2019 IEEE 35th International Conference on Data Engineering (ICDE)

# LCJoin: Set Containment Join via List Crosscutting

Dong Deng, Chengcheng Yang*, Shuo Shang*, Fan Zhu, Li Liu, Ling Shao
*Inception Institute of Artificial Intelligence*
Abu Dhabi, UAE
{dong.deng, chengcheng.yang, shuo.shang, fan.zhu, li.liu, ling.shao}@inceptioniai.org

***Abstract*—A set containment join operates on two set-valued attributes with a subset ($\subseteq$) relationship as the join condition. It has many real world applications, such as in publish/subscribe services and inclusion dependency discovery. Existing solutions can be broadly classified into union-oriented and intersection-oriented methods. Based on several recent studies, union-oriented methods are not competitive as they involve an expensive subset enumeration step. Intersection-oriented methods build an inverted index on one attribute and perform inverted list intersection on another attribute. Existing intersection-oriented methods intersect inverted lists one-by-one. In contrast, in this paper, we propose to intersect all the inverted lists simultaneously while skipping many irrelevant entries in the lists. To share computation, we utilize the prefix tree structure and extend our novel list intersection method to operate on the prefix tree. To further improve the efficiency, we propose to partition the data and use different methods to process each partition. We evaluated our methods using both real-world and synthetic datasets. Experimental results show that our approach outperforms existing methods by up to 10×.**

## I. Introduction

Set containment is an important relationship between two sets, which indicates that one set is a subset of another. It has numerous real world applications. For example, if the skills mastered by a worker and those required for a job are modeled as sets, the set containment relationship indicates whether or not a worker is competent in a job. As another example, if the keywords subscribed to by a user and the words in an article are modeled as the sets, then the set containment determines if an article aligns with the users interests and should be suggested to them. Additionally, set containment is also relevant for inclusion dependency. Specifically, if two columns of values are modeled as sets, then set containment can be used to determine if there is an inclusion dependency between them. In this paper, we study the set containment join problem, which, given two collections $\mathbb{R}$ and $\mathbb{S}$ of sets, finds all the set pairs $(\mathsf{R}, \mathsf{S})$ with a set containment relationship, i.e., $\mathsf{R} \subseteq \mathsf{S}$. Since the volume of data is increasingly large, we focus on improving the efficiency and scalability of this operation.

There are many existing works that focus on set containment joins. Based on one recent study [25], existing methods can be broadly classified into union-oriented methods [9], [13], [15], [16], [18], [24], [25] and intersection-oriented methods [3], [10], [11], [13], [14].

*Chengcheng Yang and Shuo Shang are the corresponding authors.

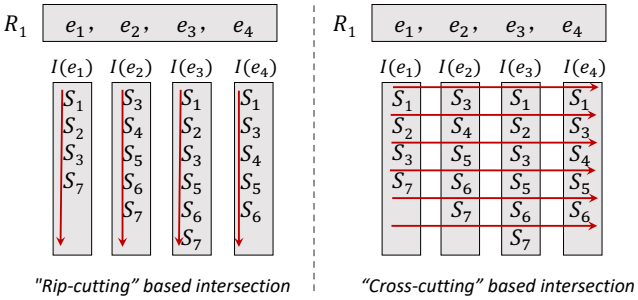



Fig. 1. Two ways to intersect inverted lists: "cross-cutting" and "rip-cutting".

**Union-oriented Methods.** Union-oriented methods first generate a signature for each set. The signature guarantees that $\mathsf{R} \subseteq \mathsf{S}$ only if $Sig(\mathsf{R}) \subseteq Sig(\mathsf{S})$, where $Sig(\mathsf{R})$ and $Sig(\mathsf{S})$ are the signatures of $\mathsf{R}$ and $\mathsf{S}$, respectively. Next, all subsets in $Sig(\mathsf{S})$ are enumerated and any $Sig(\mathsf{R})$ that is identical to any of these subsets is retrieved. Based on the guarantee provided by the signatures, $(\mathsf{R}, \mathsf{S})$ is a candidate pair. Finally, the candidates are verified and the join results obtained. Based on several recent studies [3], [24], union-oriented methods cannot compete with intersection-oriented methods. This is because a large signature size will lead to an expensive subset enumeration cost, which grows exponentially with the increasing of the signature size, while a small signature will result in many candidates and a high verification cost.

As an example, Helmer et al. [9] use a bitmap of size $b$ as a signature. To generate the signature bitmap for a set, they map each element in the set to a number $i$ between 1 and $b$ and set the $i^{th}$ bit of its signature bitmap to 1. Obviously, for any two sets $\mathsf{R}$ and $\mathsf{S}$, $\mathsf{R} \subseteq \mathsf{S}$ only if every 1 bit in $Sig(\mathsf{R})$ is also set to 1 in $Sig(\mathsf{S})$, which we denote as $Sig(\mathsf{R}) \subseteq Sig(\mathsf{S})$. In this case, union-oriented methods need to enumerate $2^b$ bitmaps for each set from $\mathbb{S}$, which is highly inefficient.

**Intersection-oriented Methods.** Intersection-oriented methods first build an inverted index $\mathcal{I}$ for $\mathbb{S}$, where the inverted list $\mathcal{I}[e]$ consists of all the sets in $\mathbb{S}$ containing the element $e$ (the sets are sorted). Then, for each set $\mathsf{R}$ in $\mathbb{R}$, they intersect all the inverted lists corresponding to the elements in $\mathsf{R}$ and obtain a list of sets from $\mathbb{S}$. For each $\mathsf{S}$ in the list, $\mathsf{R}$ is a subset of $\mathsf{S}$ and $(\mathsf{R}, \mathsf{S})$ is a result.

In this paper, we propose an intersection-oriented method. However, we intersect the inverted lists in a new way. All existing methods intersect the inverted lists one by one in

2375-026X/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDE.2019.00040

a "*rip-cutting*" fashion. For example, consider the set $\mathsf{R}_1 = \{e_1, e_2, e_3, e_4\}$ and its four inverted lists as shown on the left of Fig. 1. Existing methods first intersect $\mathcal{I}[e_1]$ with $\mathcal{I}[e_2]$ and get $\mathcal{L}_1 = \{\mathsf{S}_3, \mathsf{S}_7\}$. Then they intersect $\mathcal{L}_1$ with $\mathcal{I}[e_3]$ and get $\mathcal{L}_2 = \{\mathsf{S}_3, \mathsf{S}_7\}$. Finally, they intersect $\mathcal{L}_2$ with $\mathcal{I}[e_4]$, get the list $\mathcal{L}_3 = \{\mathsf{S}_3\}$, and generate a result $(\mathsf{R}_1, \mathsf{S}_3)$.

In our case, however, we intersect the inverted list in a "*cross-cutting*" fashion as shown on right of Fig. 1 (details of which will be provided in Section III). A huge advantage of cross-cutting based intersection is that it can use the "gap" between two consecutive entries in an inverted list to skip irrelevant entries in the other lists. For example, consider the two consecutive entries $\mathsf{S}_3$ and $\mathsf{S}_7$ in $\mathcal{I}[e_1]$ in Fig. 1. As $\mathsf{S}_4$, $\mathsf{S}_5$, and $\mathsf{S}_6$ are not in $\mathcal{I}[e_1]$, $e_1$ does not exist in the three sets and $\mathsf{R}_1$ cannot be a subset of $\mathsf{S}_4$, $\mathsf{S}_5$, or $\mathsf{S}_6$. Thus we can skip the three sets in all the other inverted lists, i.e., $\mathcal{I}[e_2]$, $\mathcal{I}[e_3]$, and $\mathcal{I}[e_4]$. To share the computation between the sets in $\mathbb{R}$, we propose to build a tree on $\mathbb{R}$ and extend the cross-cutting based intersection to operate on the tree. To further improve the performance and scalability of our method, we propose to partition the data and process each partition separately.

In summary, we make the following contributions in this paper:

- We develop a novel, intersection-oriented approach for set containment joins. Our approaches can skip irrelevant entries in the inverted lists when intersecting them.
- We design a tree-based method to share the computation between sets in $\mathbb{R}$. We propose an early termination technique for the tree-based method.
- We propose to partition the data to further improve the efficiency and scalability of our method.
- We conduct extensive experiments on both real-world datasets and synthetic datasets. The experimental results show that our approach outperform current state-of-the-art methods by up to one order of magnitude.

The rest of the paper is organized as follows. Section II defines the problem. Our set containment join framework is presented in Section III. We discuss the tree-based methods in Section IV and data partition in Section V. Section VI provides experimental results. We review related work in Section VII and conclude in Section VIII.

## II. Problem Definition

Given two collections of sets, the set containment join problem aims to find all the set pairs from the two collections in which one set is a subset of the other. A formal definition is provided below.

*Definition 1 (Set Containment Join):* Given two collections $\mathbb{R}$ and $\mathbb{S}$ of sets, the set containment join $\mathbb{R} \bowtie_{\subseteq} \mathbb{S}$ finds all pairs $(\mathsf{R}, \mathsf{S})$ such that $\mathsf{R} \subseteq \mathsf{S}$, where $\mathsf{R}$ and $\mathsf{S}$ are two sets in $\mathbb{R}$ and $\mathbb{S}$, respectively. That is $\mathbb{R} \bowtie_{\subseteq} \mathbb{S} = \{(\mathsf{R}, \mathsf{S}) | \mathsf{R} \subseteq \mathsf{S}, \mathsf{R} \in \mathbb{R}, \mathsf{S} \in \mathbb{S}\}$.

*Example 1: For example, consider the two collections $\mathbb{R}$ and $\mathbb{S}$ of sets in Table I. Each set $\mathsf{R} \in \mathbb{R}$ (or $\mathsf{S} \in \mathbb{S}$) is associated with an identifier Rid (or Sid). The set containment join $\mathbb{R} \bowtie_{\subseteq} \mathbb{S}$ will result in two pairs $(\mathsf{R}_1, \mathsf{S}_3)$ and $(\mathsf{R}_2, \mathsf{S}_5)$, where the first sets are subsets of the second ones. For all the other 19 pairs, there is no subset relationship.*

TABLE I
Running Example

| Rid | R |
|---|---|
| $\mathsf{R}_1$ | $\{e_1, e_2, e_3, e_4\}$ |
| $\mathsf{R}_2$ | $\{e_2, e_3, e_5\}$ |
| $\mathsf{R}_3$ | $\{e_1, e_2, e_5, e_6\}$ |

(a) dataset $\mathbb{R}$

| Sid | S |
|---|---|
| $\mathsf{S}_1$ | $\{e_1, e_3, e_4, e_5, e_6\}$ |
| $\mathsf{S}_2$ | $\{e_1, e_3, e_5\}$ |
| $\mathsf{S}_3$ | $\{e_1, e_2, e_3, e_4, e_6\}$ |
| $\mathsf{S}_4$ | $\{e_2, e_4, e_5, e_6\}$ |
| $\mathsf{S}_5$ | $\{e_2, e_3, e_4, e_5, e_6\}$ |
| $\mathsf{S}_6$ | $\{e_2, e_3, e_4, e_6\}$ |
| $\mathsf{S}_7$ | $\{e_1, e_2, e_3, e_6\}$ |

(b) dataset $\mathbb{S}$

## III. The Framework

In this section, we present our set containment join framework. The framework first builds an inverted index for the sets in $\mathbb{S}$ (Section III-A). Then, it calculates the set containment join using the inverted index previously built (Section III-B).

### A. Inverted Index Construction

We build an inverted index $\mathcal{I}$ for $\mathbb{S}$. The headers of the inverted lists in $\mathcal{I}$ are the distinct elements in $\mathbb{S}$. For each distinct element $e$ in $\mathbb{S}$, its corresponding inverted list $\mathcal{I}[e]$ consists of the identifiers Sid of the sets S containing $e$, i.e., $e \in \mathsf{S}$. For ease of presentation, hereinafter, we use the set and its identifier interchangeably. Note that the identifiers in the inverted lists are ordered by their subscripts in ascending order. For example, Fig. 2 shows the inverted index $\mathcal{I}$ constructed for the sets in $\mathbb{S}$ in Table I(b). The inverted index can be constructed by sequentially reading the sets in $\mathbb{S}$ and, for each element $e$ in $\mathsf{S}_i$, appending $\mathsf{S}_i$ to the end of $\mathcal{I}[e]$.

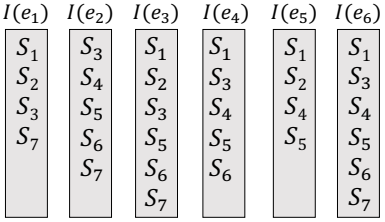



Fig. 2. The inverted index $\mathcal{I}$ for $\mathbb{S}$ in Table I (b).

### B. All Pair Set Containment Search

After the inverted index is constructed, we use it to find all the set pairs with a subset relationship. At a high level, for each set $\mathsf{R} \in \mathbb{R}$, intersection-oriented methods retrieve all the inverted lists corresponding to the elements in R and intersect them. The intersection result is a list $\mathcal{L}$ of sets in $\mathbb{S}$, which are all supersets of R. Thus, we produce a result $(\mathsf{R}, \mathsf{S})$ for each set $\mathsf{S} \in \mathcal{L}$.

For this purpose, all existing approaches intersect the inverted lists one by one. We call this "*rip-cutting*" based

intersection since all entries in a list are processed at the same time. In contrast, in our framework, we employ a "*cross-cutting*" based intersection. The basic idea is that we first check each inverted list in R to see whether all of them contain a "*specific set*" $\mathsf{S}_i$. If so, $\mathsf{R} \subseteq \mathsf{S}_i$ and we produce a result $(\mathsf{R}, \mathsf{S}_i)$. Then, for any inverted list $\mathcal{I}[e]$ in R, let $\mathsf{S}_j$ be the first entry greater than $\mathsf{S}_i$ in the list. Obviously, the element $e$ must not exist in any $\mathsf{S}_k$ where $i < k < j$ and $\mathsf{R} \not\subseteq \mathsf{S}_k$. Thus we can skip all the entries in the "*gap*" (i.e., $\mathsf{S}_{i+1}, \cdots, \mathsf{S}_{j-1}$) in all the other inverted lists. To this end, we use $\mathsf{S}_j$ as the new specific set to check and repeat the above process until we reach the end of any inverted list.

Note that we can use the first entry $\mathsf{S}_j$ greater than $\mathsf{S}_i$ in any inverted list in R to skip the irrelevant entries. To reach the end of the inverted lists as soon as possible, we propose to use the largest entry as the next specific set to check, because it will have the largest gap, enabling us to skip more entries in the other inverted lists. In addition, we initially use the smallest set $\mathsf{S}_1$ as the first specific set to check.

*Example 2: For example, consider the two datasets $\mathbb{R}$ and $\mathbb{S}$ in Table I. For $\mathsf{R}_1$, there are four inverted lists $\mathcal{I}[e_1]$, $\mathcal{I}[e_2]$, $\mathcal{I}[e_3]$, $\mathcal{I}[e_4]$, as shown on the right of Fig. 1. Our framework first initializes the specific set to check to be $\mathsf{S}_1$. Since $\mathsf{S}_1$ is not found in $\mathcal{I}[e_2]$, it cannot be a superset of $\mathsf{R}_1$. The first entries in $\mathcal{I}[e_1]$, $\mathcal{I}[e_2]$, $\mathcal{I}[e_3]$, and $\mathcal{I}[e_4]$ greater than $\mathsf{S}_1$ are $\mathsf{S}_2, \mathsf{S}_3, \mathsf{S}_2$, and $\mathsf{S}_3$, respectively. We use the largest one, $\mathsf{S}_3$, as the next specific set to check. Since $\mathsf{S}_3$ exists in all the lists, our framework generates a result $(\mathsf{R}_1, \mathsf{S}_3)$. We repeat this process, with the next specific set to check being $\mathsf{S}_7$. However, $\mathsf{S}_7$ is larger than all the entries in $\mathcal{I}[e_4]$. Thus our framework reaches the end of $\mathcal{I}[e_4]$ and terminates.*

**Correctness and soundness.** The framework is correct and sound, i.e., the set pairs found by the framework all have the set containment relationship and all the set containment pairs can be found by the framework. The correctness is obvious as the framework returns a pair only if the specific set is found in all the inverted lists of R, which indicates a set containment relationship. The framework is also sound. For any set pair $(\mathsf{R}, \mathsf{S})$ in $\mathbb{R} \times \mathbb{S}$ where $\mathsf{R} \subseteq \mathsf{S}$, all the inverted lists of R must have the entry S. Since in the framework an entry is skipped only if it does not exist in at least one of the inverted lists in R, S cannot be skipped and must be a specific set to check in the framework. Once S is checked, the framework must find it in all the inverted lists of R and return the pair $(\mathsf{R}, \mathsf{S})$.

The pseudo-code of our framework is shown in Algorithm 1. It takes two collections $\mathbb{R}$ and $\mathbb{S}$ of sets as input, and outputs their set containment join result $\mathcal{A} = \mathbb{R} \bowtie_{\subseteq} \mathbb{S}$. To do so, it first builds an inverted index $\mathcal{I}$ for the sets in $\mathbb{S}$ (Line 2). Then for each set $\mathsf{R} \in \mathbb{R}$, it initializes the specific set MaxSid to check to be $\mathsf{S}_1$ (Line 4). Next our framework binary searches for MaxSid on each inverted list in R (Line 6). If MaxSid is found in all the lists, it adds the pair $(\mathsf{R}, \mathsf{S})$ to the result $\mathcal{A}$ (Lines 7-8). Then, the framework identifies the first entry in each inverted list in R that is greater than MaxSid and use the largest one among them as the next specific set to check (Lines 9-10). These steps are repeated until the end of an inverted list is reached (Line 5). Finally, the result $\mathcal{A}$ is returned (Line 11).

**Algorithm 1**: THE CROSS-CUTTING FRAMEWORK

```
   Input: 𝕊 and ℝ: two collections of sets.
   Output: 𝒜: ℝ ⋈⊆ 𝕊 = {(R, S)|R ⊆ S, R ∈ ℝ, S ∈ 𝕊};
1  begin
2      Build an inverted index ℐ for 𝕊;
3      foreach R ∈ ℝ with identifier Rid do
4          MaxSid = 1;
5          while not reaching the end of any inverted list do
6              binary search for MaxSid on the inverted
               lists corresponding to the elements in R;
7              if MaxSid is found on all the lists then
8                  add the pair (Rid, MaxSid) to 𝒜;
9              find the first entry greater than MaxSid in
               each inverted list in R and let NextMax be
               the largest one among them;
10             update MaxSid as NextMax;
11     return 𝒜;
12 end
```

**Cost analysis.** The cost for building the inverted index is $\Sigma_{\mathsf{S}\in\mathbb{S}}|\mathsf{S}|$. Suppose on average our framework checks $x$ specific sets for each set in $\mathbb{R}$. Then the cost for binary searching for the specific sets is around $x\Sigma_{\mathsf{R}\in\mathbb{R}}\Sigma_{e\in\mathsf{R}}\log|\mathcal{I}[e]|$ and the cost for setting the next specific sets to check and producing results is $x\Sigma_{\mathsf{R}\in\mathbb{R}}|\mathsf{R}|$, where $|\mathcal{I}[e]|$ is the inverted list length and $|\mathsf{R}|$ is the set size. In total, the cost is

$$\Sigma_{\mathsf{S}\in\mathbb{S}}|\mathsf{S}| + x\Sigma_{\mathsf{R}\in\mathbb{R}}(|\mathsf{R}| + \Sigma_{e\in\mathsf{R}}\log|\mathcal{I}[e]|).$$

### C. Early Termination

In each round, our framework binary searches for a specific set MaxSid in all the inverted lists in R. We observe that we can terminate the binary searches earlier in each round. More specifically, whenever MaxSid is not found in an inverted list $\mathcal{I}[e]$ in R, we do not need to check if MaxSid is in the other inverted lists in R. This is because $e \notin$ MaxSid and MaxSid cannot be a superset of R.

Instead of using the largest gap (i.e., the first entry greater than MaxSid) in all the inverted lists in R as the new specific set NextMax to check in the next round, we set NextMax as the largest gap in all the visited inverted lists in the current round. This is because the binary searches are skipped on the unvisited lists and their gaps are unknown. In addition, we propose to visit the inverted lists in ascending order of length. This is because the short inverted lists potentially have larger "gaps" and we can skip more entries in each round.

*Example 3: In the previous example, our framework binary searches for $\mathsf{S}_1$, $\mathsf{S}_3$, and $\mathsf{S}_7$ in the four inverted lists. In total, our framework performs 12 binary searches. By employing the*

*early termination technique, we visit the lists in the order of $\mathcal{I}[e_1]$, $\mathcal{I}[e_2]$, $\mathcal{I}[e_4]$, and $\mathcal{I}[e_3]$. As the first specific set $\mathsf{S}_1$ is not found in $\mathcal{I}[e_2]$, we stop the current round and set the next specific set to check to be the larger one between $\mathsf{S}_2$ from $\mathcal{I}[e_1]$ and $\mathsf{S}_3$ from $\mathcal{I}[e_2]$, which is $\mathsf{S}_3$. As $\mathsf{S}_3$ is found in all the inverted lists, we produce a result $(\mathsf{R}_1, \mathsf{S}_3)$. Then the next specific set to check is $\mathsf{S}_7$. We terminate after binary searching $\mathsf{S}_7$ on $\mathcal{I}[e_4]$ as we reach the end of $\mathcal{I}[e_4]$. In total, the early termination only performs 9 binary searches.*

The early termination may not find the largest specific set to check in each round. Nevertheless, it can save some unnecessary binary search operations if the specific set MaxSid is not a superset of R, especially when the set size $|\mathsf{R}|$ is large.

## IV. The Tree-based Method

This section discusses how to share the computation on the sets in $\mathbb{R}$. We first introduce the prefix tree index in Section IV-A and then present the tree-based method in Section IV-B. The tree-based method is essentially traversing the tree in postorder. Finally, we integrate the early termination technique into the tree-based method in Section IV-C.

### A. The Prefix Tree

We build a prefix tree $\mathcal{T}$ for the sets in $\mathbb{R}$, where each tree node $n$ is associated with an element $n.e$. For this purpose, we first sort the elements in each set $\mathsf{R} \in \mathbb{R}$ in a global order. Then we sequentially insert the elements in R into $\mathcal{T}$. Each set $\mathsf{R} \in \mathbb{R}$ corresponds to a unique leaf node in $\mathcal{T}$ where the elements associated with the nodes on the path from the root node $\mathcal{T}.root$ to this leaf node are exactly the elements in R, sorted in the global order. For example, Fig. 3 shows the prefix tree for the three sets in $\mathbb{R}$ in Table I(a). Note, in this paper, as an example, we use an increasing order of subscripts as the global order for the elements. However, to share more computation, in our implementation we use a decreasing order of frequency as the global order of the elements. In addition, to save memory usage, we can replace the prefix tree with the Patricia tree (*a.k.a.* radix trie), where the inner nodes containing only one child are merged. All our techniques proposed in this paper apply to this more compact tree structure.

For ease of presentation, hereinafter we use the set R and its corresponding leaf node in $\mathcal{T}$ interchangeably. We also use the node $n$ and its corresponding inverted list $\mathcal{I}[n.e]$ interchangeably. For instance, a set S exists in a node $n$ really means S exists in the corresponding inverted list $\mathcal{I}[n.e]$ of $n$.

### B. Set Containment Join via Postorder Tree Traversing

In this section, we discuss how to find all the set containment pairs using the inverted index $\mathcal{I}$ and the prefix tree $\mathcal{T}$. We first give the high level idea.

As discussed in the framework, for each set R in $\mathbb{R}$, we maintain a specific set MaxSid and check whether MaxSid exists in every inverted list in R. Since each set $\mathsf{R} \in \mathbb{R}$ corresponds to a leaf node in the prefix tree $\mathcal{T}$, we propose to keep the specific set to check for each set in its corresponding

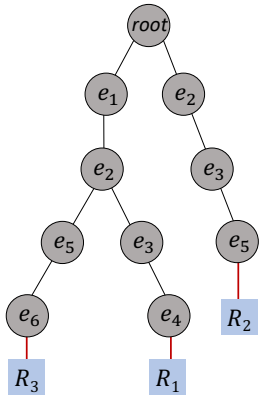



Fig. 3. The tree structure for $\mathbb{R}$ in Table I (a).

leaf node $n$, denoted as $n$.MaxSid. For the inner node $n$, we use $n$.MaxSid to keep the smallest specific set to check among all the leaf nodes in the subtree rooted at $n$ so as to. As we will see later in this section, this helps us update the specific sets in the leaf nodes in a new round.

The next step in the framework is to produce a result if the specific set exists in all the inverted lists of R. To achieve this on the tree, for each node $n$ in the tree, we use $n$.RidList to keep the list of leaf nodes where i) the specific set of the leaf node is $n$.MaxSid and ii) $n$.MaxSid exists in all the nodes on the path from $n$ to this leaf node (recall that a set S exists in a node $n$ really means S exists in the inverted list $\mathcal{I}[n.e]$). Based on this definition, all the sets in $\mathcal{T}$.root.RidList are subsets of $\mathcal{T}$.root.MaxSid, where $\mathcal{T}$.root is the root node of $\mathcal{T}$. Thus, for each set $\mathsf{R} \in \mathcal{T}$.root.RidList, we produce a result $(\mathsf{R}, \mathcal{T}$.root.MaxSid$)$.

The last step is to update the specific sets to check on the leaf nodes in next round. Intuitively, for each leaf node $n$, we can go through all the ancestor nodes of $n$ and update the new specific set of $n$ to be the largest gap of its ancestor nodes and itself (recall the gap of a node $v$ is the first entry greater than the current specific set $n$.MaxSid in the inverted list $\mathcal{I}[v.e]$).

*Example 4: Figure 4 shows a running example based on the two datasets in Table I using the idea above. For each node $n_i$ in the tree, we show its two variables $n_i$.MaxSid and $n_i$.RidList. At the beginning (as shown in Fig. 4(a)), the initial specific sets for all the leaf nodes $n_5$, $n_7$, and $n_{10}$ are $\mathsf{S}_1$. For the inner nodes, based on the definition, their specific sets are also $\mathsf{S}_1$. We also have $n_4$.RidList $= \{\mathsf{R}_3\}$. This is because i) the corresponding leaf node $n_5$ of $\mathsf{R}_3$ has the same specific set as $n_4$ (i.e., $n_5$.MaxSid $= n_4$.MaxSid $= \mathsf{S}_1$) and ii) the specific set $n_4$.MaxSid $= \mathsf{S}_1$ exists in both inverted lists of $n_4$ and $n_5$, i.e., $\mathcal{I}[e_5]$ and $\mathcal{I}[e_6]$, as shown in Figure 2. Similarly, we have $n_5$.RidList $= \{\mathsf{R}_3\}$, $n_6$.RidList $= \{\mathsf{R}_1\}$, etc. Note that $n_3$.RidList $= \phi$ as its specific set $n_3$.MaxSid $= \mathsf{S}_1$ does not exist in the inverted list $\mathcal{I}[e_2]$ of $n_3$.*

*In the second round, as shown in Fig. 4(b), we update the specific sets in the leaf nodes. For the leaf node $n_5$, as the gaps in itself and its ancestor nodes $n_4$, $n_3$, and $n_2$ are respectively*

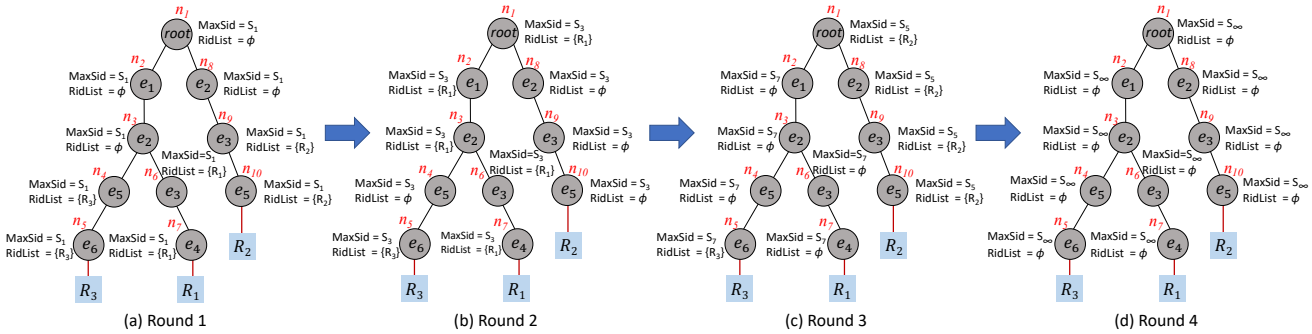



Fig. 4. A running example of the tree-based method based on the datasets in Table I.

$\mathsf{S}_3$, $\mathsf{S}_2$, $\mathsf{S}_3$, *and* $\mathsf{S}_2$, *we update its specific set* $n_5$.*MaxSid as the largest one* $\mathsf{S}_3$. *Similarly we can update the specific sets in the other nodes, which are all set to* $\mathsf{S}_3$. *We also have* $n_1$.*RidList* $= \{\mathsf{R}_1\}$ *as i) the corresponding leaf node* $n_7$ *of* $\mathsf{R}_1$ *has the same specific set as* $n_1$ *and ii) the specific set* $n_1$.*MaxSid* $= \mathsf{S}_3$ *exists in all the inverted lists* $\mathcal{I}[e_1]$, $\mathcal{I}[e_2]$, $\mathcal{I}[e_3]$, *and* $\mathcal{I}[e_4]$ *of* $n_2$, $n_3$, $n_6$, *and* $n_7$. *Thus, we produce a result* $(\mathsf{R}_1, \mathsf{S}_3)$. *Similarly, in the next round as shown in Fig. 4(c), we find another result* $(\mathsf{R}_2, \mathsf{S}_5)$. *In the last round, as shown in Fig. 4(d), the specific set for the root node is* $n_1$.*MaxSid* $= \mathsf{S}_\infty$, *which indicates that we have reached the ends of the inverted lists for all leaf nodes, and we terminate.*

**Postorder Tree Traversing.** To implement the above high-level idea, we design a postorder tree traversing method. In traversing the tree, $n$.MaxSid and $n$.RidList will get updated for every node $n$ in the tree.

More specifically, consider an inner node $n$. Suppose for every child node $c$ of $n$, $c$.MaxSid and $c$.RidList have been updated in the postorder tree traversing. We first discuss how to update $n$.MaxSid using the child nodes of $n$. On the one hand, $n$.MaxSid (or $c$.MaxSid) is defined as the smallest specific set in all the leaf nodes in the subtree rooted at $n$ (or $c$). On the other hand, the leaf nodes in the subtree rooted at $n$ are exactly the leaf nodes in all the subtrees rooted at the child nodes of $n$. Thus we can update $n$.MaxSid as the smallest specific set $c$.MaxSid among all its child nodes. For example, consider the node $n_1$ in Fig. 4(c). It has two child nodes $n_2$ and $n_8$, whose specific sets are respectively $n_2$.MaxSid $= \mathsf{S}_7$ and $n_8$.MaxSid $= \mathsf{S}_5$. Thus we set $n_1$.MaxSid as the smaller one $\mathsf{S}_5$.

Next we discuss how to calculate $n$.RidList based on the child nodes of $n$. Recall that $n$.RidList (or $c$.RidList) is the list of leaf nodes where **(1)** their specific sets are $n$.MaxSid (or $c$.MaxSid) and **(2)** $n$.MaxSid (or $c$.MaxSid) exists in all the nodes on the paths from $n$ (or $c$) to these leaf nodes. Thus we set $n$.RidList as $\phi$ if $n$.MaxSid does not exist in the node $n$ as none of the leaf nodes satisfies condition **(2)**; otherwise, we update $n$.RidList as the union of all $c$.RidList where $c$ is a child node of $n$ and $c$.MaxSid $= n$.MaxSid. This is because, on the one hand, for any leaf node in these $c$.RidList, **(1)** its specific set is $c$.MaxSid $= n$.MaxSid and **(2)** $c$.MaxSid $= n$.MaxSid exists in both all the nodes on the path from $c$ to this leaf node and the parent node $n$ of $c$. Thus this leaf node must also in $n$.RidList. On the other hand, for any leaf node not in the above $c$.RidList, either its specific set is not $c$.MaxSid or $c$.MaxSid does not exists in a node on the path from $c$ to this leaf node, which indicates that this leaf node must also not in $n$.RidList. For example, consider the node $n_3$ in Fig. 4(b). It has two child nodes $n_4$ and $n_6$ where $n_4$.RidList $= \phi$ and $n_6$.RidList $= \{\mathsf{R}_1\}$. As $n_3$.MaxSid $= \mathsf{S}_3$ exists in $n_3$'s inverted list $\mathcal{I}[n_3.e]$, we have $n_3$.RidList $= n_4$.RidList $\cup$ $n_6$.RidList $= \{\mathsf{R}_1\}$. As another example, consider the node $n_3$ in Fig. 4(a). As $n_3$.MaxSid $= \mathsf{S}_1$ does not exist in $\mathcal{I}[n_3.e = e_2]$, we have $n_3$.RidList $= \phi$. Note that if $n$ is a leaf node, based on the definition, if $n$.MaxSid exists in the inverted list $\mathcal{I}[n.e]$, we have $n$.RidList $= \{n\}$.

Lastly, we show how to update the specific set on the leaf nodes. Note that a postorder tree traversing is also a type of deep first traversing. Thus a node $n$ must be traversed to through all its ancestor nodes. As such when we traverse to a node $n$, we can get the largest gap of $n$ and all its ancestors. We keep this largest gap in a variable NextMax. This variable NextMax will be passed through the parent node to all its child nodes and get updated in the postorder (i.e., deep first) tree traversing. Then, whenever a leaf node $n$ is reached, we can update its new specific set $n$.MaxSid as NextMax.

Note that, the gap in a node $n$ (i.e., the first entry in the inverted list $\mathcal{I}[n.e]$ greater than the specific set $n$.MaxSid) can be calculated at the time of checking whether the specific set $n$.MaxSid exists in $\mathcal{I}[n.e]$. To reuse this computation later when updating the specific sets in next round, we keep the gap in a variable $n$.NextMax. More specifically, we can binary search for the first entry Sid no smaller than $n$.MaxSid. If Sid is identical to $n$.MaxSid, it means $n$.MaxSid exists in the inverted list $\mathcal{I}[n.e]$ and we use the entry next to Sid in the inverted list as the gap $n$.NextMax; otherwise, it means $n$.MaxSid does not exist in the inverted list and we use Sid as the gap $n$.NextMax.

The pseudo-code of the tree-based method is shown in

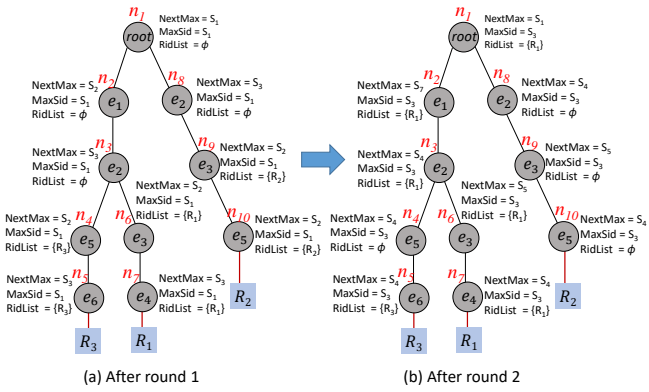



Fig. 5. Status of the prefix tree.

Algorithm 2. It takes two datasets $\mathbb{R}$ and $\mathbb{S}$ as input and returns their set containment join. The tree-based method first builds a prefix tree $\mathcal{T}$ for $\mathbb{R}$ and an inverted index $\mathcal{I}$ for $\mathbb{S}$ (Lines 2 to 3). Then for each node $n$ in the tree, it initializes both the specific set $n$.MaxSid and the gap $n$.NextMax as $\mathsf{S}_1$ (Line 4). Next it repeatedly invokes the procedure PostOrderTraverse (Line 6). Each invocation will get $\mathcal{T}$.root.MaxSid updated as the next smallest specific set to check in all the leaf nodes and get $\mathcal{T}$.root.RidList updated according. Based on the definitions, all the sets in $\mathcal{T}$.root.RidList are subsets of $\mathcal{T}$.root.MaxSid. Thus we add a pair $(\mathsf{R}, \mathcal{T}.\text{root.MaxSid})$ to the result $\mathcal{A}$ for each set $\mathsf{R} \in \mathcal{T}$.root.RidList (Lines 7 to 8). It terminates when the smallest specific set $\mathcal{T}$.root.MaxSid is identical to the "maximum" set $\mathsf{S}_\infty$, which indicates the end of an inverted list is reached for every leaf node (Line 5). Finally, the result $\mathcal{A}$ is returned (Line 9).

The pseudo-code of the postorder tree traversing PostOrderTraverse is shown in Algorithm 3. It takes a tree node $n$ and the largest gap NextMax in all the ancestors of $n$ as input. At the end of each invocation, the variables $n$.MaxSid, $n$.RidList, and $n$.NextMax will get updated. At the beginning, NextMax is the largest gap in all the ancestor nodes of $n$ and $n$.NextMax is the gap on $n$. Thus the larger one of these two is the largest gap in all the ancestor nodes of any child node $c$ of $n$ (Line 2). Then the postorder traversing recursively invoke itself on the child nodes of $n$ to get their specific sets updated (Lines 3 and 4). Note that for the child node $c$ of $n$ whose specific set $c$.MaxSid is larger than the largest gap NextMax, $c$.MaxSid does not need to be updated as this specific set has not been checked yet. Thus the procedure will not be recursively invoked on these child nodes. After this, all the child nodes of $n$ have their variables update-to-date. Next, it updates the specific set $n$.MaxSid of $n$. As discussed before, if $n$ is a leaf node, $n$.MaxSid is updated as the largest gap NextMax; otherwise, $n$.MaxSid is updated as the smallest $c$.MaxSid where $c$ is a child of $n$ (Lines 5 to 8). Then it updates $n$.RidList and the gap $n$.NextMax. For this purpose, it first binary searches for the first entry Sid in $\mathcal{I}[n.e]$ no smaller than $n$.MaxSid (Line 9). As discussed before, if Sid $==$ $n$.NextMax, we update the gap $n$.NextMax as the entry next to Sid in $\mathcal{I}[n.e]$ (Line 11); otherwise, we update the gap $n$.NextMax as Sid (Line 17). Note that if we reach the end of the inverted list, we set Sid and the gap as the "maximum" set $\mathsf{S}_\infty$, i.e., $n$.NextMax $=$ Sid $= \mathsf{S}_\infty$. For $n$.RidList, as discussed before, if Sid $==$ $n$.MaxSid, which indicates the specific set $n$.MaxSid exists in $n$ and its inverted list $\mathcal{I}[n.e]$, and $n$ is a leaf node, we set $n$.RidList as the set corresponding to $n$ itself, i.e., $n$.RidList $= \{n\}$ (Line 13). However, if Sid $==$ $n$.MaxSid and $n$ is not a leaf node, we update $n$.RidList as the union of $c$.RidList where $c$ is a child node of $n$ and $c$.MaxSid $= n$.MaxSid (Line 15). In the case Sid $\neq$ $n$.MaxSid, which indicates the specific set $n$.MaxSid does not exist in $n$, we have $n$.RidList $= \phi$ (Line 17). Now all the variables $n$.MaxSid, $n$.RidList, and $n$.NextMax of $n$ are update-to-date.

*Example 5: Consider the running example in Fig. 4. Figure 5(a) shows the status of the prefix tree after the first round of traversing. We discuss how this then transverses to the second round (as shown in Fig. 5(b)). First, the* PostOrderTraverse *algorithm traverses to the leaf node $n_5$, through its ancestor nodes $n_4$, $n_3$, $n_2$, and $n_1$, and updates the variable NextMax to be the largest one amongst $n_5$.NextMax, $n_4$.NextMax, $n_3$.NextMax, $n_2$.NextMax, and $n_1$.NextMax, which is $n_3$.NextMax $= \mathsf{S}_3$. Then, as $n_5$ is a leaf node, it updates $n_5$.MaxSid to be NextMax $= \mathsf{S}_3$. Next it binary searches for the first entry in $\mathcal{I}[n_5.e = e_6]$ no smaller than $\mathsf{S}_3$ and obtains Sid $= \mathsf{S}_3$. As Sid $==$ $n$.MaxSid, it sets $n$.NextMax to be the next entry after $\mathsf{S}_3$ in $\mathcal{I}[e_6]$, which is $\mathsf{S}_4$. It also sets $n_5$.RidList to be the corresponding set of $n$, which is $\mathsf{R}_3$, since $n_5$ is a leaf node. Next it visits the node $n_4$. As $n_4$ is not a leaf node, it sets $n_4$.MaxSid to be the smallest specific sets among all of its child nodes, which is $n_5$.MaxSid $= \mathsf{S}_3$. As the first entry no smaller than $\mathsf{S}_3$ in $\mathcal{I}[n_4.e = e_5]$ is Sid $= \mathsf{S}_4$, which is not identical to $n_4$.MaxSid $= \mathsf{S}_3$, it sets $n_4$.NextMax as Sid $= \mathsf{S}_4$ and $n_4$.RidList $= \phi$. Similarly, it traverses and updates all the nodes in the prefix tree and Fig. 5(b) shows the status of the tree at the end of the postorder tree traversing.*

**Correctness and soundness.** We first show the correctness of the tree-based method. In the algorithm, the leaf node $v$ is only added to $v$.RidList if $v$.MaxSid is found in $v$. For the inner node $n$ and its child node $c$, the sets in $c$.RidList are only added to $n$.RidList if $c$.MaxSid $=$ $n$.MaxSid and $n$.MaxSid is found in $n$. Thus, recursively, we have that for the root node $\mathcal{T}$.root and a leaf node $n$, the leaf node $n$ is only added to $\mathcal{T}$.root.RidList if $n$.MaxSid $=$ $\mathcal{T}$.root.MaxSid and $\mathcal{T}$.root.MaxSid is found in all the nodes on the path from $\mathcal{T}$.root to $n$. This implies a set containment relationship between $n$ (i.e., its corresponding set R) and $\mathcal{T}$.root.MaxSid. Thus for any $\mathsf{R} \in \mathcal{T}$.root.RidList, the pair $(\mathsf{R}, \mathcal{T}.\text{root.MaxSid})$ is a result. Next we show the soundness of our method. As discussed before, for any set pair $(\mathsf{R}, \mathsf{S})$ in $\mathbb{R} \times \mathbb{S}$ where $\mathsf{R} \subseteq \mathsf{S}$, all the inverted lists of R must have the entry S. Let $n$ be the leaf node corresponding to R. Then S

**Algorithm 2**: TREE-BASED METHOD

**Input**: $\mathbb{S}$ and $\mathbb{R}$;
**Output**: $\mathcal{A}$: $\mathbb{R} \bowtie_{\subseteq} \mathbb{S} = \{(\mathsf{R}, \mathsf{S}) | \mathsf{R} \subseteq \mathsf{S}, \mathsf{R} \in \mathbb{R}, \mathsf{S} \in \mathbb{S}\}$;

```
1  begin
2      build a prefix tree T on R;
3      build an inverted index I on S;
4      for each node n ∈ T, n.MaxSid = n.NextMax = S_1;
5      while T.root.MaxSid ≠ S_∞ do
6          PostOrderTraverse(T.root, 1);
7          foreach R ∈ T.root.RidList do
8              add (R, T.root.MaxSid) to A;
9      return A;
10 end
```

**Algorithm 3**: POSTORDERTRAVERSE($n$, NEXTMAX)

**Input**: $n$: a tree node; NextMax: the largest gap in all the ancestor nodes of $n$.

```
1  begin
2      NextMax = max(NextMax, n.NextMax);
3      foreach child c of n where c.MaxSid ≤ NextMax do
4          DeepFirstTraverse(c, NextMax);
5      if n is a leaf node then
6          set n.MaxSid as NextMax;
7      else
8          set n.MaxSid as the smallest c.MaxSid, where c
           is a child node of n;
9      binary search for the first entry Sid no smaller than
       n.MaxSid in I[n.e];
10     if Sid == n.MaxSid then
11         set n.NextMax as the entry next to Sid in I[n.e];
12         if n is a leaf node then
13             set n.RidList as the set corresponding to n;
14         else
15             set n.RidList as the union of c.RidList where
               c is a child of n and c.MaxSid = n.MaxSid;
16     else
17         set n.NextMax as Sid and n.RidList = φ;
18 end
```

exists in all the nodes on the path from $\mathcal{T}$.root to $n$ and cannot be skipped. That is, the specific set $n$.MaxSid must be updated to be S at some point. At the time when $n$.MaxSid = S, R will be added to $n$.RidList as $n$.MaxSid exists in $n$. In our algorithm, $n$.MaxSid and $n$.RidList will be propagated to $n$'s ancestor nodes once $n$.MaxSid becomes the smallest specific set in the subtree rooted at these ancestor node. When $n$.MaxSid = S and $n$.RidList = {R} are propagated to the root node $\mathcal{T}$.root, the pair (R, S) will be returned in our algorithm.

**Algorithm 4**: EARLYTERMINATION

```
1  begin
       // add to Algorithm 3 after Line 17
          within the if-else condition
2      PostOrderTraverse(n, NextMax);
3  end
```

### C. Early Termination for the Tree-based Method

In this section, we extend the early termination technique to support the tree-based method.

In the framework, whenever the specific set is not found in an inverted list, the early termination stops binary searching the rest of inverted lists and uses the largest gap in the visited inverted list as the new specific set in the next round. Similarly, in the tree-based method, if a specific set $n$.MaxSid is not found in the inverted list $\mathcal{I}[n.e]$ of a node $n$, the specific sets of some leaf nodes in the subtree rooted at $n$ need to be updated. To this end, we recursively invoke the procedure POSTORDERTRAVERSE on node $n$ to update the variables of $n$. Only if the specific set $n$.MaxSid is found in the inverted list $\mathcal{I}[n.e]$, will this procedure traverse to the parent node of $n$.

## V. DATA PARTITIONING

In this section, we discuss our data partition methods. We first introduce how to partition the data in Section V-A and then discuss how to deal with each partition using different methods in Section V-B.

### A. Partitioning the Sets

In this section, we further improve the efficiency and scalability of our proposed tree-based method by partitioning the data. The basic idea is that we can first partition the sets in $\mathbb{R}$ into disjoint partitions. Then, for each partition, we construct a "*local*" inverted index using a small part of the sets in $\mathbb{S}$ such that the rest of sets in $\mathbb{S}$ are not a superset of any set R in this partition. Then we can use the previous tree-based method to process each partition and its corresponding local inverted index to get all the results in this partition. Together, we can get the set containment join result.

For this purpose, in this paper, we propose to partition the sets in $\mathbb{R}$ by their smallest elements in the global order. For example, consider the dataset $\mathbb{R}$ in Table I. The smallest elements in $\mathsf{R}_1$, $\mathsf{R}_2$, and $\mathsf{R}_3$ are $e_1$, $e_2$, and $e_1$, respectively. Thus, we partition $\mathbb{R}$ into two partitions $\mathbb{R}_{e_1}$ and $\mathbb{R}_{e_2}$ where $\mathbb{R}_{e_1} = \{\mathsf{R}_1, \mathsf{R}_3\}$ and $\mathbb{R}_{e_2} = \{\mathsf{R}_2\}$. Clearly, each set in $\mathbb{R}$ is allocated into one and only one partition in this partition scheme. Next, we deal with the sets in $\mathbb{S}$. Let $\mathbb{R}_e$ denote the partition with all the sets whose smallest elements are $e$. Clearly, all the sets in $\mathbb{R}_e$ contain the element $e$. Thus, for a set S to be a superset of any set R in the partition $\mathbb{R}_e$, S must also contain the element $e$. Thus we construct a local inverted index $\mathcal{I}_e$ using only those sets in $\mathbb{S}$ containing the element $e$. The rest of sets in $\mathbb{S}$ do not contain $e$ and cannot be a superset

of any set in $\mathbb{R}_e$. Then we use the tree-based method to deal with the partition $\mathbb{R}_e$ and its corresponding local inverted index $\mathcal{I}_e$ to get results in this partition. In our implementation, we use the element frequency order as the global order and the most frequent element to partition the data.

*Example 6: Consider the datasets in Table I. As discussed above, our data partition scheme will partition $\mathbb{R}$ into $\mathbb{R}_{e_1}$ and $\mathbb{R}_{e_2}$. For the partition $\mathbb{R}_{e_1}$, we construct an inverted index $\mathcal{I}_{e_1}$ with four sets $\mathsf{S}_1$, $\mathsf{S}_2$, $\mathsf{S}_3$, and $\mathsf{S}_7$ containing $e_1$. For the partition $\mathbb{R}_{e_2}$, we construct another inverted index $\mathcal{I}_{e_2}$ with five sets $\mathsf{S}_3$, $\mathsf{S}_4$, $\mathsf{S}_5$, $\mathsf{S}_6$, and $\mathsf{S}_7$ containing $e_2$. Take the left subtree rooted at $n_2$ where $n_2.e = e_1$ in Fig. 4 as an example. As shown in Fig. 2 and Fig. 4, initially, the average inverted list length is 5 for the left subtree, and the length is reduced to 2.8 (i.e., 4, 2, 4, 2, 2, 3 for each inverted list respectively) after the partition method is applied.*

Obviously, the prefix tree for the partition $\mathbb{R}_e$ is a branch of the prefix tree for the entire dataset $\mathsf{R}$, i.e., the subtree rooted at the child node of $\mathcal{T}.\mathsf{root}$ with element $e$. In addition, the size of the local inverted index $\mathcal{I}_e$ is much smaller than the original inverted index $\mathcal{I}$. Actually, each inverted list in the local inverted index $\mathcal{I}_e$ is a sub-list of the corresponding inverted list in the original inverted index $\mathcal{I}$. Thus, the binary search cost decreases in the tree-based method for each partition. However, for some extremely small partitions, the overhead of constructing the local inverted index may be even larger than directly applying the tree-based method on the original inverted index. In the next section, we discuss how to determine which inverted index to use for each partition.

### B. Processing the Partitions

Note that to efficiently find all the sets in $\mathbb{S}$ containing a specific element $e$, we still need to construct the original inverted index $\mathcal{I}$ based on the entire dataset $\mathbb{S}$. Then we can use the sets in $\mathcal{I}[e]$ to construct the local inverted index $\mathcal{I}_e$. However, for some small partitions, the local inverted index construction cost may be even larger than the benefit of replacing the original inverted index with the local inverted index. Since the local inverted index construction cost grows linearly with increasing partition size while the set containment join cost on each partition grows quadratically, we propose to use the original inverted index for the small partitions and the local inverted index for the large partitions.

To dynamically determine the partition size boundary for these two inverted indexes, we propose to visit the partitions in increasing order of size. For each partition $\mathbb{R}_e$, we first use the original inverted index to process it. In the meanwhile, we count the cost $Y$ of using the original inverted index. Then, we estimate the cost of using the local inverted index to process this partition as $Y \cdot \frac{|\mathcal{I}[e]|}{|\mathbb{S}|}$, where $|\mathcal{I}[e]|$ and $|\mathbb{S}|$ are the number of sets in $\mathcal{I}[e]$ and $\mathbb{S}$. The local inverted index construction cost can be estimated as the total size of sets in $\mathcal{I}[e]$. Once the total estimated cost is steadily no greater than $Y$, we start directly using the local inverted index to process the remaining partitions.

TABLE II
STATISTICS OF THE REAL-WORLD DATASETS

| Dataset | # of Sets | Min/Max/Avg Size | # of Elements | $z$-value |
|---|---|---|---|---|
| FLICKR | 3,546,729 | 1 / 1230 / 5.4 | 618,971 | 0.63 |
| AOL | 36,389,577 | 1 / 125 / 2.5 | 3,849,556 | 0.68 |
| ORKUT | 15,301,901 | 2 / 9120 / 7 | 2,322,299 | 0.13 |
| TWITTER | 28,819,434 | 2 / 4998 / 9 | 13,096,918 | 0.3 |

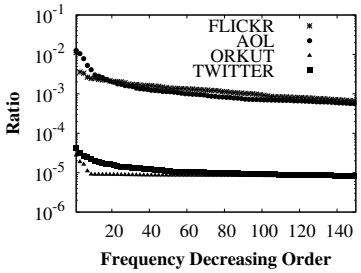



Fig. 6. Frequency Distribution of Real-world Datasets.

## VI. EXPERIMENT

In this section, we evaluate the efficiency and scalability of our proposed methods.

### A. Experimental Setup

We conducted experiments on both real-world and synthetic datasets. As with previous studies [3], [24], we evaluate all the methods on the self-join case, i.e., $\mathbb{R} = \mathbb{S}$. However, all our proposed techniques can be seamlessly adapted for the two-relation join case. Particularly, we use the following four real-world datasets: FLICKR[1], AOL[2], ORKUT[3] and TWITTER[4]. The FLICKR dataset is a photo-tag dataset. Each photo corresponds to a set and each photo tag corresponds to an element. The AOL dataset is a query log dataset. Each query corresponds to a set and each whitespace-split query word corresponds to an element. The ORKUT dataset contains community information from a free on-line social network. Each community is a set while each user in the community is an element. The TWITTER dataset is a social network dataset. Each user corresponds to a set. The followers of a user are the elements of the corresponding set. Note that in TWITTER dataset, we removed the sets with more than 5000 elements to keep the number of results reasonable. Table II provides some statistics for these four datasets.

As with the previous work [25], we makes use of *Zipf's* law [17] to generate the synthetic datasets with four parameters. (1) Data cardinality, i.e., the number of sets in the dataset, ranges from 2.5 million to 10 million. (2) The average set size ranges from 4 to 128. (3) The number of distinct elements ranges from 10 thousands to 10 million. (4) the $z$-value, which measures the skewness of the datasets, ranges from 0.25 to 1.0. The higher the $z$-value is, the more "skew" the dataset is.

[1]https://www.flickr.com
[2]http://www.cim.mcgill.ca/∼dudek/206/Logs/
[3]https://snap.stanford.edu/data/com-Orkut.html
[4]https://snap.stanford.edu/data/twitter-2010.html

TABLE III
STATISTICS OF THE SYNTHETIC DATASETS

| Parameter | Values |
|---|---|
| data cardinality | 2.5M, 5M, **10M**, 20M |
| average set size | 4, **8**, 16, 32, 64, 128 |
| number of distinct elements | 10K, 100K, **1M**, 10M |
| $z$-value | 0.25, **0.5**, 0.75, 1.0 |

More specifically, a dataset with $z$-value $1 - \frac{\log(a/100)}{log(b/100)}$ means the most frequent $b$ percent of elements accounts for $a$ percent of the total number of elements in the dataset. For example, if in a dataset the most frequent 20% of elements accounts for 80% of the total number of elements, i.e., $a = 80$ and $b = 20$, the $z$-value is $0.86$. Similarly, for a more even dataset where $a = b = 50$, the $z$-value is $0$. Table III summarizes the statistics of the synthetic datasets. On each experiment, we vary one of the parameters and set the other parameters to their default values (in bold font in the table). Note that the previous work [25] uses significantly higher $z$-values (greater than 1.0). We argue that in the real-world, the $z$-values of most datasets are within 1.0 based on the 80/20 law [17].

For the four real-world datasets, Fig. 6 shows the percentage of the total number of elements that the top 150 most frequent elements account for. We can see that the most frequent elements in the FLICKR and AOL datasets account for much higher percentage (about $100\times$) of the total elements than those of the ORKUT and TWITTER datasets, which indicates that the FLICKR and AOL datasets are more skew than the ORKUT and TWITTER datasets.

We compared LCJoin with three state-of-the-art algorithms, PRETTI [10], LIMIT+ [3] and TT-Join [24], [25]. PRETTI indexes $\mathbb{R}$ with a prefix tree and $\mathbb{S}$ with an inverted index. The prefix tree is traversed in a depth-first manner and the corresponding inverted lists on the nodes are intersected so that the list intersections on common prefix are shared. LIMIT+ improves PRETTI by employing a cost model to decide online whether to stop list intersections as the number of candidates may be small. In our experiment, we used the trained cost model provided by the author. TT-Join uses $k$ least frequent elements as the signature and a candidate is generated and verified each time the signature is matched when traversing the prefix tree built on $\mathbb{S}$. In our experiment, we set the parameter $k$ as 3, which is the same as in [25].

All the methods were implemented in C++ and compiled using g++ 5.4.0 with -O3 flag. We reimplemented PRETTI and TT-Join and got the source code of LIMIT+ from its author. The experiments were ran on a workstation powered by a 20-core Intel Xeon Gold-6148 CPU on Linux (Ubuntu 16.04) with 64 GB main memory.

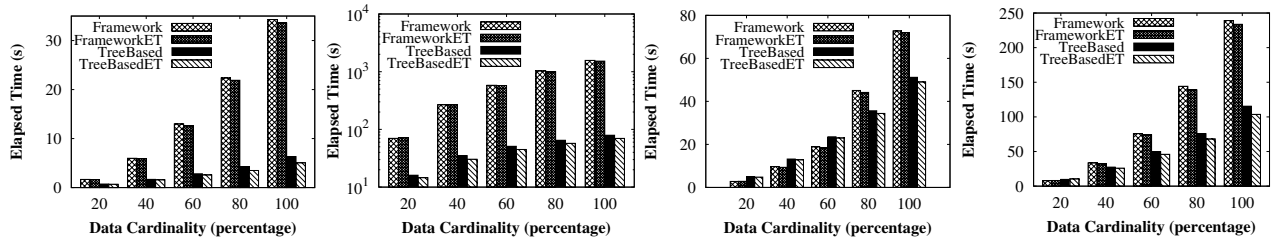
### B. Evaluating the Tree-based Methods

In this section, we evaluate the efficiency of our proposed framework method and tree-based method, along with the early termination techniques. We implemented the following four methods. (1) Framework uses the framework method as described in Algorithm 1. It intersects the inverted lists in a cross-cutting way. (2) FrameworkET improves Framework with the early termination technique as discussed in Section III-C. (3) TreeBased utilizes a prefix tree index to share the computation on $\mathbb{R}$ as described in Algorithm 2. (4) TreeBasedET integrates the early termination technique into the TreeBased method as discussed in Section IV-C.

We varied the data cardinality (using 20%, 40%, 60%, 80%, and 100% of the sets in the datasets) and reported the runtime of different methods. Figure 7 shows the experimental results on the four real-world datasets. We observed that the two tree-based methods TreeBasedET and TreeBased outperformed the two framework methods Framework and FrameworkET by up to $20\times$ when the data cardinality was large ($\geq 80\%$). For example, on the AOL dataset with 100% data cardinality, the time elapsed for TreeBasedET, TreeBased, FrameworkET, and Framework were 70s, 79s, 1524s, and 1568s, respectively. For small data cardinality, the framework methods occasionally outperformed the tree-based methods. This is because the tree methods can share the computation in the common prefixes of the sets. The larger the data cardinality is, the more computation can be shared. In contrast, if the data cardinality is too small, the overhead, such as constructing and initializing the prefix tree, may be larger than the benefit of shared computation. We also observed that the early termination techniques helped improve the performance. This is because they can avoid unnecessary binary search operations.

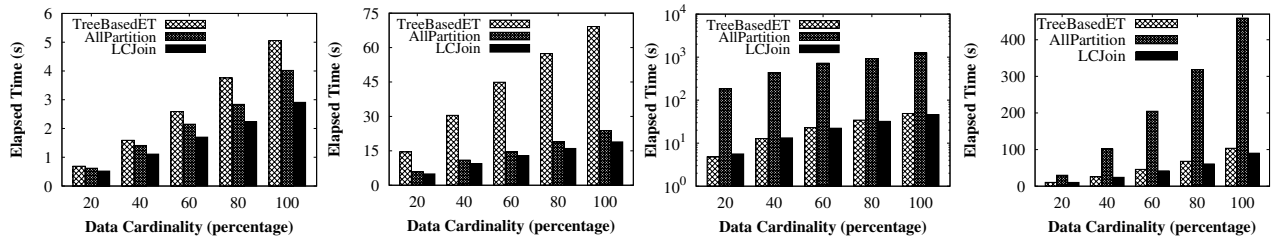
### C. Evaluating the Data Partition Methods

In this section, we evaluate the data partition methods. We implemented two methods. 1) AllPartition partitions the sets in $\mathbb{R}$ using their smallest elements in the global order and uses the local inverted index and the tree-based method to process all partition as discussed in Section V-A. 2) LCJoin uses the method as described in Section V-B to determine whether to use the local inverted index or the original inverted index to process each partition . We varied the data cardinality and reported the runtime of TreeBasedET, AllPartition, and LCJoin. Note TreeBasedET does not partition the data. The results on the real-world datasets are shown in Fig. 8. We can see from the figure that LCJoin always achieved the best performance. For example, on the AOL dataset with 100% data cardinality, the runtime of TreeBasedET, AllPartition, and LCJoin were 79s, 24s, and 19s, respectively. This is because the partition based methods can reduce the inverted index size for each partition, which results in less binary search cost and more skipping of irrelevant entries in the inverted lists. We also noticed that the partition based method AllPartition sometimes did not perform as well as the non partition method TreeBasedET. This is because, when the partition size is very small, the local inverted index construction cost for this partition may be larger than the cost of directly using the original inverted index. LCJoin alleviates this issue by dynamically determining whether to use the original inverted index or to build a local inverted index to process each partition.




(a) FLICKR (b) AOL (c) ORKUT (d) TWITTER

Fig. 7. Evaluation of the tree-based methods.




(a) FLICKR (b) AOL (c) ORKUT (d) TWITTER

Fig. 8. Evaluation of the data partition methods.

### *D. Comparing with Existing Methods on Real-world Datasets*

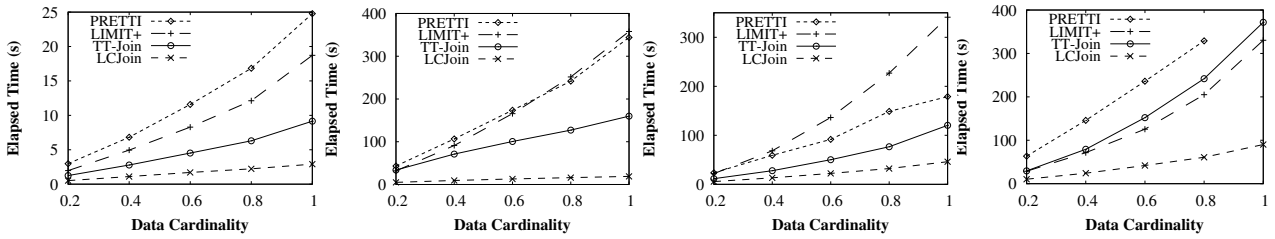
In this section, we compared LCJoin with three state-of-the-art methods, PRETTI, LIMIT+, and TT-Join. We varied the data cardinality and reported the runtime of different methods. Figure 9 shows the results. Note that on the TWITTER dataset with data cardinalities 80% and 100%, PRETTI failed to return results due to an out-of-memory error. We observed that LCJoin always achieved the best performance and improved existing methods by up to 10×. For example, on the AOL dataset with 100% cardinality, the runtime for PRETTI, LIMIT+, TT-Join, LCJoin were 344s, 358s, 160s, and 19s, respectively. This is because our cross-cutting based list intersection can skip many irrelevant entries in the inverted lists and the data partition technique can reduce the size of the inverted index in each partition. We also observed that our approach LCJoin scaled very well when data cardinality was increased. For example, on FLICKR dataset, when the data cardinality was 20%, 40%, 60%, 80% and 100%, the runtime of our approach were respectively 0.52s, 1.11s, 1.7s, 2.23s, and 2.91s, which suggests an almost linear growth. This is due to the fact that our tree-based method can share computation in the common prefixes of the sets.

We also measured the peak memory usage of the four algorithms. Figure 10 shows the results. We observed that LCJoin had the lowest peak memory usage in nearly all cases. For example, on the FLICKR dataset, the peak memory usage for PRETTI, LIMIT+, TT-Join, and LCJoin were 2.4GB, 1.21GB, 1.03GB, and 0.83GB, respectively. The main reason for this is that TT-Join utilizes two sparse tree structures in its algorithm, which consumes more memory than our compact tree structure. Though PRETTI and LIMIT+ also make use of compact tree structures, their top-down list intersections generate a large number of intermediate results, which leads to bad memory fragmentation, and thus have larger peak memory usage. Although our data partition technique may require building local inverted indexes for the partitions, all of these indexes can share the memory through a memory pre-allocation mechanism.

### *E. Comparing with Existing Methods on Synthetic Datasets*

In this section, we compared our method LCJoin with existing methods on the synthetic datasets. We evaluated these methods on synthetic datasets using different parameters and reported their runtime. Figure 11 shows the results. We can see that our approach outperformed existing methods in all settings by up to 70×. For example, as shown in Fig. 11(c), when the $z$-value was 0.5, the average set size was 8, the number of distinct elements was 10 thousand, and the data cardinality was 10 million, the runtimes for PRETTI, LIMIT+, TT-Join, and LCJoin were 1687s, 1393s, 3604s and 52s, respectively.

More specifically, Fig. 11(a) depicts the results for scalability experiments. We can see that our approach showed good scalability with the increasing of data cardinality, just as we observed for the real-world datasets. Note that the performance of TT-Join decreased faster than other algorithms with the increase of the data cardinality, this is because the fixed-length signature scheme ($k$ least frequent prefix) is not adaptive to the datasets. Figure 11(b) shows the results for different average set sizes. PRETTI failed to return results when the average set size were greater than 32. It also shows good scalability of our algorithm with the increase of the average set size. Figure 11(c) gives the runtime of all algorithms when varying the number of distinct elements. We observe that the




(a) FLICKR (b) AOL (c) ORKUT (d) TWITTER

Fig. 9. Comparing with existing approaches on real-world datasets.

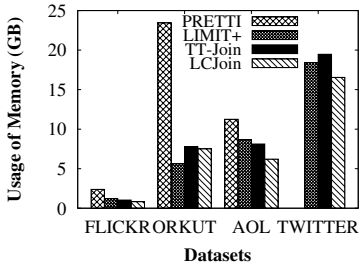


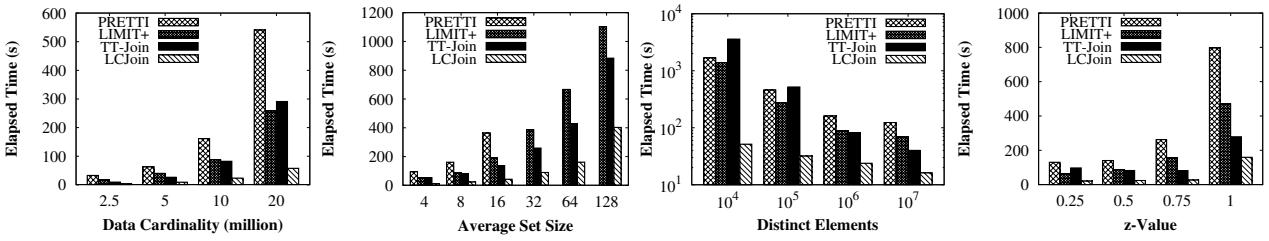

Fig. 10. Comparing peak memory usage with existing methods on real-world datasets.

performance of LCJoin is rather steady. For example, when the number of distinct element were 10 thousands and 10 million, the runtime of our approach were respectively 52s and 16s, while for TT-join, PRETTI, and LIMIT+ the run time were 3604s/1687s/1393s and 40s/124s/69s. The main reason is that our data partition method can effectively reduce the inverted index size, which alleviate the side-effect caused by long inverted lists, especially when the number of distinct elements is small. Figure 11(d) presents the results of varying the $z$-value of elements. LCJoin performed well under different $z$-value and outperformed PRETTI, LIMIT+, and TT-Join by up to 9.8×, 5.8×, and 4.7×.

## VII. Related Work

**Set Containment Joins.** Several methods have been developed to address the set containment join problem, some of them union-oriented and others intersection-oriented. For union-oriented methods, Helmer and Moerkotte [9] proposed to use the Signature-Hash Join (SHJ) to address the set containment join problem. SHJ first uses a signature structure [19] to compactly represent sets, and then performs signature enumerations and comparisons to filter unqualified set pairs. As SHJ involves an expensive signature enumeration cost, it scales poorly with the dataset cardinality. Ramasamy et al. [18] proposed the Partitioned Set Join (PSJ) method, while Melnik et al. [15] developed the Divide-and-Conquer Set Join (DCJ) method. PSJ and DCJ both employ a hash function to partition the sets into different buckets such that two sets have a set containment relationship only if they reside in the same bucket. The pairs in the same bucket are then further verified. Melink et al. [16] improved PSJ and DCJ by using a comprehensive model to analyze the partitioning algorithms. They proposed to use more sophisticated partitioning strategies to improve the filtering efficiency. Luo et al. [13] further improved these methods by using a Patricia tree to reduce the signature enumeration cost. Yang et al. [24], [25] proposed the TT-Join, which takes the data skewness into account. For each set in $\mathbb{R}$, TT-Join uses its $k$ least frequent elements as the signature. However, as shown in several previous studies [3], [24], [25], most union-oriented methods are not competitive with intersection-oriented methods. Mamoulis [14] proposed the Block Nested Loop Join (BNL), which first builds an invert index over $\mathbb{S}$ and then performs list intersections for all the elements in each $\mathsf{R} \in \mathbb{R}$. Jampani and Pudi [10] improved BNL by using a prefix tree to share the computation across the sets in $\mathbb{R}$. Luo et al. [13] further improved BNL by replacing the prefix tree with a more compact tree structure, the Patricia tree. To avoid performing too many inverted list intersections for a large prefix tree, Bouros et al. [3] proposed the LIMIT+ method which only uses up to $l$ elements in the sets to construct the prefix tree. In addition, they also proposed the Order and Partition Join technique to build the inverted index incrementally, resulting in a lower list intersection cost. Finally, Kunkel et al. [11] proposed the PIEJoin method, which uses a tree structure to reduce the size of the inverted index on $\mathbb{S}$. Note that all these intersection-oriented methods utilize the "rip-cutting" based list intersection while we propose to use the "cross-cutting" based list intersection.

**Set Similarity Joins.** Another relevant line of research is the set similarity joins. A set similarity join takes two collections of sets and identifies all the set pairs that are similar with regards to a given similarity function and threshold. For example, Deng et al. [7] proposed a partition-based method for set similarity joins under Jaccard similarity constraints. Bayardo et al. [2], on the other hand, proposed to apply the prefix filtering technique for the set similarity join. For each set, the prefix filter first sorts the elements using a global order and then uses the first few elements as the prefix. The prefix filter technique guarantees that two sets are similar only if their prefixes share at least one element. Xiao et al. [23] improved the prefix filter with a position filter. Wang et al. [22] proposed AdaptJoin algorithm, which uses a longer prefix to filtering more dissimilar pairs. Deng et al. [8] proposed a size-aware




(a) varying the data cardinality (b) varying the average set size (c) varying the distinct elements (d) varying the $z$-value

Fig. 11. Comparing with existing approaches on synthetic datasets.

method for the overlap set similarity join problem, which finds all the set pairs with a sufficient large overlap size. Deng et al. [5] proposed an efficient method to find related sets using two-tier similarity functions. To further improve the scalability, Vernica et al. [21], Sun et al. [20], and Deng et al. [6] proposed to perform the set similarity join on MapReduce [4] or Spark [26]. Agrawal et al. [1] proposed to solve the error-tolerant set containment join problem. Li et al. [12] developed algorithms to solve the T-occurrence problem. These methods can be adapted to solve our set containment join problem. However, as shown in [25], they did not perform well when applied to our problem.

## VIII. Conclusion

In this paper, we studied the set containment join problem, which, given two collections $\mathbb{R}$ and $\mathbb{S}$ of sets, finds all the set pairs in $\mathbb{R} \times \mathbb{S}$ with a set containment relationship. Existing methods can be broadly classified into union-oriented and intersection-oriented methods. The union-oriented methods are not competitive because they involve an expensive signature enumeration step. The intersection-oriented methods build an inverted index on $\mathbb{S}$. In contrast to existing intersection-oriented methods, which use the rip-cutting fashion to intersect inverted lists, we design a cross-cutting based list intersection method. The cross-cutting based list intersection can skip many irrelevant entries in the inverted lists by using the gaps between two consecutive entries in the inverted lists. To share computation across sets, we built a prefix tree on $\mathbb{R}$ and extend the cross-cutting based list intersection to operate on this prefix tree. To further improve the efficiency and scalability of our proposed method, we partitioned the sets in $\mathbb{R}$ according to their smallest elements in the global order. We also developed a method to apply different approaches to each partition. We evaluated our techniques on both real-world and synthetic datasets. Experimental results showed that our approach outperformed existing methods by up to $10\times$ for the real-world datasets.

## References


[1] P. Agrawal, A. Arasu, and R. Kaushik. On indexing error-tolerant set containment. In *SIGMOD*, pages 927–938, 2010.

[2] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.

[3] P. Bouros, N. Mamoulis, S. Ge, and M. Terrovitis. Set containment join revisited. *Knowl. Inf. Syst.*, 49(1):375–402, 2016.

[4] J. Dean and S. Ghemawat. Mapreduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010.

[5] D. Deng, A. Kim, S. Madden, and M. Stonebraker. Silkmoth: An efficient method for finding related sets with maximum matching constraints. *PVLDB*, 10(10):1082–1093, 2017.

[6] D. Deng, G. Li, S. Hao, J. Wang, and J. Feng. Massjoin: A mapreduce-based method for scalable string similarity joins. In *ICDE*, pages 340–351, 2014.

[7] D. Deng, G. Li, H. Wen, and J. Feng. An efficient partition based method for exact set similarity joins. *PVLDB*, 9(4):360–371, 2015.

[8] D. Deng, Y. Tao, and G. Li. Overlap set similarity joins with theoretical guarantees. In *SIGMOD*, pages 905–920, 2018.

[9] S. Helmer and G. Moerkotte. Evaluation of main memory join algorithms for joins with set comparison join predicates. In *VLDB*, pages 386–395, 1997.

[10] R. Jampani and V. Pudi. Using prefix-trees for efficiently computing set joins. In *DASFAA*, pages 761–772, 2005.

[11] A. Kunkel, A. Rheinländer, C. Schiefer, S. Helmer, P. Bouros, and U. Leser. Piejoin: Towards parallel set containment joins. In *SSDBM*, pages 11:1–11:12, 2016.

[12] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.

[13] Y. Luo, G. H. L. Fletcher, J. Hidders, and P. D. Bra. Efficient and scalable trie-based algorithms for computing set containment relations. In *ICDE*, pages 303–314, 2015.

[14] N. Mamoulis. Efficient processing of joins on set-valued attributes. In *SIGMOD*, pages 157–168, 2003.

[15] S. Melnik and H. Garcia-Molina. Divide-and-conquer algorithm for computing set containment joins. In *EDBT*, pages 427–444, 2002.

[16] S. Melnik and H. Garcia-Molina. Adaptive algorithms for set containment joins. *ACM Trans. Database Syst.*, 28:56–99, 2003.

[17] M. E. J. Newman. Power laws, Pareto distributions and Zipf's law. *Contemporary Physics*, 46:323–351, Sept. 2005.

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

[19] C. Roberts. Partial-match retrieval via the method of superimposed codes. *Proceedings of the IEEE*, 67(12):1624–1642, 1979.

[20] J. Sun, Z. Shang, G. Li, D. Deng, and Z. Bao. Dima: A distributed in-memory similarity-based query processing system. *PVLDB*, 10(12):1925–1928, 2017.

[21] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD*, pages 495–506, 2010.

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

[23] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW*, pages 131–140, 2008.

[24] J. Yang, W. Zhang, S. Yang, Y. Zhang, and X. Lin. Tt-join: Efficient set containment join. In *ICDE*, pages 509–520, 2017.

[25] J. Yang, W. Zhang, S. Yang, Y. Zhang, X. Lin, and L. Yuan. Efficient set containment join. *VLDB J.*, 27(4):471–495, 2018.

[26] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65, 2016.