum_{s \in \mathcal{S}} \sum_{\sigma \in M_s} |I_{\mathcal{R}}(\sigma)| + C_{vef} \\ &= \sum_{s \in \mathcal{S}} \sum_{e \in s} |I_{\mathcal{R}}(e)| + C_{vef} \\ &= |\mathcal{S}| \times |s|_{avg} \times \sum_{e \in \mathcal{E}} P(e) \times |I_{\mathcal{R}}(e)| + C_{vef} \\ &\overset{(6)}{=} (nm)^2 \times \sum_{e \in \mathcal{E}} P(e)^2 \times F(e)^{m-1} + C_{vef}. \quad (7) \end{aligned}$$

Compared with Equation 4, it is immediate that the number of records explored by our *union-oriented RI-Join* algorithm is smaller than that of *intersection-oriented IS-Join* algorithm since $F(e) < 1$. Our empirical study below clearly shows that this gain will eventually pay off the verification cost (C_{vef}) when the skewness of the data increases.

Empirical evaluation. To evaluate the impact of the skewness towards the performance of two algorithms, we conduct a simple experiment on synthetic datasets. In particular, we generate datasets where the frequency of the elements follow the well-known Zipfian distribution with exponent z value varying from 0.2 to 1. Note that the data skewness increases when z grows. The number of records and the average record size are set to $100,000$ and 10, respectively.

It is observed in Fig. 9 that *intersection-oriented IS-Join* algorithm outperforms our simple *union-oriented RI-Join* algorithm when z is small due to the extra verification cost of *RI-Join*. However, as z increases, the processing time of *IS-Join* continuously grows, while *RI-Join* can take great advantage of the skewness.

B3. Extending to k least frequent elements (kIS-Join)

According to the above cost analysis, the least frequent element is a promising signature for *union-oriented* methods. To enhance the pruning capacity, it is natural to consider k least frequent elements. Following the existing inverted index technique, now each record is mapped to k elements (signatures). Fig. 10 shows an example of the inverted index on $\mathcal{R}$ in Fig. 1(a) when $k = 2$.

Then, for a given record $s \in \mathcal{S}$, we count the number of appearances for the records in C (Line 6 in Algorithm 4). If a record $r \in C$ appears k times (i.e., all k least frequent elements of r are contained in s), r is a candidate. Otherwise, we can prune r directly. We use **kIS-Join** to denote this algorithm which corresponds to *IS-Join* algorithm when $k = 1$.

Estimating C_{kIS}. Similar to the cost analysis for *IS-Join* algorithm, we first estimate the size of inverted list $I_{\mathcal{R}}(e)$ for an element e. Note that $I_{\mathcal{R}}$ is the inverted index based on the k least frequent elements of records in $\mathcal{R}$. Given a record $r \in \mathcal{R}$, r is in $I_{\mathcal{R}}(e)$ iff e is one of r's k least frequent elements. Thus, the probability that r is in $I_{\mathcal{R}}(e)$, denoted by $P(r \in I_{\mathcal{R}}(e))$, is:

$$P(r \in I_{\mathcal{R}}(e)) \approx P(e) \times \sum_{i=1}^{k} F(e)^{l-i} = \sum_{l=1}^{|r|_{max}} \theta(l) \times l \times P(e) \times \sum_{i=1}^{k} F(e)^{l-i}. \quad (8)$$

Now, the size of list $I_{\mathcal{R}}(e)$ is as follows:

$$|I_{\mathcal{R}}(e)| = \sum_{r \in \mathcal{R}} P(r \in I_{\mathcal{R}}(e)) \overset{(8)}{=} |\mathcal{R}| \times \sum_{l=1}^{|r|_{max}} \theta(l) \times l \times P(e) \times \sum_{i=1}^{k} F(e)^{l-i}. \quad (9)$$

According to Equation 2, we have

$$\begin{aligned} C_{kIS} &= \sum_{s \in \mathcal{S}} \sum_{\sigma \in M_s} |I_{\mathcal{R}}(\sigma)| + C_{vef} \\ &= \sum_{s \in \mathcal{S}} \sum_{e \in s} |I_{\mathcal{R}}(e)| + C_{vef} \\ &= |\mathcal{S}| \times |s|_{avg} \times \sum_{e \in \mathcal{E}} P(e) \times |I_{\mathcal{R}}(e)| + C_{vef} \\ &\overset{(9)}{=} (nm)^2 \times \sum_{e \in \mathcal{E}} P(e)^2 \times \sum_{i=1}^{k} F(e)^{m-i} + C_{vef}. \quad (10) \end{aligned}$$

By comparing Equation 10 and Equation 7, we know that the later is a special case of the former when $k = 1$. Clearly, on one hand, the pruning cost of C_{kIS} increases with k because *kIS-Join* touches more records due to the large inverted index size. On the other hand, the verification cost C_{vef} decreases with k since a larger k can prune more non-promising records. Therefore, there is a trade-off between these two costs. Our experimental results in Section V-B show that the performance gain for C_{vef} brought by a larger k value usually cannot pay-off the increased pruning costs.

C. Tree Based Method (TT-Join)

It is rather intuitive that the pruning power of our simple least frequent element based *union-oriented* method can be enhanced by increasing k. However, as shown in the above analysis and empirical study, the overhead cost brought by a straightforward extension of the inverted index is expensive and the gain of the enhanced pruning capacity may not be well paid off. In this subsection, we aim to develop a new *union-oriented* algorithm which enables us to: (i) enhance the pruning capacity with small overhead; and (ii) output some join result pairs during the tree traversal without going to the verification phase. Section IV-C1 introduces the k-length least frequent prefix tree structure, namely *kLFP-Tree*, which is built on records in $\mathcal{R}$. Together with a prefix tree constructed on records in $\mathcal{S}$, Section IV-C2 presents our *TT-Join* algorithm by traversing two prefix trees simultaneously. Section IV-C3 conducts performance analysis on the *TT-Join* algorithm.

C1. k-length least frequent prefix tree (*kLFP-Tree*)

The *kLFP-Tree* is constructed based on the k-length least frequent prefix of each record, which is defined as follows.

*Definition 3 (**k-length least frequent prefix**):* Given a record $x = \{e_1, ..., e_n\}$, we define $\{e_n, ..., e_{n-k+1}\}$ as its *k-length least frequent prefix*, denoted by $LFP_k(x)$. Note that, $LFP_k(x)$ is the reverse of x if $|x| \leq k$.

Given a set of k-length least frequent prefixes of the records in $\mathcal{R}$, the prefix tree (*kLFP-Tree*) is built up following Definition 2. Specifically, for each record x, we insert the last k elements (i.e., k least frequent elements in x) into the prefix tree following the reverse order, and it takes $O(1)$ time to insert each element as a hash table is used to maintain child entries for each node in *kLFP-Tree*. Thus, the time complexity to construct *kLFP-Tree* is $O(|\mathcal{R}|k)$. With the same time complexity, we may remove a record x in *kLFP-Tree* by deleting its k least frequent elements in order. Note that there is only one replica of a record x, whose ID is kept on the corresponding node of *kLFP-Tree* based on $LFP_k(x)$.

Example 3: Take the relation $\mathcal{R}$ in Fig. 1(a) as an example. When $k = 2$, we have $LFP_k(r_1) = \{e_3, e_2\}$, $LFP_k(r_2) = \{e_4, e_2\}$, $LFP_k(r_3) = \{e_4, e_3\}$, and $LFP_k(r_4) = \{e_5, e_2\}$. Then the corresponding *kLFP-Tree* is illustrated in Fig. 11(a).

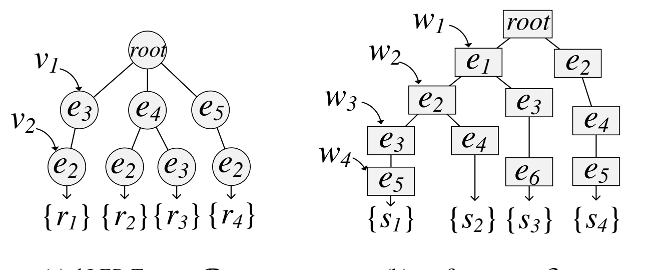
C2. *TT-Join* algorithm

We use $T_\mathcal{R}$ to denote the *kLFP-Tree* built on relation $\mathcal{R}$. To share computational cost, we also build a regular prefix tree for records in $\mathcal{S}$ following Definition 2, which is denoted by $T_\mathcal{S}$. Fig. 11 illustrates the example of $T_\mathcal{R}$ and $T_\mathcal{S}$ based on the records in Fig. 1. Note that we use a circle (resp. rectangle) to represent the node of the tree built on $\mathcal{R}$ (resp. $\mathcal{S}$), and each tree node is denoted by v_i (resp. w_i).

Algorithm 5 illustrates the details of *TT-Join* algoirthm. In general, we traverse $T_\mathcal{S}$ following a depth-first strategy (Lines 4-12). Lines 4-10 compute the relevant join result for each visited node w. Specifically, for the record s associated with w (i.e., $s = w.set$), we find all records in $\mathcal{R}(s)$. Recall that $\mathcal{R}(s)$ denotes the records within $\mathcal{R}$ which are a subset of s. We use R_1 to denote those records within $\mathcal{R}(s)$ without element $w.e$, and R_2 to denote the remaining records. In the procedure **processNode** (Lines 4-12), the *list* passed from the parent node corresponds to R_1 because we have $w.prefix \subset s$ and $w.prefix = s \backslash w.e$. Then Lines 5-7 identify the records in R_2. Particularly, Line 5 finds the node associated with element $w.e$ in $T_\mathcal{R}$. Then we only need to continue the search in its subtree because $w.e$ is the least frequent element in r. As shown in the procedure **traverse** (Lines 13-23), for each node v in $T_\mathcal{R}$ accessed, Line 17 can immediately **validate** a record r in $v.list$ if $|r| \leq k$ (i.e., r is reported **without verification**). Otherwise, we need to verify if $r \subseteq s$ at Line 19. At Lines 20-22, we continue to find potential records within R_2 if any of the child nodes matches an element in $w.prefix$. After all records within R_2 are identified, we use the updated *list* to

Algorithm 5: **TT-Join**($T_\mathcal{R}$, $T_\mathcal{S}$, k)

```
   Input  : $T_\mathcal{R}$ : index tree on $\mathcal{R}$, $T_\mathcal{S}$ : index tree on $\mathcal{S}$,
            $k$ : length of least frequent prefix for $\mathcal{R}$
   Output : $\mathcal{R} \bowtie_\subseteq \mathcal{S}$
1  for each child node $w$ of the root of $T_\mathcal{S}$ do
2      processNode($w$, $\emptyset$, $J$);
3  return $J$
4  procedure processNode($w$, $list$, $J$)
5  $v \leftarrow$ findChild($T_\mathcal{R}$.root, $w.e$);
6  if $v \neq NULL$ then
7      $list \leftarrow list \cup$ traverse($v$, $w$);
8  for each record $s \in w.list$ do
9      for each record $r \in list$ do
10         $J \leftarrow J \cup \{(r, s)\}$;
11 for each child node $w_i$ of node $w$ do
12     processNode($w_i$, $list$, $J$);
13 procedure traverse($v$, $w$)
14 $list \leftarrow \emptyset$;
15 for each record $r \in v.list$ do
16     if $|r| \leq k$ then
17         $list \leftarrow list \cup \{r\}$;
18     else
19         verify($r, w.set, list$);
20 for each child node $v_i$ of node $v$ do
21     if $v_i.e \in w.prefix$ then
22         traverse($v_i$, $w$);
23 return $list$
```



(a) *kLFP-Tree* on $\mathcal{R}$ (b) prefix tree on $\mathcal{S}$

Fig. 11. Tree structures for tree based method

keep all records within $R_1 \cup R_2$. Lines 8-10 output the join results associated with the node w accessed.

Algorithm correctness. For any record $s \in \mathcal{S}$, s must appear in one of the tree nodes, say w, in $T_\mathcal{S}$. Because we traverse $T_\mathcal{S}$ in a depth-first manner, w must be considered during the traversal. For each reocord s in $w.list$, we can find all records $r \in \mathcal{R}$ with $r \subseteq s$. Particularly, every record r from R_1, which does not contain element $w.e$, will be passed from w's parent node because we have $r \subseteq w.set$ if $r \subseteq w.prefix$ and $w.set = w.prefix \cup w.e$. For any record $r \in R_2$, it must appear within the subtree rooted at node v with $v.e = w.e$ (Line 5) because $w.e$ is the least frequent element in r. Meanwhile, none of the record in R_1 may appear in this subtree since $w.e \notin r$ for every $r \in R_1$. For a record $r \in R_2$, we use v to denote the corresponding node of r in $T_\mathcal{R}$ with $r \in v.list$. Since we explore *all* child nodes v_i with $v_i.e \in w.prefix$ in the procedure **traverse**, we will eventually reach v and identify r. On the other hand, because we *only* explore child nodes v_i with $v_i.e \in w.prefix$, this implies that $v.set \subseteq w.set$ for every node v accessed in the procedure **traverse**. Consequently, all results validated at Lines 16-17 are correct. Thus, the join results on each node are complete and correct.

Example 4: Consider the example in Fig. 1. The index trees on $\mathcal{R}$ and $\mathcal{S}$ are shown in Fig. 11(a) and Fig. 11(b), respectively. We traverse $T_{\mathcal{S}}$ in a depth-first manner starting from w_1. We immediately turn to $T_{\mathcal{R}}$ to see if there is a child node of the root of $T_{\mathcal{R}}$ matching the element of w_1 (i.e., e_1). The answer is no. We then continue the traversal processing until at w_3 where we find a child node v_1 in $T_{\mathcal{R}}$ with $w_3.e = v_1.e$. Next, we switch to traverse $T_{\mathcal{R}}$ starting from v_1 in a depth-first manner and find that v_2 matches w_2. At this point, we get a non-empty list (i.e., r_1) in v_2, which means that we get a candidate. We then conduct the verification and find that the remaining element 1 of r_1 is in $w_3.set$. Therefore, r_1 is a subset of w_3. After that, we continue traversing $T_{\mathcal{S}}$ and reach w_4 where we would get two subsets r_1 and r_4. In particular, r_1 is passed from w_3 and r_4 is collected at w_4. Since the list of w_4 is not empty, we then generate join pairs, namely (r_1, s_1) and (r_4, s_1). We find the full join results after finishing traversing $T_{\mathcal{S}}$.

C3. Cost analysis

Next, we analyse the cost of *TT-Join*, followed by a cost comparison with *IS-Join* and *kIS-Join* introduced in Section IV-B.

Estimating C_{TT}. In *TT-Join*, we build the inverted index for the least frequent prefix of each record in $\mathcal{R}$, which means that the size of the inverted index is fixed at $|\mathcal{R}|$. Besides, because the inverted index is determined by the least frequent element of each record in $\mathcal{R}$, we have that the inverted index size is exactly the same as shown in Equation 6. On the other hand, for each least frequent prefix, we have to sequentially check whether a given record $s \in \mathcal{S}$ contains the remaining $k-1$ least frequent elements in the worst case. Therefore the overall cost of *TT-Join* is as follows:

$$\begin{aligned} C_{TT} &= \sum_{s \in \mathcal{S}} \sum_{\sigma \in M_s} |I_{\mathcal{R}}(\sigma)| + C_{check} + C_{vef} \\ &= (nm)^2 \times \sum_{e \in \mathcal{E}} P(e)^2 \times F(e)^{m-1} \\ &+ C_{check} + C_{vef}, \end{aligned} \tag{11}$$

where C_{check} is the overhead to check the least frequent elements.

Comparison with *IS-Join*. Equation 11 and Equation 7 indicate that, *TT-Join* and *IS-Join* have the same pruning cost. However, in terms of the verification cost, C_{vef} in Equation 11 is smaller than that in Equation 7, because *TT-Join* uses k least frequent elements as the signature of a record to enhance the pruning capacity. Therefore, with a reasonable checking cost C_{check}, *TT-Join* may benefit from increasing k.

Comparison with *kIS-Join*. Because both *kIS-Join* and *TT-Join* use the k least frequent elements as signature, C_{vef} in Equation 10 and Equation 11 are exactly the same. The experimental results in Section V-B show that the C_{check} is insignificant compared with the growth of the number of explored records when k increases. Therefore, compared with *kIS-Join*, *TT-Join* can achieve better trade-off by increasing k within a reasonable range (e.g., $1 \leq k \leq 5$ in our empirical study).

D. Discussion

In this subsection, we first briefly discuss the difference between proposed method and the techniques in the existing solutions. Then we discuss how to support set containment join in the context of data streams.

Comparison of *TT-Join* and other methods. As shown in Section III-A and III-C, both *intersection-oriented* methods and three modified similarity based methods are $\mathcal{S}$-driven where their main index is built on $\mathcal{S}$. The key of their technique is, for each record $r \in \mathcal{R}$, how to utilize the index structure on $\mathcal{S}$ to find all records in $\mathcal{S}$, each of which contains r. On the contrary, *TT-Join* is $\mathcal{R}$-driven since the main index structure is built on $\mathcal{R}$. Moreover, although all algorithms use the variants of the inverted index as the main index, we show in the paper that *TT-Join* keeps one copy of the ID for each record in $\mathcal{R}$ while $\mathcal{S}$-driven methods need to maintain multiple copies of the ID for each record in $\mathcal{S}$. Consequently, the corresponding join algorithm of *TT-Join* is different to that of existing solutions.

Handle streaming records in $\mathcal{R}$ ($\mathcal{S}$). Our *TT-Join* techniques can efficiently support the scenario where $\mathcal{S}$ is streaming because the main index of *TT-Join* is based on $\mathcal{R}$. For each incoming record $s \in \mathcal{S}$, we can directly apply *TT-Join* (Algorithm 5) with $\mathcal{T}_{\mathcal{S}} = \{s\}$. Although *TT-Join* can also support the scenario of streaming $\mathcal{R}$ by setting $\mathcal{T}_{\mathcal{R}} = \{r\}$, many optimization techniques proposed in this paper cannot be applied. With similar rationale, we can see the *intersection-oriented* method can support the streaming $\mathcal{R}$, but are not applicable to streaming $\mathcal{S}$. It will be interesting to devise efficient algorithm to support the scenario where records from both $\mathcal{R}$ and $\mathcal{S}$ come in a stream fashion.

V. Performance Studies

In this section, we empirically evaluated the performance of the proposed techniques. All experiments were conducted on PCs with Intel Xeon 2x2.3GHz CPU and 128GB RAM running Debian Linux.

A. Experimental Setup

Algorithms. In the experiment, we evaluated the following algorithms.

- **TT-Join**. Our approach proposed in Algorithm 5 in Section IV-C, where *kLFP-Tree* and a regular prefix tree are built on $\mathcal{R}$ and $\mathcal{S}$, respectively. By default, we set *k*=4 under all settings.
- **LIMIT**. *Intersection-oriented* algorithm proposed in [19] (Section III-A). The optimized version of LIMIT (OPJ) is employed for performance evaluation.
- **PIEJoin**. *Intersection-oriented* algorithm proposed in [20] (Section III-A).
- **PRETTI+**. *Intersection-oriented* algorithm proposed in [18] (Section III-A).
- **PTSJ**. *Union-oriented* algorithm proposed in [18] (Section III-B).
- **DivideSkip**. *Adapted* algorithm proposed in [2] (Section III-C).
- **Adapt**. *Adapted* algorithm proposed in [13] (Section III-C).
- **FreqSet**. *Adapted* algorithm proposed in [3] (Section III-C).

Among the 8 algorithms, DivideSkip and Adapt were implemented in C++, where the source codes were obtained from the authors of [2] and [13] respectively. The rest 6 algorithms were all implemented in Java and the JVM maximum heap size was set to 32GB. For LIMIT and PIEJoin, we obtained the source codes from the authors of [20] since the source code of LIMIT was implemented in C++ and the authors of [20] re-implemented LIMIT in Java. For PRETTI+ and PTSJ, we obtained the source code from the authors of [18]. We implemented FreqSet in Java, where FP-growth [37] method was

Dataset	Abbreviation	Type	Record	Elements	#Records	AvgLength	#Elements	z-value
Amazon [25]	AMAZ	Rating	Product	Rating	1,230,915	4.67	2,146,057	0.52
AOL [26]	AOL	Text	Query	Keyword	10,054,183	3.01	3,873,246	0.68
BMS [19]	BMS	Sale	Transaction	Product	515,597	6.53	1,657	1.07
Bookcrossing [27]	BOOKC	Rating	Book	User	340,523	3.38	105,278	0.6
Delicious [28]	DELIC	Folksonomy	User	Tag	833,081	98.42	4,512,099	0.56
Discogs [29]	DISCO	Affiliation	Artist	Label	1,754,823	3.02	270,771	0.75
Enron [30]	ENRON	Text	Email	Word	517,431	133.57	1,113,219	0.65
Flickr-london [19]	FLICKR-L	Folksonomy	Photo	Word/Tag	1,680,490	9.78	810,660	0.75
Flickr-set [18]	FLICKR-S	Folksonomy	Photo	Word/Tag	3,546,729	5.36	618,970	0.63
Kosarak [19]	KOSRK	Interaction	User	Link	990,001	8.10	41,269	0.9
Lastfm [31]	LAST	Interaction	User	Song	1,084,620	4.07	992	0.51
Linux [32]	LINUX	Interaction	Thread	User	337,509	1.78	42,045	0.81
Livejournal [33]	LIVEJ	Affiliation	User	Group	3,201,203	35.08	7,489,073	0.62
Netflix [19]	NETFLIX	Rating	Movie	Rating	480,189	209.25	17,770	0.33
Orkut [18]	ORKUT	Interaction	User	Community	1,853,285	57.16	15,293,693	0.13
Stack [34]	STACK	Rating	User	Post	545,196	2.39	96,680	0.54
Sualize [35]	SUALZ	Folksonomy	Picture	Tag	495,402	3.63	82,035	0.95
Teams [36]	TEAMS	Affiliation	Athlete	Team	901,166	1.52	34,461	0.39
Twitter [18]	TWITTER	Interaction	Partition	User	371,586	65.96	1,318	1.4
Webbase [18]	WEBBS	Web	Page	Outlink	168,707	463.64	15,146,263	0.04

TABLE II. CHARACTERISTICS OF DATASETS

employed to compute the frequent sets. Among the 4 state-of-the-art algorithms, PIEJoin and PRETTI+ were parameter free. For LIMIT, we followed the same strategy adopted by authors of [20], where parameter tuning was carried out manually and individually for each dataset. Particularly, for datasets used in [20], we used the parameters tuned in [20], and for the rest datasets, we tuned the best values individually. For PTSJ, we followed the strategy proposed by the authors, which showed that a suitable signature length was between 16 and 32 times of the average length of records. In the experiments, we applied the middle value 24 for PTSJ. It was demonstrated in [20] that the frequency order of elements in records had a huge impact for LIMIT, PIEJoin, and PRETTI+. Therefore, we followed their empirical conclusion to apply infrequent sort order for LIMIT and PIEJoin, and frequent sort order for PRETTI+, which were stated optimal for the corresponding algorithms. Among the three adapted algorithms, DivideSkip and Adapt were parameter free. For FreqSet, the frequency threshold a was set to 1000.

Datasets. We deployed 20 real-life datasets selected from different domains with various data properties. The detailed characteristics of the 20 datasets were shown in Table II. For each dataset, we showed the type of the dataset, what the record and element represent, the number of records in the dataset, the average record length, and the number of unique elements in the dataset. We also reported the z-value (skewness) of the top 500 most frequent elements on each dataset by assuming that data follows Zipfian distribution. The datasets covered all datasets deployed in the state-of-the-art algorithms. In specific, Flickr-set, Orkut, Twitter, and Webbase were used in [18] to evaluate PRETTI+ and PTSJ algorithms, while BMS, Flickr-london, Kosarak, and Netflix were used in [19] to evaluate LIMIT algorithm. All of the eight datasets (with bold font in Table II) were employed in [20] to evaluate PIEJoin. Same as the previous studies, we evaluated the self set containment join on the 20 datasets.

B. Performance Tuning

To better evaluate the impact of k value as well as the advantage of *kLFP-Tree* compared with the inverted index, we also implemented an algorithm, namely **IT-Join**, which was an extension of *kIS-Join* algorithm where records in $\mathcal{S}$ were organized by a regular prefix tree. The traversal of the prefix tree was exactly the same as *TT-Join* (Algorithm 5) and the process of each visited node was based on *kIS-Join* algorithm in Section IV-B3.

We chose four representative datasets from Table II, including DISCO, KOSRK, NETFLIX, and TWITTER, which covered different types of dataset, various values of the average record size, as well as different z values. Note that, besides *TT-Join* and *IT-Join*, we also reported the performance of *IT-Join* with $k = 1$ to see if the increase of k value in *TT-Join* and *IT-Join* algorithms got paid off.

Fig. 12 reported the running time of three algorithms on the above four datasets with k increasing from 1 to 5. It was observed that *IT-Join* can only benefit from small k values, such as 1 and 2, which implied that the performance gain from large k value can not pay-off the growth of filtering costs, i.e., the increase of the number of records explored on the inverted index. On the contrary, *TT-Join* performed much better than *IT-Join* when k increased. In particular, it can continuously benefit from the growth of k on KOSRK, while achieved the best performance when $k = 4$, $k = 2$ and $k = 3$ on DISCO, NETFLIX and TWITTER, respectively. This behaviour verified our cost analysis in Section IV-B and Section IV-C that the overhead of a straightforward extension of inverted index was expensive and the gain may not be well paid off, while *TT-Join* can achieve a much better trade-off. Since the performance of *IT-Join* was fully dominated by *TT-Join* under all datasets, it was excluded from the following experiments. By default, we set $k = 4$ for *TT-Join* algorithm for all datasets.

C. Comparison with Existing Algorithms

In this subsection, we compared our *TT-Join* algorithm with four state-of-the-art algorithms LIMIT, PIEJoin, PRETTI+, and PSTJ as well as three modified algorithms DivideSkip, Adapt, and FreqSet on all 20 datasets. Recall that LIMIT, PIEJoin, and PRETTI+ were *intersection-oriented* methods and PSTJ was *union-oriented* method.

Processing Time. The experiment results in terms of processing time were reported in Fig. 13. Besides the set containment join time, the processing time also included the index construction time because the indexes of all algorithms were generated on the fly. It was reported that our *TT-Join* algorithm outperforms all state-of-the-art algorithms on all datasets, except that it was slightly outperformed by LIMIT on NETFLIX. Among the existing algorithms, LIMIT achieved the best performance

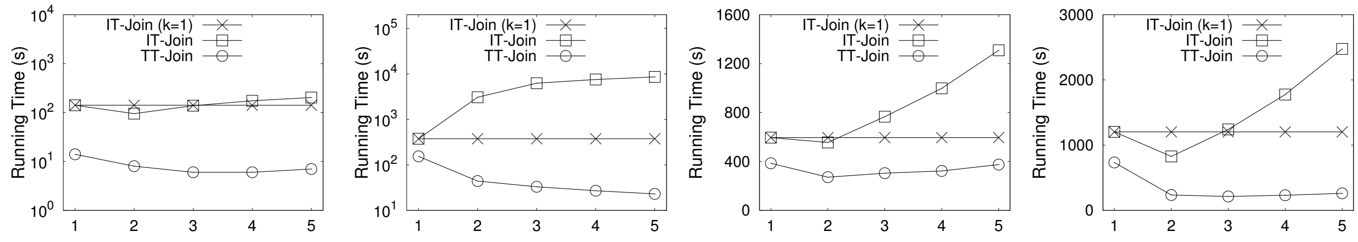

(a) DISCO (b) KOSRK (c) NETFLIX (d) TWITTER

Fig. 12. Effect of k on running time

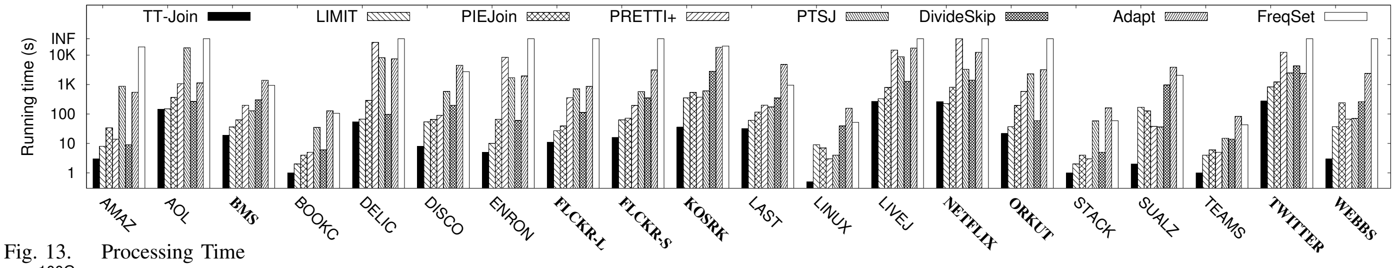


Fig. 13. Processing Time

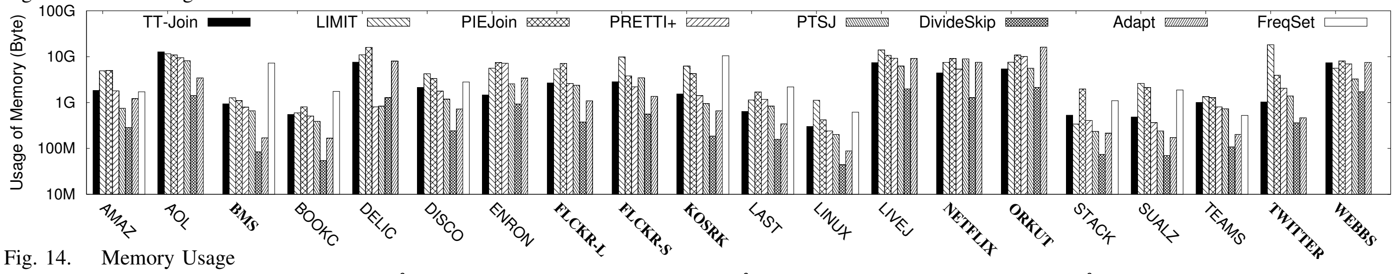


Fig. 14. Memory Usage

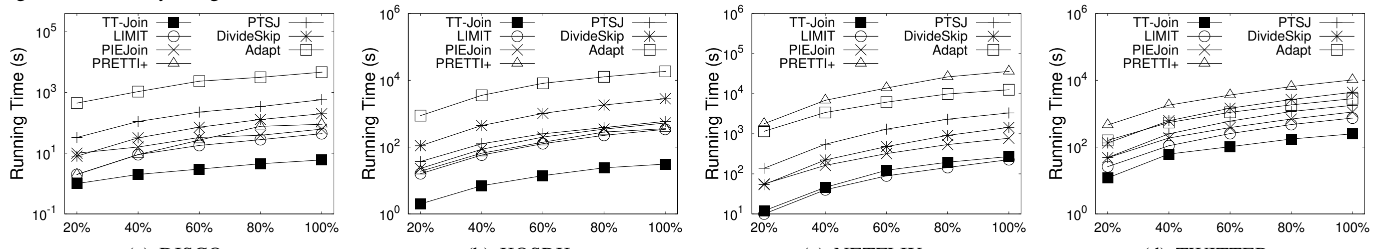

(a) DISCO (b) KOSRK (c) NETFLIX (d) TWITTER

Fig. 15. Vary # records

except on LINUX and SUAIZ. The performance of PIEJoin was quite stable on all datasets, but it was always suppressed by LIMIT except on LINUX and SUAIZ. The processing time of PRETTI+ and PTSJ was quite sensitive to the record length [18]. It was observed that PRETTI+ favored datasets with small record size, such as AMAZ, DISCO, LINUX, SUAILZ, and TEAMS. However, PRETTI+ was extremely slow on datasets with relatively large record size, such as DELIC, ENRON, LIVEJ, NETFLIX, and TWITTER. It took more than 10 hours on NETFLIX in which the average record size is 209. As reported in [18], PTSJ, on the other hand, cannot efficiently handle datasets with small record size. For example, it took hours for PTSJ to process AOL, while *TT-Join* spent less than 2 minutes. Generally, PTSJ had the worst overall performance. The reasons are two-fold. First, PTSJ is a bitmap-signature based method, which is data-independent and does not make use of the distribution of the elements. Second, it considers records in $\mathcal{S}$ individually, which means there is no computation share between records, even for identical records. The results showed that DivideSkip significantly outperformed other two adapted algorithms. Interestingly, DivideSkip even beat two state-of-the-art algorithms PRETTI+ and PTSJ on several datasets, such as AOL, DELIC, ENRON, FLICKR-L, LIVEJ, and ORKUT. The reason was that DivideSkip used the same index strategy as PRETTI+, but DivideSkip can take advantage of the careful processing of long and short inverted lists in different ways. It was reported that Adapt and FreqSet were not competitive under all datasets. In particular, FreqSet failed to return results on half of the 20 datasets (we set allowed running time to be 10 hours).

As reported in Fig. 13, *TT-Join* had the best overall performance on 20 real-life datasets and significantly outperformed other competitors on the majority of the datasets. This was because *TT-Join* not only enhanced the nice properties of the *union-oriented* approach, e.g., exploited the skewness of the data and had less number of records explored, but also can directly validate a significant number of pairs, which were verification free. In particular, *TT-Join* beat other algorithms by at least around one order of magnitude on datasets with large z-values, such as DISCO, KOSRK, LINUX, SUALZ, and TWITTER. This was because *union-oriented TT-Join* can effectively exploit the skewness of the data distribution. It was very interesting that *TT-Join* can also significantly outperform other competitors on ORKUT and WEBBS although they were not skewed, with z values 0.13 and 0.04, respectively. For instance, *TT-Join* outperformed other algorithms on WEBBS by one order of magnitude. We observed that there were a large number of distinct elements in ORKUT and WEBBS, and the average size of the records was large, which favored the least frequent element based signature technique. For some datasets with moderate or small z-value, such as AMAZ, LAST, and TEAMS, *TT-Join* can also achieve a superior performance, at least 2 times faster than the second ranked algorithm. The reason was that the *kLFP-Tree* enabled us to increase the filtering capacity with small overhead and validate a significant number of join results without explicitly invoking the verification during the join processing. NETFLIX was the only dataset in which *TT-Join* was slightly outperformed by other competitors. We observed that it was not skewed ($z = 0.33$) and the number of distinct element ($|\mathcal{E}| = 17,770$) was small compared to the dataset size ($n = 480,189$ and $m = 209$), both of which were not in favour of *TT-Join*.

Memory Usage. Fig. 14 reported the memory usage of 8 algorithms. Same as [18], the used memory was measured by the difference between the total memory and free memory of JVM after indexes were constructed for algorithms implemented in Java. For algorithms implemented in C++, we measured the maximum amount of used memonry. It was observed that, DivideSkip consumed the smallest amount of memory under all datasets. Under most of the datasets, PTSJ and Adapt consumed the second smallest amount of memory because PTSJ only built Patricia trie index on $\mathcal{R}$ while Adapt only built inverted list on $\mathcal{S}$. They were followed by *TT-Join* and PRETTI+. The memory usage of LIMIT and PIEJoin were similar and relatively larger than that of the other algorithms. This was because both of them use complicated index structures. Particularly, besides the prefix trees on both $\mathcal{R}$ and $\mathcal{S}$, PIEJoin also needed some auxiliary data structures to speed up the join processing.

Scalability Evaluation. In the last set of experiments, we evaluated the scalability of 7 algorithms on 4 representative datasets. FreqSet was excluded from the evaluation because it failed to give response within allowed time on most of the experimental settings. For each dataset, we randomly sampled 20%, 40%, 60%, 80%, 100% of records from the original dataset, and conducted experiment on each sampled dataset. Fig. 15 showed that the running time of the algorithms grew steadily as the number of records increasesd on all datasets, and the performance ranks of the algorithms remained the same under most of the settings.

VI. Conclusion

In this paper, we study the problem of set containment join. Several in-memory set containment join algorithms have been developed in the literature. Based on the computing paradigms, we classify them into two categories, namely *intersection-oriented* and *union-oriented* methods. Through a comprehensive analysis, we show the advantages and limits of the algorithms in each category. Then we propose a new *union-oriented* method, namely *TT-Join*, which can take advantage of both *union-oriented* and *intersection-oriented* approaches. Extensive experiments on 20 real-life set-valued datasets from a variety of applications demonstrate the superior performance of *TT-Join* compared with the state-of-the-art techniques.

Acknowledgment

Wenjie Zhang is supported by ARC DP150103071 and DP150102728. Ying Zhang is supported by ARC DE140100679 and DP170103710. Xuemin Lin is supported by NSFC61232006, ARC DP150102728, DP140103578 and DP170101628.

References

[1] T. W. Yan and H. García-Molina, "Index structures for selective dissemination of information under the boolean model," *TODS*, vol. 19, no. 2, pp. 332–364, 1994.

[2] C. Li, J. Lu, and Y. Lu, "Efficient merging and filtering algorithms for approximate string searches," in *ICDE*, 2008, pp. 257–266.

[3] P. Agrawal, A. Arasu, and R. Kaushik, "On indexing error-tolerant set containment," in *SIGMOD*, 2010, pp. 927–938.

[4] Z. Hmedeh, H. Kourdounakis, V. Christophides, C. Du Mouza, M. Scholl, and N. Travers, "Subscription indexes for web syndication systems," in *EDBT*, 2012, pp. 312–323.

[5] M. Terrovitis, P. Bouros, P. Vassiliadis, T. Sellis, and N. Mamoulis, "Efficient answering of set containment queries for skewed item distributions," in *EDBT*, 2011, pp. 225–236.

[6] M. Terrovitis, S. Passas, P. Vassiliadis, and T. Sellis, "A combination of trie-trees and inverted files for the indexing of set-valued attributes," in *CIKM*, 2006, pp. 728–737.

[7] E. Zhu, F. Nargesian, K. Q. Pu, and R. J. Miller, "Lsh ensemble: Internet scale domain search," in *VLDB*, 2016, pp. 1185–1196.

[8] A. Arasu, V. Ganti, and R. Kaushik, "Efficient exact set-similarity joins," in *VLDB*, 2006, pp. 918–929.

[9] R. J. Bayardo, Y. Ma, and R. Srikant, "Scaling up all pairs similarity search," in *WWW*, 2007, pp. 131–140.

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

[11] C. Xiao, W. Wang, X. Lin, and J. X. Yu, "Efficient similarity joins for near duplicate detection," in *WWW*, 2008, pp. 131–140.

[12] C. Xiao, W. Wang, X. Lin, and H. Shang, "Top-k set similarity joins," in *ICDE*, 2009, pp. 916–927.

[13] J. Wang, G. Li, and J. Feng, "Can we beat the prefix filtering?: an adaptive framework for similarity join and search," in *SIGMOD*, 2012, pp. 85–96.

[14] D. Deng, G. Li, H. Wen, and J. Feng, "An efficient partition based method for exact set similarity joins," in *VLDB*, 2015, pp. 360–371.

[15] W. Mann, N. Augsten, and P. Bouros, "An empirical evaluation of set similarity join techniques," in *VLDB*, 2016, pp. 636–647.

[16] N. Mamoulis, "Efficient processing of joins on set-valued attributes," in *SIGMOD*, 2003, pp. 157–168.

[17] R. Jampani and V. Pudi, "Using prefix-trees for efficiently computing set joins," in *DASFAA*, 2005, pp. 761–772.

[18] Y. Luo, G. H. Fletcher, J. Hidders, and P. De Bra, "Efficient and scalable trie-based algorithms for computing set containment relations," in *ICDE*, 2015, pp. 303–314.

[19] P. Bouros, N. Mamoulis, S. Ge, and M. Terrovitis, "Set containment join revisited," *Knowledge and Information Systems*, pp. 1–28, 2015.

[20] A. Kunkel, A. Rheinländer, C. Schiefer, S. Helmer, P. Bouros+3, and U. Leser, "Piejoin: Towards parallel set containment joins," in *SSDBM*, 2016, p. 11.

[21] S. Helmer and G. Moerkotte, "Evaluation of main memory join algorithms for joins with set comparison predicates," in *VLDB*, 1997, pp. 386–395.

[22] K. Ramasamy, J. M. Patel, J. F. Naughton, and R. Kaushik, "Set containment joins: The good, the bad and the ugly." in *VLDB*, 2000, pp. 351–362.

[23] S. Melnik and H. Garcia-Molina, "Divide-and-conquer algorithm for computing set containment joins," in *EDBT*, 2002, pp. 427–444.

[24] S. Melnik and H. Garcia Molina, "Adaptive algorithms for set containment joins," *TODS*, vol. 28, no. 1, pp. 56–99, 2003.

[25] http://liu.cs.uic.edu/download/data/.

[26] http://www.cim.mcgill.ca/~dudek/206/Logs/AOL-user-ct-collection.

[27] http://www.informatik.uni-freiburg.de/~cziegler/BX/.

[28] http://dai-labor.de/IRML/datasets.

[29] http://www.discogs.com/.

[30] http://www.cs.cmu.edu/~enron.

[31] http://www.dtic.upf.edu/~ocelma/MusicRecommendationDataset/lastfm-1K.html.

[32] http://konect.uni-koblenz.de/networks/lkml_person-thread.

[33] http://socialnetworks.mpi-sws.org/data-imc2007.html.

[34] http://www.clearbits.net/torrents/1881-dec-2011.

[35] http://vi.sualize.us/.

[36] http://wiki.dbpedia.org/Downloads.

[37] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," in *SIGMOD*, 2000, pp. 1–12.