The VLDB Journal
https://doi.org/10.1007/s00778-020-00644-3

REGULAR PAPER

Internal and external memory set containment join

Chengcheng Yang[2] · Dong Deng[1] · Shuo Shang[2] · Fan Zhu[2] · Li Liu[2] · Ling Shao[2]

Received: 31 May 2019 / Revised: 1 October 2020 / Accepted: 6 November 2020

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 process each partition separately. Each partition will be associated with a much smaller inverted index, and the set containment join cost can be significantly reduced. Moreover, to support large-scale datasets that are beyond the available memory space, we develop a novel adaptive data partition method that is designed to fully leverage the available memory and achieve high I/O efficiency, and thereby exhibiting outstanding performance for external memory set containment join. We evaluate our methods using both real-world and synthetic datasets. Experimental results demonstrate that our method outperforms state-of-the-art methods by up to 10× when the dataset is completely resided in memory. Furthermore, our approach achieves up to two orders of magnitude improvement on I/O efficiency compared with a baseline method when the dataset size exceeds the main memory space.

Keywords Set containment · Containment join · Query processing · Join algorithm · Similarity join

1 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 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 determines if there is an inclusion dependency between them. Moreover, set containment can be used for recommendations. For example, in the bookmark web service, if the URLs tagged by two users are modeled as sets, the set containment relationship indicates they have similar hobbies or interests and should

✉ Dong Deng
dong.deng@rutgers.edu

✉ Shuo Shang
jedi.shang@gmail.com

Chengcheng Yang
chengcheng.yang@kaust.edu.sa

Fan Zhu
shuo.shang@inceptioniai.org

Li Liu
shuo.shang@inceptioniai.org

Ling Shao
ling.shao@inceptioniai.org

[1] Rutgers University, New Brunswick, USA

[2] University of Electronic Science and Technology of China, Chengdu, China

Published online: 23 February 2021

introduce them to each other. 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 (R, 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 [59], existing methods can be broadly classified into union-oriented methods [16,27,30,31,35,58,59] and intersection-oriented methods [5,19,21,27,28].

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 R and 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, (R, S) is a candidate pair. Finally, the candidates are verified, and the join results obtained. Based on several recent studies [5,58], 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 increase in the signature size, while a small signature will result in many candidates and a high verification cost.

As an example, Helmer et al. [16] 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 ith bit of its signature bitmap to 1. Obviously, for any two sets R and 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 R in $\mathbb{R}$, they intersect all the inverted lists corresponding to the elements in R and obtain a list of sets from $\mathbb{S}$. For each S in the list, R is a subset of S and (R, 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 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 Sect. 3). A huge advantage of

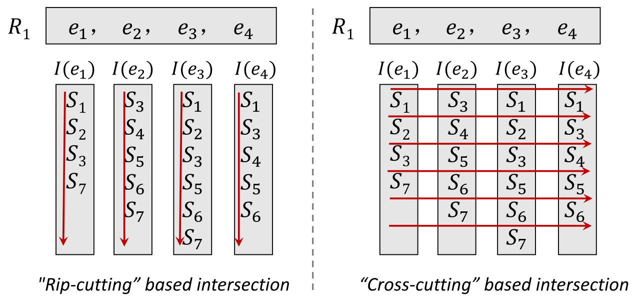

R_1
e_1, e_2, e_3, e_4
$I(e_1)$ $I(e_2)$ $I(e_3)$ $I(e_4)$
S_1 S_2 S_3 S_7
S_3 S_4 S_5 S_6 S_7
S_1 S_2 S_3 S_5 S_6 S_7
S_1 S_3 S_4 S_5 S_6
"Rip-cutting" based intersection
"Cross-cutting" based intersection

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

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 S_3 and S_7 in $\mathcal{I}[e_1]$ in Fig. 1. As S_4, S_5, and S_6 are not in $\mathcal{I}[e_1]$, e_1 does not exist in the three sets and R_1 cannot be a subset of S_4, S_5, or 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 prefix tree on $\mathbb{R}$ and extend the cross-cutting-based intersection to operate on the prefix 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 approach 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.
– To support large-scale datasets that do not fit into the available memory, we introduce an adaptive and I/O efficient data partition method that partitions all sets according to the memory budget.
– We conduct extensive experiments on both real-world datasets and synthetic datasets. The experimental results show that our approach is up to one order of magnitude faster than current state-of-the-art methods, and up to two orders of magnitude faster than the baseline method if the dataset cannot be entirely loaded into the available memory.

The rest of the paper is organized as follows: Section 2 defines the problem. Our set containment join framework is presented in Sect. 3. We discuss the tree-based methods in Sect. 4 and data partition in Sect. 5. Section 6 describes our external memory set containment join algorithm. Section 7 provides experimental results. We review related work in Sect. 8 and conclude in Sect. 9.

2 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 (R, S) such that $\mathsf{R} \subseteq \mathsf{S}$, where R and 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 1. 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.

3 The framework

In this section, we present our set containment join framework. The framework first builds an inverted index for the sets in $\mathbb{S}$ (Sect. 3.1). Then, it calculates the set containment join using the inverted index previously built (Sect. 3.2).

Table 1 A running example

Rid	R
(a) dataset $\mathbb{R}$	
R_1	$\{e_1, e_2, e_3, e_4\}$
R_2	$\{e_2, e_3, e_5\}$
R_3	$\{e_1, e_2, e_5, e_6\}$
Sid	**S**
(b) dataset $\mathbb{S}$	
S_1	$\{e_1, e_3, e_4, e_5, e_6\}$
S_2	$\{e_1, e_3, e_5\}$
S_3	$\{e_1, e_2, e_3, e_4, e_6\}$
S_4	$\{e_2, e_4, e_5, e_6\}$
S_5	$\{e_2, e_3, e_4, e_5, e_6\}$
S_6	$\{e_2, e_3, e_4, e_6\}$
S_7	$\{e_1, e_2, e_3, e_6\}$

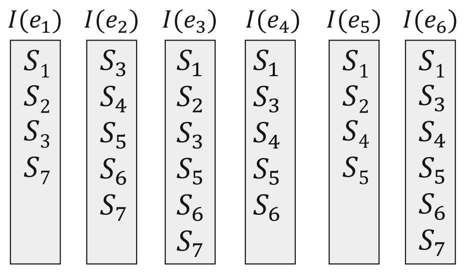


Fig. 2 The inverted index $\mathcal{I}$ for $\mathbb{S}$ in Table 1b

3.1 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 1(b). The inverted index can be constructed by sequentially reading the sets in $\mathbb{S}$ and, for each element e in S_i, appending S_i to $\mathcal{I}[e]$.

3.2 Set containment join framework

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 (R, 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 "candidate set" 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 S_j be the first entry greater than S_i in the list. Obviously, the element e must not exist in any 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., $\mathsf{S}_{i+1}, \ldots, \mathsf{S}_{j-1}$) in all the other inverted lists. To this end, we use S_j as the new candidate 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 S_j greater than 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 candidate 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 S_1 as the first candidate set to check.

Example 2 For example, consider the two datasets $\mathbb{R}$ and $\mathbb{S}$ in Table 1. For 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 candidate set to check to be S_1. Since S_1 is not found in $\mathcal{I}[e_2]$, it cannot be a superset of 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 S_1 are $\mathsf{S}_2, \mathsf{S}_3, \mathsf{S}_2$, and S_3, respectively. We use the largest one, S_3, as the next candidate set to check. Since 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 candidate set to check being S_7. However, 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.

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 candidate set MaxSid to check to be 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 (R, 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 candidate 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).

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 candidate 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 (R, 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 candidate 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 (R, S).

3.3 Early termination

In each round, our framework binary searches for a candidate 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

Algorithm 1: THE CROSS- CUTTING FRAMEWORK

```
   Input: $\mathbb{S}$ and $\mathbb{R}$: two collections of sets.
   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 an inverted index $\mathcal{I}$ for $\mathbb{S}$;
3      foreach $\mathsf{R} \in \mathbb{R}$ 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 $\mathcal{A}$;
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 $\mathcal{A}$;
```

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 candidate 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 S_1, S_3, and 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 candidate set S_1 is not found in $\mathcal{I}[e_2]$, we stop the current round and set the next candidate set to check to be the larger one between S_2 from $\mathcal{I}[e_1]$ and S_3 from $\mathcal{I}[e_2]$, which is S_3. As S_3 is found in all the inverted lists, we produce a result $(\mathsf{R}_1, \mathsf{S}_3)$. Then, the next candidate set to check is S_7. We terminate after binary searching 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 candidate set to check in each round. Nevertheless, it can save some unnecessary binary search operations if the candidate set MaxSid is not a superset of R, especially when the set size |R| is large.

4 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 Sect. 4.1

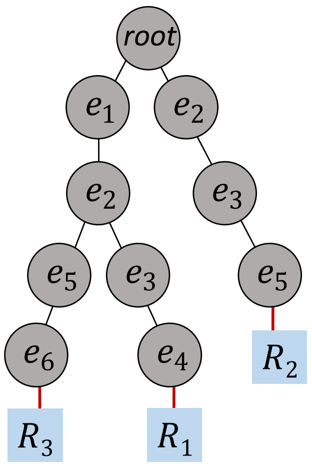


Fig. 3 The tree structure for $\mathbb{R}$ in Table 1a

and then present the tree-based method in Sect. 4.2. The tree-based method is essentially traversing the tree in postorder. Finally, we integrate the early termination technique into the tree-based method.

4.1 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 1(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.

4.2 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 candidate 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 $\mathcal{T}$, for each set R in $\mathbb{R}$, we propose to keep the candidate set to check in its corresponding leaf node n (denoted as n.MaxSid). For the inner node n, we use n.MaxSid to keep the smallest candidate set to check among all the leaf nodes in the subtree rooted at n. As we will see later in this section, this helps us update the candidate sets in the leaf nodes in a new round.

The next step in the framework is to produce a result if the candidate 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 candidate 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 (R, $\mathcal{T}$.root.MaxSid).

The last step is to update the candidate 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 candidate set of n as the largest gap of its ancestor nodes and itself (recall the gap of a node v is the first entry greater than the current candidate 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 1 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. 4a), the initial candidate sets for all the leaf nodes n_5, n_7, and n_{10} are S_1. For the inner nodes, based on the definition, their candidate sets are also S_1. We also have n_4.RidList $= \{\mathsf{R}_3\}$. This is because i) the corresponding leaf node n_5 of R_3 has the same candidate set as n_4 (i.e., n_5.MaxSid $= n_4$.MaxSid $= \mathsf{S}_1$) and ii) the candidate 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 Fig. 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 candidate 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. 4b, we update the candidate 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, S_3, S_2, S_3, and S_2, we update its candidate set

n_5.MaxSid as the largest one S_3. Similarly, we can update the candidate sets in the other nodes, which are all set to S_3. We also have n_1.RidList $= \{R_1\}$ as i) the corresponding leaf node n_7 of R_1 has the same candidate set as n_1 and ii) the candidate set n_1.MaxSid $= S_3$ 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 (R_1, S_3). Similarly, in the next round as shown in Fig. 4c, we find another result (R_2, S_5). In the last round, as shown in Fig. 4d, the candidate set for the root node is n_1.MaxSid $= S_\infty$, 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 based on the child nodes of n. On the one hand, n.MaxSid (or c.MaxSid) is defined as the smallest candidate set among 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 candidate set c.MaxSid among all its child nodes. For example, consider the node n_1 in Fig. 4c. It has two child nodes n_2 and n_8, whose candidate sets are, respectively, n_2.MaxSid $= S_7$ and n_8.MaxSid $= S_5$. Thus, we set n_1.MaxSid as the smaller one 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 candidate 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 ϕ 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 candidate 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, based on the definition, this leaf node must also exist in n.RidList. On the other hand, for any leaf node that is not in the above c.RidList, either its candidate set is not c.MaxSid or c.MaxSid does not exist in a certain node on the path from c to this leaf node, which indicates that this leaf node must also not exist in n.RidList. For example, consider the node n_3 in Fig. 4b. It has two child nodes n_4 and n_6 where n_4.RidList $= \phi$ and n_6.RidList $= \{R_1\}$. As n_3.MaxSid $= S_3$ exists in n_3's inverted list $\mathcal{I}[n_3.e]$, we have n_3.RidList $= n_4$.RidList $\cup$ n_6.RidList $= \{R_1\}$. As another example, consider the node n_3 in Fig. 4a. As n_3.MaxSid $= S_1$ 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 candidate sets 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 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 candidate set n.MaxSid as NextMax.

Note that when checking whether the candidate set n.MaxSid exists in the inverted list $\mathcal{I}[n.e]$ of a node n, the gap of n (i.e., the first entry in $\mathcal{I}[n.e]$ greater than the candidate set n.MaxSid) can be calculated simultaneously. To reuse this computation later when updating the candidate 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.

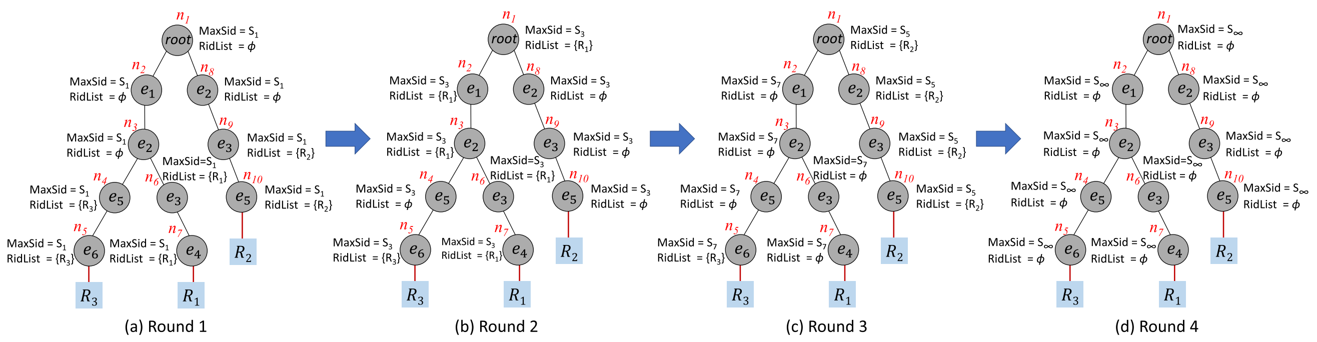


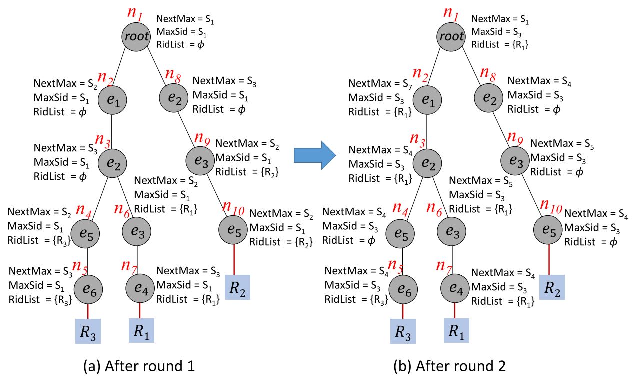
Fig. 4 A running example of the tree-based method based on the datasets in Table 1

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 $\mathcal{T}$ on $\mathbb{R}$;
3      build an inverted index $\mathcal{I}$ on $\mathbb{S}$;
4      for each node $n \in \mathcal{T}$, $n$.MaxSid = $n$.NextMax = $\mathsf{S}_1$;
5      while $\mathcal{T}$.root.MaxSid $\neq \mathsf{S}_\infty$ do
6          POSTORDERTRAVERSE($\mathcal{T}$.root, 1);
7          foreach R $\in \mathcal{T}$.root.RidList do
8              add (R, $\mathcal{T}$.root.MaxSid) to $\mathcal{A}$;
9      return $\mathcal{A}$;
```

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 $\leq$ NextMax do
4          POSTORDERTRAVERSE($c$, 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 $\mathcal{I}[n.e]$;
10     if Sid == $n$.MaxSid then
11         set $n$.NextMax as the entry next to Sid in $\mathcal{I}[n.e]$;
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 = $\phi$;
```



n_1 root NextMax = S_1 MaxSid = S_1 RidList = ϕ
n_2 e_1 NextMax = S_2 MaxSid = S_1 RidList = ϕ
n_8 e_2 NextMax = S_3 MaxSid = S_1 RidList = ϕ
n_3 e_2 NextMax = S_3 MaxSid = S_1 RidList = ϕ
n_9 e_3 NextMax = S_2 MaxSid = S_1 RidList = $\{R_2\}$
n_6 e_3 NextMax = S_2 MaxSid = S_1 RidList = $\{R_1\}$
n_4 e_5 NextMax = S_2 MaxSid = S_1 RidList = $\{R_3\}$
n_{10} e_5 NextMax = S_2 MaxSid = S_1 RidList = $\{R_2\}$
n_5 e_6 NextMax = S_3 MaxSid = S_1 RidList = $\{R_3\}$
n_7 e_4 NextMax = S_3 MaxSid = S_1 RidList = $\{R_1\}$
R_3 R_1 R_2
(a) After round 1
n_1 root NextMax = S_1 MaxSid = S_3 RidList = $\{R_1\}$
n_2 e_1 NextMax = S_7 MaxSid = S_3 RidList = $\{R_1\}$
n_8 e_2 NextMax = S_4 MaxSid = S_3 RidList = ϕ
n_3 e_2 NextMax = S_4 MaxSid = S_3 RidList = $\{R_1\}$
n_9 e_3 NextMax = S_5 MaxSid = S_3 RidList = ϕ
n_6 e_3 NextMax = S_5 MaxSid = S_3 RidList = $\{R_1\}$
n_4 e_5 NextMax = S_4 MaxSid = S_3 RidList = ϕ
n_{10} e_5 NextMax = S_4 MaxSid = S_3 RidList = ϕ
n_5 e_6 NextMax = S_4 MaxSid = S_3 RidList = $\{R_3\}$
n_7 e_4 NextMax = S_4 MaxSid = S_3 RidList = $\{R_1\}$
R_3 R_1 R_2
(b) After round 2

Fig. 5 Status of the prefix tree

The pseudo-code tree-based method is shown in Algorithm 2. It takes two datasets $\mathbb{R}$ and $\mathbb{S}$ as input and returns their set containment join results. The tree-based method first builds a prefix tree $\mathcal{T}$ for $\mathbb{R}$ and an inverted index $\mathcal{I}$ for $\mathbb{S}$ (Lines 2–3). Then, for each node n in the tree, it initializes both the candidate set n.MaxSid and the gap n.NextMax as 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 candidate set to check in all the leaf nodes. Moreover, it also gets $\mathcal{T}$.root.RidList updated accordingly. Based on the definitions, all the sets in $\mathcal{T}$.root.RidList are subsets of $\mathcal{T}$.root.MaxSid. Thus, we add a pair (R, $\mathcal{T}$.root.MaxSid) to the result $\mathcal{A}$ for each set R $\in \mathcal{T}$.root.RidList (Lines 7 to 8). The algorithm terminates when the smallest candidate set $\mathcal{T}$.root.MaxSid is identical to the "maximum" set S_∞, 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, for any child node c of n, the larger one of these two is the largest gap in all the ancestor nodes of c (Line 2). Then, the postorder traversing recursively invokes itself on the child nodes of n to get their candidate sets updated (Lines 3 and 4). Note that for the child node c of n whose candidate set c.MaxSid is larger than the largest gap NextMax, c.MaxSid does not need to be updated as this candidate 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 up-to-date. Next, it updates the candidate 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.MaxSid, 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 S_∞, i.e., n.NextMax = Sid = S_∞. For n.RidList, as discussed before, if Sid == n.MaxSid and n is a leaf node, which indicates the candidate set n.MaxSid exists in n and its inverted list $\mathcal{I}[n.e]$, 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 candidate 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 up-to-date.

Example 5 Consider the running example in Fig. 4. Figure 5a shows the status of the prefix tree after the first round of traversing. We discuss how this then transfers to the second round (as shown in Fig. 5b). 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 among n_5.NextMax, n_4.NextMax, n_3.NextMax, n_2.NextMax, and n_1.NextMax, which is n_3.NextMax $=$ S_3. Then, as n_5 is a leaf node, it updates n_5.MaxSid to be NextMax $=$ S_3. Next, it binary searches for the first entry in $\mathcal{I}[n_5.e = e_6]$ no smaller than S_3 and obtains Sid $=$ S_3. As Sid $==$ n.MaxSid, it sets n.NextMax to be the next entry after S_3 in $\mathcal{I}[e_6]$, which is S_4. It also sets n_5.RidList to be the corresponding set of n, which is R_3, because 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 candidate sets among all of its child nodes, which is n_5.MaxSid $=$ S_3. As the first entry no smaller than S_3 in $\mathcal{I}[n_4.e = e_5]$ is Sid $=$ S_4, which is not identical to n_4.MaxSid $=$ S_3, it sets n_4.NextMax as Sid $=$ S_4 and n_4.RidList $= \phi$. Similarly, it traverses and updates all the nodes in the prefix tree, and Fig. 5b 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 R $\in$ $\mathcal{T}$.root.RidList, the pair (R, $\mathcal{T}$.root.MaxSid) is a result. Next, we show the soundness of our method. As discussed before, for any set pair (R, S) in $\mathbb{R} \times \mathbb{S}$ where R $\subseteq$ S, all the inverted lists of R must have the entry S. Let n be the leaf node corresponding to R. Then, S exists in all the nodes on the path from $\mathcal{T}$.root to n and cannot be skipped. That is, the candidate 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 candidate set in the subtrees rooted at these ancestor nodes. 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.

4.3 Early termination for the tree-based method

In the framework, whenever the candidate 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 candidate set in the next round. Similarly, in the tree-based method, if a candidate set n.MaxSid is not found in the inverted list $\mathcal{I}[n.e]$ of a node n, the candidate 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 candidate set n.MaxSid is found in the inverted list $\mathcal{I}[n.e]$, will this procedure traverse to the parent node of n.

Algorithm 4: EarlyTermination

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

4.4 Avoiding the leaf nodes propagation

As discussed before, in each round, to update the sets in $\mathcal{T}$.root.RidList, the postorder tree traversing method needs to propagate the list of corresponding leaf nodes through every inner node up to the root node $\mathcal{T}$.root. That is, the list of leaf nodes is replicated in n.RidList of every inner node n on the path from $\mathcal{T}$.root to these leaf nodes, which would result in high propagation cost if the result size is large. To address this issue, we propose a "lazy" result generation strategy, which can eliminate the cost of leaf nodes propagation. In each round, instead of generating the join results in the current round, we postpone this process to the beginning of next round. When starting a new round, recall that the candidate set n.MaxSid of a leaf node n should be updated as the largest gap of its ancestor nodes and itself. In addition, this largest gap is kept in a variable NextMax (see Algorithm 3), which can be passed through the parent node to all its child nodes in the postorder tree traversing. Similarly, we propose to keep the returned superset $\mathcal{T}$.root.MaxSid of the last round in a variable ResSid. Then, in the new round, together with the variable NextMax, the variable ResSid will be passed to the leaf nodes whose sets are subsets of ResSid through the postorder tree traversing. Once a leaf node n is reached, we first check whether n.MaxSid $==$ ResSid. If so, we produce a result (R, ResSid), where R $\in$ n.RidList. Then, we proceed to update the variables n.MaxSid, n.RidList, and n.NextMax of that leaf node. For each inner node n, we drop the variable n.RidList and only update the variables n.MaxSid and

n.NextMax. In this way, the RidLists are only stored in leaf nodes and we do not need to propagate them up to the root node.

5 Data partitioning

In this section, we discuss our data partition methods. Section 5.1 shows how to partition the sets in $\mathbb{R}$ and constructs its corresponding inverted index. Section 5.2 proposes to use two different methods to process the partitions.

5.1 Partitioning the sets

In this section, we further improve the efficiency and scalability of the tree-based method by partitioning the data. The basic idea is that we can first partition all 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 $\mathbb{S}$ must not be a superset of any set R in this partition. After that, we can use the previous tree-based method to process each partition with its corresponding local inverted index to get all the results in this partition. Together, we can get all the set containment join result. The local inverted index is much smaller than the original inverted index, and thus, the set containment join cost can be significantly reduced.

For this purpose, in this paper, we propose to partition the sets in $\mathbb{R}$ by their smallest elements in the global order.[1] Each partition, denoted by $\mathbb{R}_e$, consists of all the sets whose smallest elements are e. In this way, each set in $\mathbb{R}$ is allocated into one and only one partition. For example, consider the dataset $\mathbb{R}$ in Table 1. The smallest elements in R_1, R_2, and 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 $\mathbb{R}_{e_2} = \{\mathsf{R}_2\}$. Next, we deal with the sets in $\mathbb{S}$. 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 must not 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 the results in this partition.

Example 6 Consider the datasets in Table 1. 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 the four sets S_1, S_2, S_3, and S_7 containing e_1. For the partition $\mathbb{R}_{e_2}$, we construct another inverted index $\mathcal{I}_{e_2}$ with the five sets S_3, S_4, S_5, S_6, and 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 Figs. 2 and 4, the average inverted list length of the original inverted index $\mathcal{I}$ is 5, while the average length of local inverted index $\mathcal{I}_{e_1}$ is only 2.8 for the left subtree (because the lengths of $\mathcal{I}_{e_1}[e_1]$, $\mathcal{I}_{e_1}[e_2]$, $\mathcal{I}_{e_1}[e_3]$, $\mathcal{I}_{e_1}[e_4]$, $\mathcal{I}_{e_1}[e_5]$, and $\mathcal{I}_{e_1}[e_6]$ are 4, 2, 4, 2, 2, and 3, respectively) if 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 $\mathbb{R}$, i.e., the subtree rooted at the child node of $\mathcal{T}$.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, set containment join cost using the tree-based method decreases for each partition. However, for some very small partitions, the overhead of constructing the local inverted index may be even larger than the cost of directly applying the tree-based method on the original inverted index. In next section, we discuss how to determine which inverted index to use for each partition.

5.2 Processing the partitions

As all the sets in $\mathbb{S}$ containing the element e are within $\mathcal{I}[e]$, we use all the sets in $\mathcal{I}[e]$ to construct the local inverted index $\mathcal{I}_e$, the same as how we construct $\mathcal{I}$ using $\mathbb{S}$. Notice that the local inverted index construction cost may be even larger than the benefit of replacing the original inverted index with the local inverted index in the tree based method. Thus, we need to dynamically determine whether to construct and use the local inverted index when processing each partition separately. Next, we give a high-level analysis of the cost and benefit of constructing and using a local inverted index.

The set containment join cost is proportional to the product of the sizes of the two input collections of sets. For a partition $\mathbb{R}_e$, using the local inverted index in our tree-based method is to perform the join over the sets in $\mathbb{R}_e$ and $\mathcal{I}[e]$. Thus, the set containment join cost of using the local inverted index $\mathcal{I}_e$ is proportional to $|\mathbb{R}_e| \times |\mathcal{I}[e]|$. In contrary, using the original inverted index $\mathcal{I}$ is to perform the join over the sets in $\mathbb{R}_e$ and $\mathbb{S}$ and the set containment cost is proportional to $|\mathbb{R}_e| \times |\mathbb{S}|$. Thus, the benefit of using the local inverted index is proportional to $|\mathbb{R}_e| \times (|\mathbb{S}| - |\mathcal{I}[e]|)$. On the other hand, the cost of constructing the local inverted index $\mathcal{I}_e$ is proportional to the number of sets in $\mathbb{S}$ containing e, which is $|\mathcal{I}[e]|$.

Choosing from two indexes. Clearly, it is hard, if not impossible, to accurately estimate the benefit and cost of using the local inverted index for a partition. However, we can analyze their trends with the growth of partition sizes, and it will guide us to choose from the two inverted indexes to use. Consider a partition $\mathbb{R}_e$, as the constant $|\mathbb{S}|$ is usually

[1] In our implementation, we use the element frequency order as the global order and use the most frequent element to partition the data.

far larger than $|\mathcal{I}[e]|$, the benefit, which is proportional to $|\mathbb{R}_e| \times (|\mathbb{S}| - |\mathcal{I}[e]|)$, is dominated by the partition size $|\mathbb{R}_e|$. At the same time, as the inverted list length $|\mathcal{I}[e]|$ is proportional to the partition size $|\mathbb{R}_e|$, the cost is also proportional to the partition size $|\mathbb{R}_e|$. However, with the increase in the partition size, the benefit grows much faster than the cost due to the large slope $|\mathbb{S}|$ in the benefit and the benefit would ultimately exceeds the cost. Thus, we tend to construct and use the local inverted index for large partitions.

For a small partition $\mathbb{R}_e$, there is little benefit and the cost may even be larger than the benefit, in which case we should directly use the original inverted index $\mathcal{I}$ in the tree-based method instead of constructing and using a local inverted index $\mathcal{I}_e$. Note that, in many datasets, the element frequency follows a power-law distribution. There are a few frequent elements and enormous infrequent elements. Since we partition the sets in $\mathbb{R}$ by their smallest elements, we will have a few large partitions and a huge number of small partitions. Though, for small partitions, the set containment join cost using either inverted index is negligible compared to that for large partitions, there are a huge number of small partitions and the local inverted index construction cost would add up and affect the performance to a large extent. Thus, for small partitions, we propose to use the original inverted index in the tree-based method instead of constructing and using a local inverted index.

Dichotomize small and large partitions. To judiciously dichotomize the small and large partitions, in our implementation, we use a simple sampling method. We first sort all the partitions by the ascending order of their sizes and then randomly sample some partitions based on an empirically determined sample rate. Next, we sequentially process each sampled partitions by using the local inverted index and the original inverted index, respectively. We can afford to do so because the set containment join cost is negligible using either inverted index for very small partitions and the number of sample partitions is small. Once for the first time using the local inverted index consistently outperforms using the original inverted index for several times[2], we terminate the sampling process and use the first partition of them as the size boundary to dichotomize small and large partitions. For small partitions, we use the original inverted index, and for larger partitions, we construct and use the local inverted index.

6 External memory set containment join

This section discusses how to deal with large-scale datasets that are beyond the available memory space. We first introduce the basic idea of our external memory set containment join method in Sect. 6.1 and then design an adaptive data partitioning method that is tailored for high I/O efficiency in Sect. 6.3.

6.1 Memory-constraint data partitioning

The basic idea. For external memory set containment join, the I/O cost dominates the CPU cost. Thus, we focus on improving the I/O efficiency. The basic idea of our external memory method is to partition the sets in $\mathbb{R}$ into partitions with proper sizes so that each partition can fit in the main memory while maximizing the memory usage. Note that each in-memory partition consists of two indispensable components for set containment join processing: a local prefix tree built by all sets in the partition and a local inverted index constructed by the part of sets in $\mathbb{S}$ that might be supersets of a set R in the partition. To this end, all the sets in $\mathbb{R}$ are partitioned by one or more smallest elements in the global order. Initially, we partition the sets in $\mathbb{R}$ by their smallest elements. If a partition cannot be entirely loaded into the main memory, we split it into multiple smaller partitions by their next smallest elements recursively until each partition can fit into the memory individually.

Data layout and loading. To efficiently implement the above idea, we first sort all the sets in $\mathbb{R}$ in ascending lexicographical order and store them continuously on the disk. The sets in $\mathbb{S}$ and the inverted index $\mathcal{I}$ are also stored on disk (without particular order). Before conducting set containment joins on a partition $\mathbb{R}_{fit}$ that can fit in memory, we need to perform a few data loading steps. First, to build the local prefix tree, all the sets in $\mathbb{R}_{fit}$ are loaded into main memory. Since these sets are stored continuously on disk, this step can be achieved by several sequential I/O reads. Second, to construct the corresponding local inverted index $\mathcal{I}_{fit}$, we first need to retrieve all the sets in $\mathbb{S}$ that might be a superset of a set in $\mathbb{R}_{fit}$. By definition, the elements in the common prefix of the local prefix tree appear in all sets in $\mathbb{R}_{fit}$. Thus, for a set S in $\mathbb{S}$ to be a superset of any set in $\mathbb{R}_{fit}$, S must contain all the elements in the common prefix. Based on this, in the second step, we retrieve from disk the inverted lists corresponding to the elements in the common prefix. The intersection of these inverted lists is exactly the part of sets in $\mathbb{S}$ containing all the elements in the common prefix, which we denote as the *target-sets* of this partition. Finally, the local inverted index is constructed by loading all the *target-sets* of this partition from disk.

The primary I/O cost above is embodied in loading the *target-sets* in $\mathbb{S}$. This is because, in the first step, each set in $\mathbb{R}$ is loaded only once and they are read sequentially from the disk. In the second step, the I/O cost of loading inverted lists is relatively small since the common prefix is often very short and only a few inverted lists are loaded for each partition.

[2] In the experiment, we empirically set it to 3.

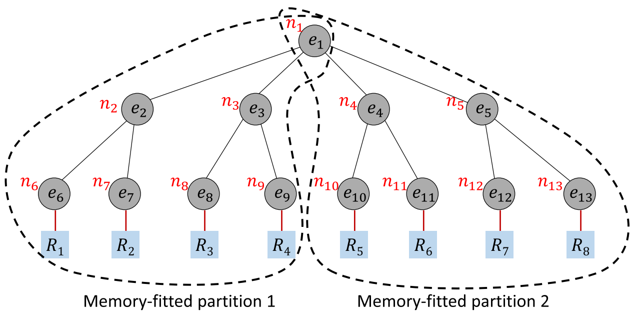


Fig. 6 An example of memory constrained partition

On the contrary, in the third step, the *target-sets* could be numerous and induce many random I/Os. Note that loading a *target-set* or an inverted list needs a random read.

Union small partitions. To reduce the I/O cost, we propose to union multiple small partitions into a larger memory-fitted partition. This is because these small partitions may share some *target-sets*. We only need to load the common *target-sets* into memory once by union these small partitions. We will discuss this in detail in Sect. 6.3.

Example 7 Consider the prefix tree for $\mathbb{R} = \{\mathsf{R}_1, \ldots, \mathsf{R}_8\}$ in Fig. 6. Assuming that the size of each tree node is 1, the size of the inverted index $\mathcal{I}_{e_1}$ is 17, and the memory budget is 19. Obviously, the prefix tree and the inverted index cannot be entirely loaded in memory. Thus, we partition the sets in $\mathbb{R}$ by their two smallest elements and get four partitions $\{\mathsf{R}_1, \mathsf{R}_2\}$, $\{\mathsf{R}_3, \mathsf{R}_4\}$, $\{\mathsf{R}_5, \mathsf{R}_6\}$, and $\{\mathsf{R}_7, \mathsf{R}_8\}$. The sizes of the four corresponding local prefix trees are all 4. Suppose the sizes of the four corresponding local inverted index are all 5. Then, the size of each partition is 9. To maximize the usage of memory and reduce I/O cost, we can union the 4 small partitions into 2 memory-fitted partitions $\{\mathsf{R}_1, \mathsf{R}_2, \mathsf{R}_3, \mathsf{R}_4\}$ and $\{\mathsf{R}_5, \mathsf{R}_6, \mathsf{R}_7, \mathsf{R}_8\}$, as shown in Fig. 6 in the dotted lines. The sizes of the two local prefix trees and the two local inverted indexes are 7 and 10 at most.

Challenges. Implementing the above ideas imposes two challenges: First, it requires effective monitoring on the sizes of a local prefix tree and its corresponding local inverted index to get partitions that can be fitted into the available memory; second, how to adaptively partition the prefix tree and $\mathbb{S}$ to efficiently compute the set containment join correctly and completely. In Sect. 6.3, we devise an adaptive data partition method that utilizes incremental data loading and online partition size monitoring to address the two challenges.

6.2 Size estimation

In this section, we discuss how to estimate the sizes of the local inverted index and the local prefix tree for a given partition. Before getting into the details, we first formally define the partition.

Definition 2 The partition $\mathbb{R}_{e_{i_1}\cap\cdots\cap e_{i_m}\cap\{e_{j_1}\cup\cdots\cup e_{j_n}\}}$ consists of the sets in $\mathbb{R}$ that have $e_{i_1}, \ldots, e_{i_m}$ as the first m smallest elements and have one of the elements in $\{e_{j_1}, \ldots, e_{j_n}\}$ as the $(m+1)$th smallest element.

We denote the local inverted index and local prefix tree of a partition $\mathbb{R}_{e_{i_1}\cap\cdots\cap e_{i_m}\cap\{e_{j_1}\cup\cdots\cup e_{j_n}\}}$, respectively, as $\mathcal{I}_{e_{i_1}\cap\cdots\cap e_{i_m}\cap\{e_{j_1}\cup\cdots\cup e_{j_n}\}}$ and $\mathcal{T}_{e_{i_1}\cap\cdots\cap e_{i_m}\cap\{e_{j_1}\cup\cdots\cup e_{j_n}\}}$. For a set S in $\mathbb{S}$ to be a superset of any set in the partition $\mathbb{R}_{e_{i_1}\cap\cdots\cap e_{i_m}\cap\{e_{j_1}\cup\cdots\cup e_{j_n}\}}$, S must contain all elements in $\{e_{i_1}, \ldots, e_{i_m}\}$ and at least one element in $\{e_{j_1}, \ldots, e_{j_n}\}$. Thus, to obtain the *target-sets* of a partition $\mathbb{R}_{e_{i_1}\cap\cdots\cap e_{i_m}\cap\{e_{j_1}\cup\cdots\cup e_{j_n}\}}$, we can first union all the inverted lists $\mathcal{I}[e_{j_1}], \ldots, \mathcal{I}[e_{j_n}]$ and then intersect the union result with all the inverted lists $\mathcal{I}[e_{i_1}], \ldots, \mathcal{I}[e_{i_m}]$. That is, $\mathcal{I}[e_{i_1}] \cap \cdots \cap \mathcal{I}[e_{i_m}] \cap (\mathcal{I}[e_{j_1}] \cup \cdots \cup \mathcal{I}[e_{j_n}])$.

Estimating the local inverted index size. Given a partition, we estimate its corresponding local inverted index size as the multiplication of the number of *target-sets* of this partition and the average size of sets in $\mathbb{S}$. Next, we estimate the number of *target-sets* of a partition.

We first estimate the intersection size of multiple inverted lists. Let $P(e \in \mathsf{S})$ be the probability of a set S in $\mathbb{S}$ appearing in the inverted list $\mathcal{I}[e]$. Assume that each element e is evenly distributed across all sets in $\mathbb{S}$, then $P(e \in \mathsf{S})$ can be estimated as the proportion of the sets in $\mathbb{S}$ that contain the element e, whose number is $|\mathcal{I}[e]|$. Thus, we have

$$P(e \in \mathsf{S}) = \frac{|\mathcal{I}[e]|}{|\mathbb{S}|}$$

where $|\mathbb{S}|$ is the number of sets in $\mathbb{S}$.

Further, we assume that the appearance of the elements in the set in $\mathbb{S}$ is independent. That is to say, the appearance of a set S in each of the inverted lists is mutually independent. Thus, the probability of S appears in all of the inverted lists $\mathcal{I}[e_{i_1}], \ldots, \mathcal{I}[e_{i_m}]$ is $P(e_{i_1} \in \mathsf{S}) \times \cdots \times P(e_{i_m} \in \mathsf{S})$. Under this, we have the expected intersection size of multiple inverted lists as:

$$\begin{aligned}
&\mathbb{E}(|\mathcal{I}[e_{i_1}] \cap \cdots \cap \mathcal{I}[e_{i_m}]|) \\
&\quad = \sum_{\mathsf{S}\in\mathbb{S}} P(e_{i_1} \in \mathsf{S}) \times \cdots \times P(e_{i_m} \in \mathsf{S}) \\
&\quad = \sum_{\mathsf{S}\in\mathbb{S}} \frac{|\mathcal{I}[e_{i_1}]|}{|\mathbb{S}|} \times \cdots \times \frac{|\mathcal{I}[e_{i_m}]|}{|\mathbb{S}|} \\
&\quad = \frac{\prod_{k=1}^{m} |\mathcal{I}[e_{i_k}]|}{|\mathbb{S}|^{m-1}}.
\end{aligned}$$

Next, we estimate the union size of multiple inverted lists $\mathcal{I}[e_{j_1}], \ldots, \mathcal{I}[e_{j_n}]$. Since our goal is to make the local inverted index fit in memory, we estimate an upper bound of the union size, which is achieved when the inverted lists do not share any set. Thus, the expected union size of multiple inverted lists is

$$\mathbb{E}(|\mathcal{I}[e_{j_1}] \cup \cdots \cup \mathcal{I}[e_{j_n}]|) \leq \sum_{k=1}^{n} |\mathcal{I}[e_{j_k}]|.$$

As described above, the appearance of a set S in each of the inverted lists is mutually independent. Similarly, for each set S in the union $\mathcal{I}[e_{j_1}] \cup \cdots \cup \mathcal{I}[e_{j_n}]$, the probability of it appearing in all of the inverted lists $\mathcal{I}[e_{i_1}], \ldots, \mathcal{I}[e_{i_m}]$ is $P(e_{i_1} \in \mathsf{S}) \times \cdots \times P(e_{i_m} \in \mathsf{S})$. Thus, we have

$$\begin{aligned}
&\mathbb{E}(|\mathcal{I}[e_{i_1}] \cap \cdots \cap \mathcal{I}[e_{i_m}] \cap (\mathcal{I}[e_{j_1}] \cup \cdots \cup \mathcal{I}[e_{j_n}])|) \\
&= \sum_{\mathsf{S} \in \mathcal{I}[e_{j_1}] \cup \cdots \cup \mathcal{I}[e_{j_n}]} P(e_{i_1}) \times \cdots \times P(e_{i_m}) \\
&= \sum_{\mathsf{S} \in \mathcal{I}[e_{j_1}] \cup \cdots \cup \mathcal{I}[e_{j_n}]} \frac{\prod_{k=1}^{m} |\mathcal{I}[e_{i_k}]|}{|\mathbb{S}|^m} \\
&\leq \frac{\prod_{k=1}^{m} |\mathcal{I}[e_{i_k}]| \times \sum_{k=1}^{n} |\mathcal{I}[e_{j_k}]|}{|\mathbb{S}|^m}.
\end{aligned}$$

After estimating the number of *target-sets* for a local inverted index, the expected size of the local inverted index can be obtained by multiplying the number with the average size of the sets in $\mathbb{S}$, which can be calculated as:

$$|\mathsf{S}|_{avg} = \frac{\Sigma_{\mathsf{S} \in \mathbb{S}} |\mathsf{S}|}{|\mathbb{S}|}.$$

Together, we have the expected size of the local inverted index $\mathcal{I}_{e_{i_1} \cap \cdots \cap e_{i_m} \cap \{e_{j_1} \cup \cdots \cup e_{j_n}\}}$ as:

$$\begin{aligned}
&\mathbb{E}(|\mathcal{I}_{e_{i_1} \cap \cdots \cap e_{i_m} \cap \{e_{j_1} \cup \cdots \cup e_{j_n}\}}|) \\
&= \mathbb{E}(|\mathcal{I}[e_{i_1}] \cap \cdots \cap \mathcal{I}[e_{i_m}] \cap (\mathcal{I}[e_{j_1}] \cup \cdots \cup \mathcal{I}[e_{j_n}])|) \times |\mathsf{S}|_{avg} \\
&= \Sigma_{\mathsf{S} \in \mathbb{S}} |\mathsf{S}| \times \frac{\prod_{k=1}^{m} |\mathcal{I}[e_{i_k}]| \times \sum_{k=1}^{n} |\mathcal{I}[e_{j_k}]|}{|\mathbb{S}|^{m+1}}.
\end{aligned}$$

Note that $|\mathbb{S}|$ and $\sum_{\mathsf{S} \in \mathbb{S}} |\mathsf{S}|$ are constant for any local inverted index in the formula above. To estimate the size of a local inverted index using the formula above, we keep a lookup table of the sizes of all inverted lists. Then, the size of a local inverted index can be estimated easily from the common elements $e_{i_1}, \ldots, e_{i_m}$ and $e_{j_1}, \ldots e_{j_n}$. The size of prefix tree can be easily estimated by the number of nodes in the tree and the associated sets in the leaf nodes.

6.3 Incremental data partitioning

To implement the basic idea described in Sect. 6.1, we propose an incremental data partitioning method EXLCJoin. This method incrementally loads data from disk and maintains a local prefix tree based on the data in memory until the estimated sizes of the local prefix tree and local inverted index exceed the memory budget, at which point the local inverted index is materialized and a set containment join is conducted.

More specifically, we load the sets in $\mathbb{R}$ in lexicographical order to the main memory, one page at a time. At the same time, we maintain a local prefix tree $\mathcal{T}_{mem}$ in the main memory based on loaded sets in $\mathbb{R}$. Let n_{lca} be the lowest common ancestor (LCA) of all leaf nodes in $\mathcal{T}_{mem}$. Based on the discussion in Sect. 6.2, the estimation of the local inverted index size depends on n_{lca} and its child nodes. Each time we add the newly loaded sets to $\mathcal{T}_{mem}$, we re-estimate the local inverted index size only if n_{lca} changes or its child nodes changes. If the memory budget is exceeded by the estimated sizes of the local inverted index and the local prefix tree, we drop the newly loaded sets and regard the sets in $\mathcal{T}_{mem}$ as a partition. Then, we load the *target-sets* from disk, materialize the local inverted index, and conduct set containment join on this partition.

Algorithm 5 illustrates the details of our external memory set containment join algorithm. The algorithm first sorts sets in $\mathbb{R}$ lexicographically (Line 2) and builds an inverted index $\mathcal{I}$ for the sets in $\mathbb{S}$ (Line 3). Then, it incrementally loads the sets in $\mathbb{R}$ and continuously monitors the memory usage (Lines 5–10). If the memory required to process this partition approaches the memory budget, it loads the *target-sets* in $\mathbb{S}$ from the disk and build a local inverted index (Line 14). Then, it processes the partition in memory (Line 15). More specifically, two steps are included to get the *target-sets* in $\mathbb{S}$. At first, it unites the inverted lists corresponding to the elements of all n_{lca}'s child nodes. Then, it gets the *target-sets* in $\mathbb{S}$ based on the intersection of the union list with all the inverted lists corresponding to the elements of n_{lca} and all its ancestors. Note that there may be no common prefixes between small partitions, then the union list is directly used to get the *target-sets* in $\mathbb{S}$. Additionally, the algorithm loads *target-sets* in $\mathbb{S}$ in an increasing order of their positions on the disk, which takes advantage of fast sequential accesses of disks and further reduce the overall I/O cost. Finally, after processing the partition, the algorithm removes the partition and the local prefix tree from memory (Line 16) and continues loading sets in $\mathbb{R}$ incrementally.

Example 8 Consider the prefix tree for $\mathbb{R} = \{\mathsf{R}_1, \ldots, \mathsf{R}_8\}$ in Fig. 6, we use the same assumption as Example 7, and then, we have $|\mathcal{I}_{e_1}| = 17$ and $|\mathcal{I}_{e_1 \cap e_i}| = 5\,(i = 2, \ldots, 5)$. After loading R_1 and R_2, n_2 is exactly the lowest common ancestor n_{lca}. As the memory requirement of processing the

Algorithm 5: EXTERNAL MEMORY SCJ

```
Input: $\mathbb{S}$ and $\mathbb{R}$; $M$: Memory Budget.
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      sort all sets in $\mathbb{R}$ in ascending lexicographical order and write them to disk;
3      build an inverted index $\mathcal{I}$ on $\mathbb{S}$ and write it to disk;
4      init an empty in-memory prefix tree $\mathcal{T}_{mem}$;
5      foreach set $\mathsf{R} \in \mathbb{R}$ do
6          add R to $\mathcal{T}_{mem}$;
7          update the size of prefix tree $|\mathcal{T}_{mem}|$;
8          if $n_{lca}$ or the set of its child nodes changes then
9              $n_{lca}$ = $\mathcal{T}_{mem}$'s lowest common ancestor;
10             re-estimate the size of the local inverted index $|\mathcal{I}(\mathcal{T}_{mem})|$;
11         if $|\mathcal{T}_{mem}| + |\mathcal{I}(\mathcal{T}_{mem})| \geq M$ then
12             remove R from $\mathcal{T}_{mem}$;
13             retrieve the inverted lists corresponding to $n_{lca}$, all its ancestor nodes, and all its child nodes to calculate the target-sets;
14             retrieve all the target-sets to construct the local inverted index for $\mathcal{T}_{mem}$;
15             conduct set containment join on $\mathcal{T}_{mem}$ and the local inverted index and add the results to $\mathcal{A}$;
16             set $\mathcal{T}_{mem}$ as an empty tree and drop the local inverted index;
17     return $\mathcal{A}$;
```

in-memory partition is 9 (4 for the prefix tree and 5 for the local inverted index $\mathcal{I}_{e_1 \cap e_2}$), which is smaller than the memory budget (i.e., 19), then we continue loading the sets in $\mathbb{R}$. When adding R_3, the n_{lca} is updated to n_1. Then, we re-estimate the size of the local inverted index $\mathcal{I}_{e_1 \cap (e_2 \cup e_3)}$. The memory requirement is 16 (6 for the prefix tree and 10 for the local inverted index), which is still smaller than the memory budget; then, the loading process is continued. After loading R_5, a new child n_4 is added to the n_{lca}. Similarly, we need to re-estimate the size of the local inverted index $\mathcal{I}_{e_1 \cap (e_2 \cup e_3 \cup e_4)}$. We find that the memory requirement increases to 24 (9 for the prefix tree and 15 for the local inverted index) and exceeds the memory budget. Thus, we remove R_5 from the in-memory prefix tree $\mathcal{T}_{mem}$, and regard the in-memory sets $\{\mathsf{R}_1, \mathsf{R}_2, \mathsf{R}_3, \mathsf{R}_4\}$ as a partition. Thereafter, we materialize the local inverted index and process the partition. Finally, we clean up the memory by emptying $\mathcal{T}_{mem}$ and dropping $\mathcal{I}(\mathcal{T}_{mem})$, and then continue loading the set R_5.

Correctness and soundness. We first show the correctness of the incremental data partitioning-based method EXLCJoin. The correctness is obvious as EXLCJoin uses the tree-based method to process each memory-fitted partition, which returns a pair (R, $\mathcal{T}_{mem}$.root.MaxSid) only if the candidate set $\mathcal{T}_{mem}$.root.MaxSid is found in all the nodes (recall that a set S exists in a node n means S exists in the inverted list $\mathcal{I}[n.e]$) on the path from $\mathcal{T}_{mem}$.root to R's corresponding node n_{R}. This indicates a set containment relationship between R and $\mathcal{T}_{mem}$.root.MaxSid. Next, we show the soundness of EXLCJoin. As discussed in Sect. 4, for any set pair (R, S) in $\mathbb{R} \times \mathbb{S}$ where $\mathsf{R} \subseteq \mathsf{S}$, S must exist in all the nodes on the path from $\mathcal{T}_{mem}$.root to R's corresponding node n_{R} and will not be skipped by the tree-based method. That is, the candidate set S must be put in all the inverted lists of R. Suppose the memory-fitted partition that contains R is $\mathbb{R}_{e_{i_1} \cap \cdots \cap e_{i_m} \cap \{e_{j_1} \cup \cdots \cup e_{j_n}\}}$ and R has $e_{i_1}, \ldots, e_{i_m}$ as the first m smallest elements and has $e_{j_k} (1 \leq k \leq n)$ as the $(m+1)$th smallest element. Then, S must contain $e_{i_1}, \ldots, e_{i_m}$ and e_{j_k} since $\mathsf{R} \subseteq \mathsf{S}$. On the other hand, the algorithm loads all the sets in $\mathbb{S}$ that contain all elements in $\{e_{i_1}, \ldots, e_{i_m}\}$ and at least one element in $\{e_{j_1}, \ldots, e_{j_n}\}$ from the disk as the *target-sets*, which are then used to construct the local inverted index. Clearly, S belongs to the *target-sets*. Therefore, S will be put in the inverted lists corresponding to each of its distinct elements, which includes R's inverted lists as $\mathsf{R} \subseteq \mathsf{S}$.

7 Experiment

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

7.1 Internal memory evaluations

7.1.1 Experimental setup

We conducted experiments on both real-world and synthetic datasets. As with the previous studies [5,58], 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 eight real-world datasets: FLICKR,[3] AOL,[4] ORKUT,[5] TWITTER,[6] DBLP,[7] LINKEDIN,[8] AMAZON[9] and DELICI.[10] The FLICKR dataset is a photo-tag dataset. Each photograph corresponds to a set, and each photograph 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

[3] https://www.flickr.com.
[4] http://www.cim.mcgill.ca/~dudek/206/Logs/.
[5] https://snap.stanford.edu/data/com-Orkut.html.
[6] https://snap.stanford.edu/data/twitter-2010.html.
[7] https://www.aminer.cn/citation.
[8] https://www.aminer.cn/data-sna.
[9] http://jmcauley.ucsd.edu/data/amazon/links.html.
[10] http://konect.uni-koblenz.de/networks/delicious-ui.

Table 2 Statistics of the real-world datasets

Dataset	# of sets	Min/max/avg size	# of Elements	z-value	# of results
FLICKR	3,546,729	1/1230/5.4	618,971	0.63	6.27 billion
AOL	36,389,577	1/125/2.5	3,849,556	0.68	1341.87 billion
ORKUT	15,301,901	2/9120/7	2,322,299	0.13	240.4 million
TWITTER	28,819,434	2/4998/9	13,096,918	0.3	545.14 million
DBLP	2,423,403	1/1532/10.3	1,985,921	0.44	10.02 million
LINKEDIN	3,099,144	1/869/6.2	6,447,707	0.06	3.97 million
AMAZON	10,270,965	2/44,557/7	9,760,640	0.43	14.87 million
DELICI	7,333,910	2/29,233/10.2	760,414	0.3	22.49 million

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. The DBLP dataset consists of citation relationships extracted from the DBLP bibliography database. Each paper corresponds to a set and each reference of the paper corresponds to an element. In this case, we create containment relationships between papers, which could be used for further studies on paper recommendations. The LINKEDIN dataset is a professional network dataset. Each user corresponds to a set. The professional connections of a user are the elements of the corresponding set. Set containment join on this dataset can help people find professionals in related area of their work. The AMAZON dataset contains user ratings from the e-commerce site Amazon. Each user is a set, while each product rated by the user is an element. The DELICI dataset is a bookmark-tag dataset. Each user corresponds to a set, and each URL tagged by the user corresponds to an element. For these two datasets, set containment join can help people find friends with similar interests and hobbies. Note that in AMAZON and DELICI datasets, there are a large number of users with only one product rating or URL tag. We removed these users from the datasets, because their supersets can be directly obtained from the inverted lists corresponding to the element in each user. Table 2 provides some statistics for these eight datasets.

As with the previous work [59], we make use of *Zipf's* law [33] 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. 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,

Table 3 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

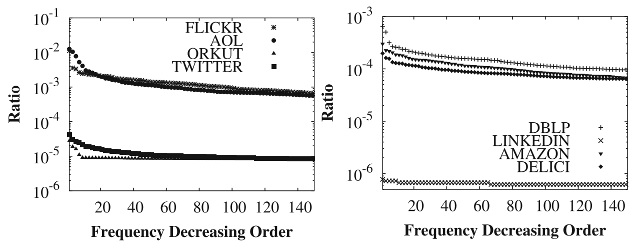


Fig. 7 Frequency distribution of real-world datasets

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 3 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 [59] 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 [33].

For the eight real-world datasets, Fig. 7 shows the percentage of the total number of elements that the top 150 most frequent elements account for. We can see that the eight datasets cover a wide range of data skews. Additionally, we can also notice that the most frequent elements in the FLICKR and AOL datasets account for much higher percentage (about $10\times$–$10,000\times$) of the total elements than those of the other datasets, which indicates that the FLICKR and AOL datasets are more skew than the other datasets.

We compared LCJoin with three state-of-the-art algorithms, PRETTI [19], LIMIT+ [5] and TT-Join [58,59]. 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 [59]. In addition, when performing samples in LCJoin to determine the partition size boundary, the sample fraction was set to 1%. In all implementations, the element frequency order was used as the global order.

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.

7.1.2 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 Sect. 3.3. (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 Sect. 4.3.

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 8 shows the experimental results on the eight 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× 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 was 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. Note that on the LINKEDIN dataset, the performance of framework-based methods was slightly better than the tree methods. This is because the LINKEDIN dataset has a relatively even data distribution (z-value = 0.06), which indicates that there are not much computation can be shared. As a result, the benefit of shared computation is exceeded by the overhead of prefix tree. We also observed that the early termination techniques helped improve the performance. This is because they can avoid unnecessary binary search operations.

7.1.3 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 partitions. (2) LCJoin uses the method as described in Sect. 5 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 that TreeBasedET does not partition the data. The results on the real-world datasets are shown in Fig. 9. 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 was 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 on some datasets, the partition-based method AllPartition did not perform as well as the non-partition method TreeBasedET. For example, on the dataset AMAZON in which there exist a large number of small partitions, AllPartition had 2.8× more run time than TreeBasedET when the data cardinality was 100%. 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. Furthermore, the cost for the local inverted index construction would add up when the number of small partitions is large, which can dramatically affect the overall performance. 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.

7.1.4 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 10a–h shows the results. Note that on the TWITTER dataset with data cardinality 100%, PRETTI failed to return results due to an out-of-memory error. We

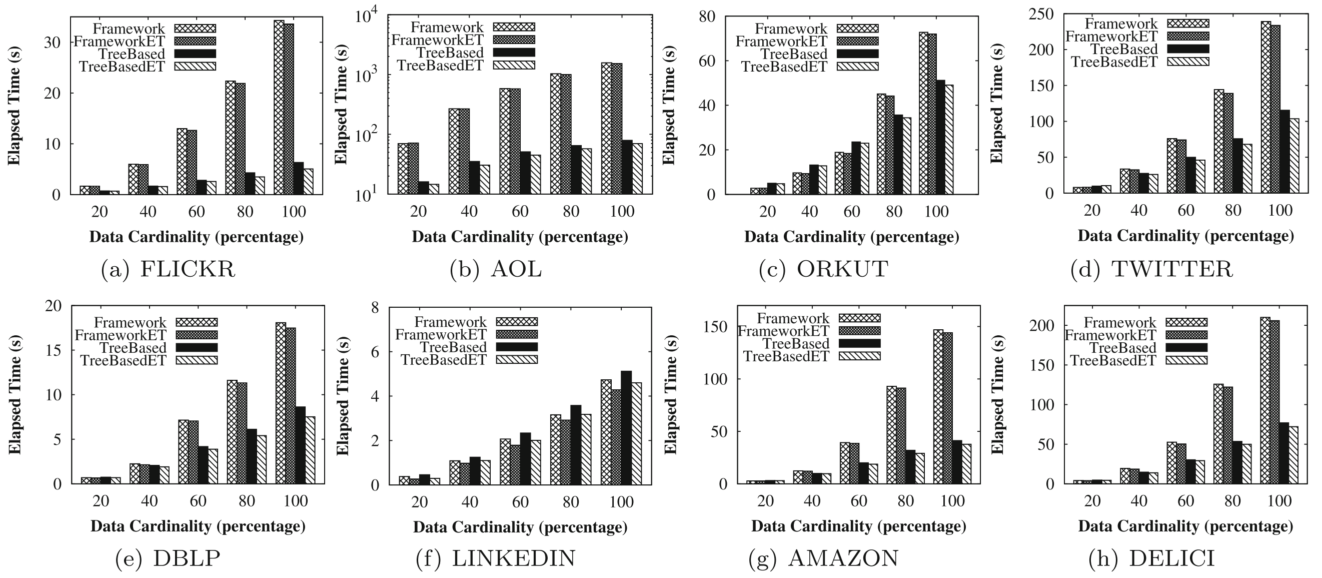


Fig. 8 Evaluation of the tree-based methods

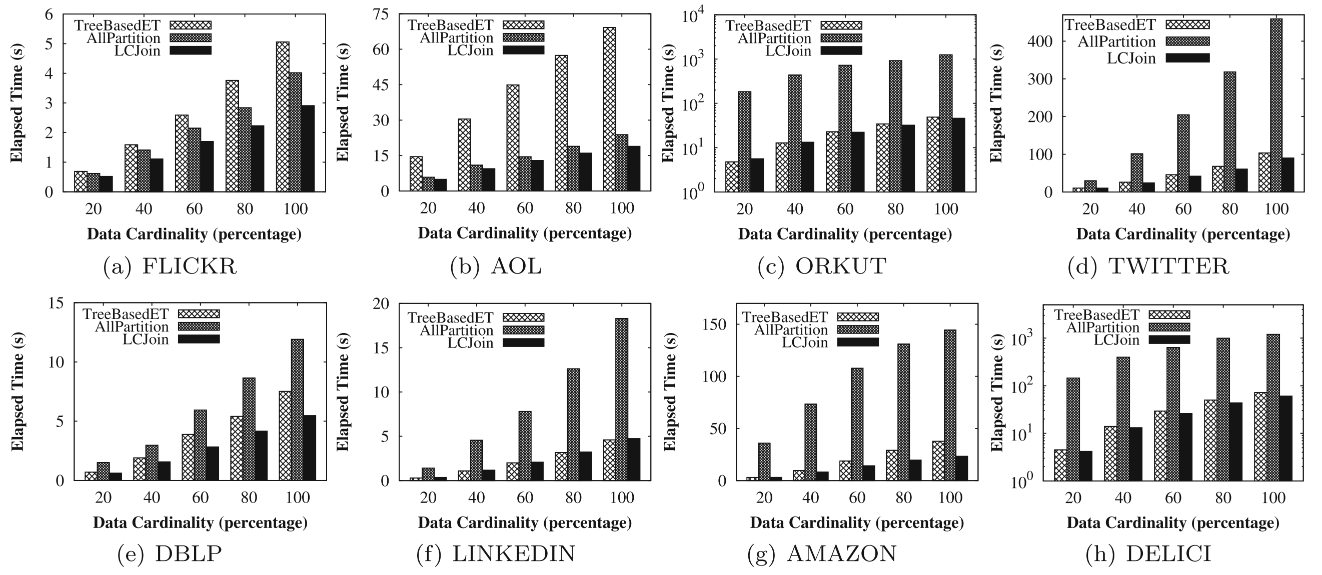


Fig. 9 Evaluation of the data partition methods

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 was 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 noticed that TT-Join outperformed PRETTI and LIMIT+ in most cases. However, on datasets with large data cardinalities and moderate z-values (such as TWITTER, AMAZON and DELICI), its superiority is not obvious. This is because there are a huge number of candidates to check and verify when applying the fixed-length signature scheme (k least frequent prefix) on datasets that are not very skew. Additionally, 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 was, respectively, 0.52 s, 1.11 s, 1.7 s, 2.23 s, and 2.91 s, 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.

In addition, we evaluated the effect of data duplicates on all the methods. Specifically, we ran all the methods on the deduplicated FLICKR, AOL, ORKUT, and TWIT-

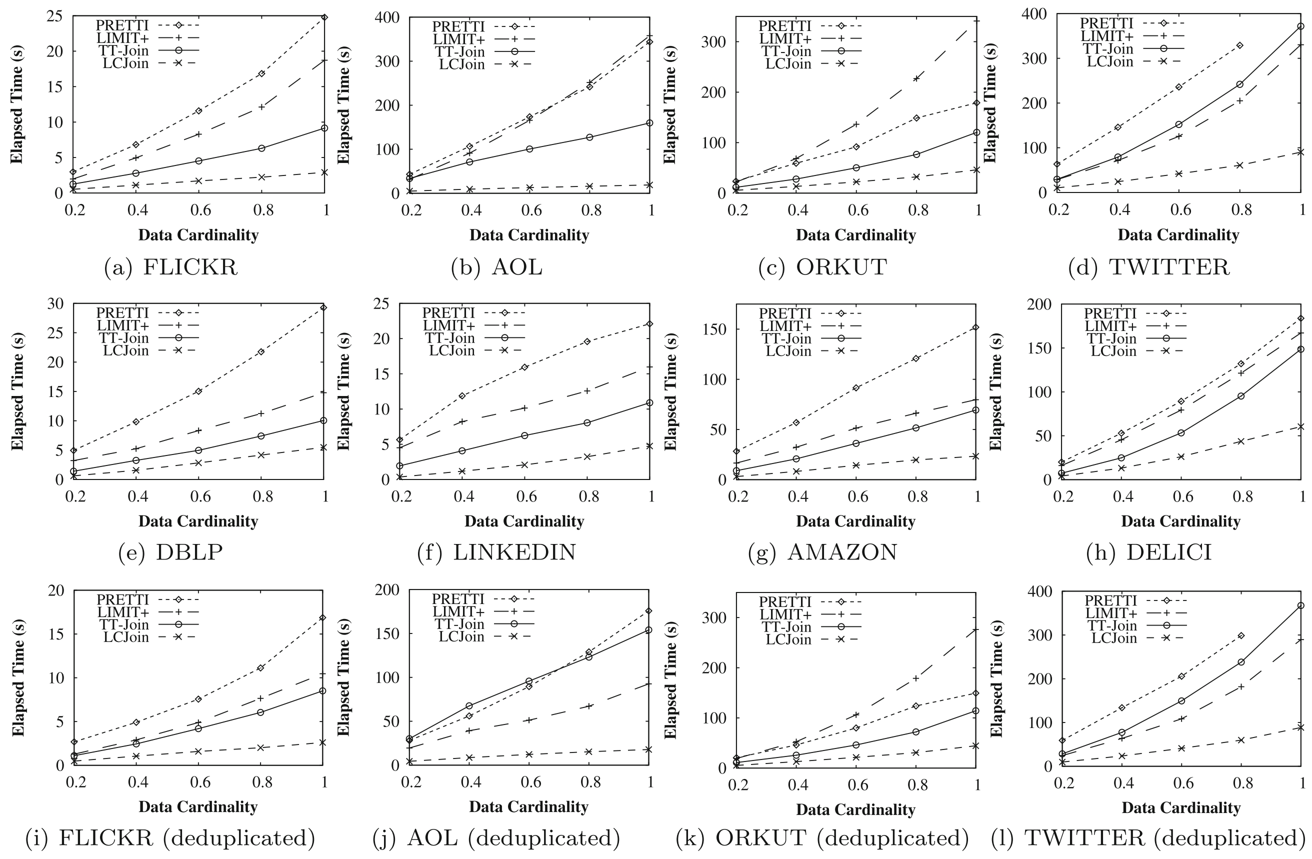


Fig. 10 Comparing with existing approaches on real-world datasets

TER datasets and reported the elapsed time. Note that the other four datasets (i.e., DBLP, LINKEDIN, AMAZON, and DELICI) have no duplicate sets and were excluded from this experiment. After deduplication, the data cardinality of the four datasets FLICKR, AOL, ORKUT, and TWITTER is 1.13 million, 10.05 million, 11.07 million, and 25.11 million, respectively. The results are shown in Fig. 10i–l. We can see that our method still performed significantly better than the other state-of-the-art methods. Moreover, our method was most robust to duplicates. Its runtime almost remained the same. This is because, to efficiently deal with duplicates, instead of explicitly indexing the identifiers of all duplicate sets in the inverted index $\mathcal{I}$, in our implementation we propose to index the leaf nodes (node ID in the prefix tree $\mathcal{T}$). As each leaf node contains all the duplicate sets sharing the same path from the root to this node, we only index the leaf node once in the inverted index. The inverted index $\mathcal{I}$ can be efficiently constructed by left-to-right postorder tree traversing. Thus, the computation is on the basis of leaf nodes, and no redundant computation is performed. We also observed that the runtime of LIMIT+ reduced a lot. This is because it indexes a short prefix (usually 2) on the tree index, which cannot distinguish the duplicate sets and needs to conduct the same computation for each of the duplicate sets. Similarly, TT-Join indexes k-length least frequent prefix in a tree and the entire sets on the other prefix tree. Thus, it is less robust to duplicates than our method. However, we noticed that the runtime of TT-Join also almost remained the same. This is because most of the duplicates are rather short and they can be fully indexed both by the two prefix trees. We also observed that the runtime of PRETTI reduced. This is because it scans shorter inverted lists after deduplication. Thus, our leaf node-based inverted index construction method can also help improve the performance of PRETTI on the datasets with duplicates.

We also measured the peak memory usage of the four algorithms. Figure 11 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 was 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

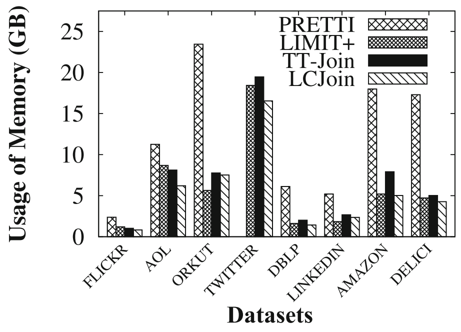


Fig. 11 Comparing peak memory usage with existing methods on real-world datasets

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.

7.1.5 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 12 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. 12c, 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. 12a depicts the results for scalability experiments. We can see that our approach showed good scalability with the increase in 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 in the data cardinality, this is because the fixed-length signature scheme (k least frequent prefix) is not adaptive to the datasets. Figure 12b shows the results for different average set sizes. PRETTI failed to return results when the average set size was greater than 32. It also shows good scalability of our algorithm with the increase in the average set size. Figure 12c gives the runtime of all algorithms when varying the number of distinct elements. We observe that the performance of LCJoin is rather steady. For example, when the number of distinct element was 10 thousands and 10 million, the runtime of our approach was, respectively, 52 s and 16 s, while for TT-join, PRETTI, and LIMIT+ the run time was 3604 s/1687 s/1393 s and 40 s/124 s/69 s. 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 12d 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×.

7.2 External memory evaluations

7.2.1 Experimental setup

The datasets and experimental environment are the same as that used in Sect. 7.1. In addition, a hard disk drive (HDD) with 15,000 RPM was used in the experiment. All experiments were conducted using the direct I/O mode in Linux to eliminate influences caused by the data caching of file systems. Specifically, the page size was set to 4 kB. The experiments were performed to test the run time and I/O counts (i.e., physical disk reads) under various experimental settings.

7.2.2 Baseline method

As there is no existing work to process external memory set containment joins under a limited memory budget, we adapt the intersection-oriented PRETTI algorithm as a baseline for comparison. We denote it as EXPRETTI.

EXPRETTI. A straightforward way to extend PRETTI to the external memory scenario is amortizing the I/O cost of loading inverted lists among the sets with common prefixes. To implement this, we first sort sets in $\mathbb{R}$ in an lexicographical increasing order so that the sets sharing common prefixes are clustered. Next, we build an inverted index $\mathcal{I}$ for the sets in $\mathbb{S}$ and persist it to disk. Finally, for each set $\mathsf{R} \in \mathbb{R}$, we load all the inverted lists corresponding to the elements in R from the disk and intersect them. Each time when processing a set R, we can reuse the intermediate intersection results if it shares common prefixes with the sets that have already been processed, which can avoid the I/O cost of loading inverted lists corresponding to the common prefixes. As all sets in $\mathbb{R}$ are sorted lexicographically, we only need to reserve the intermediate intersection results of the latest processed set, including the inverted list of its smallest element.

7.2.3 Evaluation on real-world datasets

In this section, we compared EXLCJoin (described in Sect. 6.3) with EXPRETTI on real-world datasets by varying the memory budget (2^{-3}%, 2^{-2}%, 2^{-1}%, 2^{0}%, 2^{1}%, 2^{2}%, 2^{3}%, and 2^{4}% of the dataset size). The results are shown in Figs. 13 and 14. We observed that EXLCJoin consistently outperformed EXPRETTI by a big margin with respect to both I/O cost and

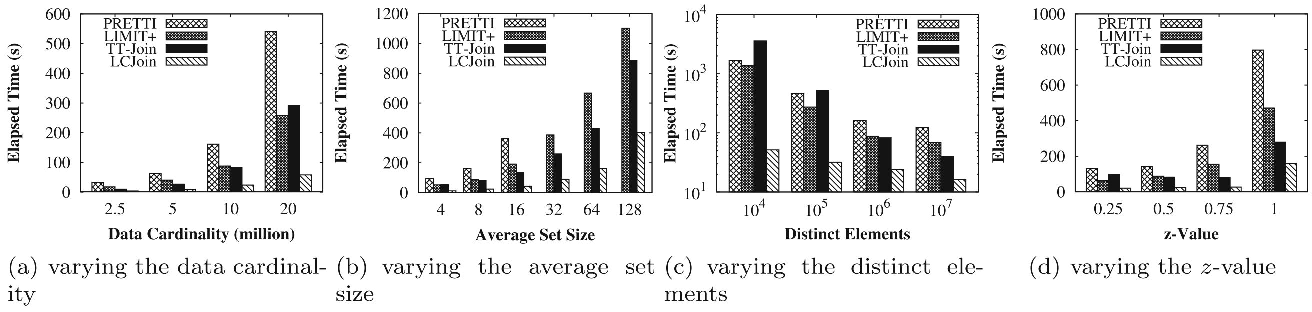


Fig. 12 Comparing with existing approaches on synthetic datasets

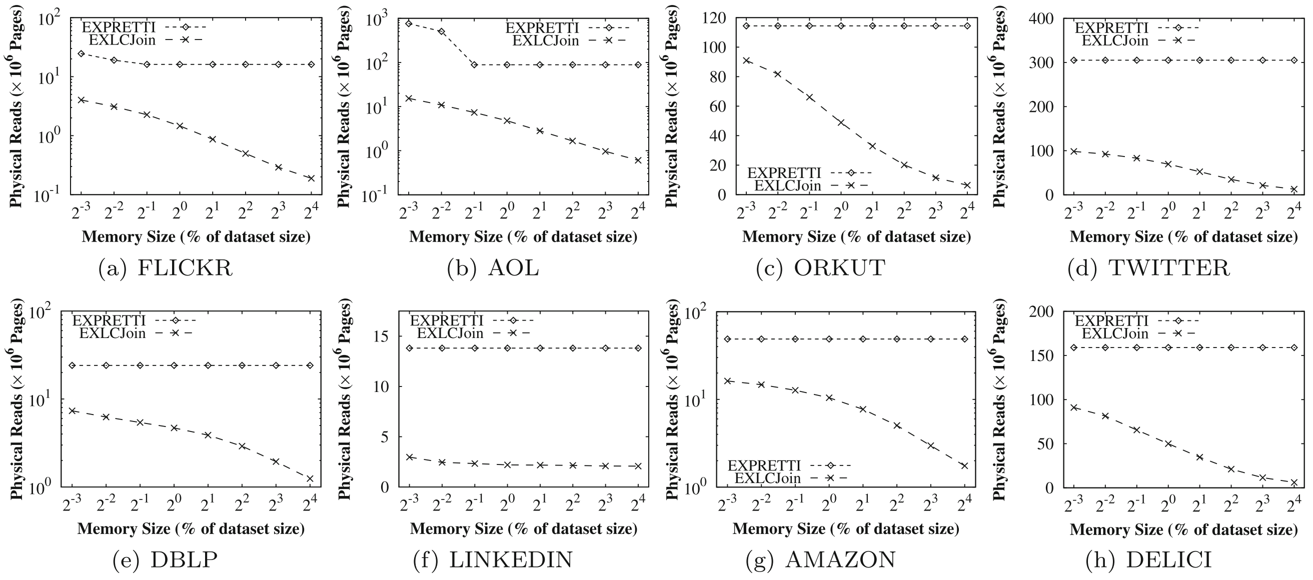


Fig. 13 I/O counts of EXLCJoin and the baseline on real-world datasets

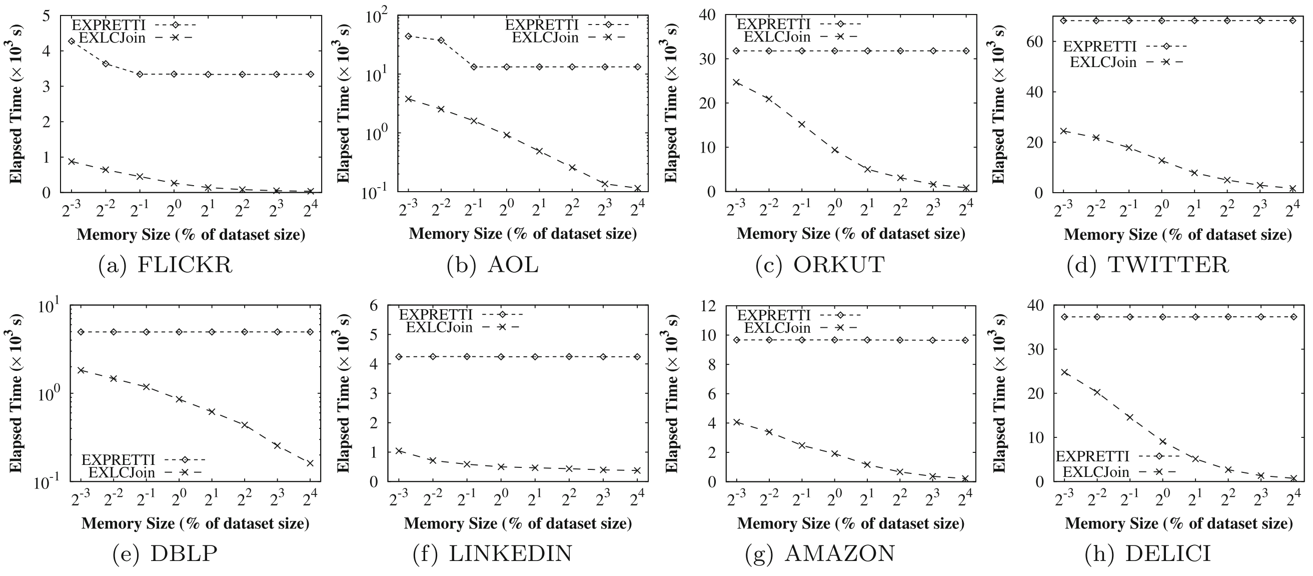


Fig. 14 Run time of EXLCJoin and the baseline on real-world datasets

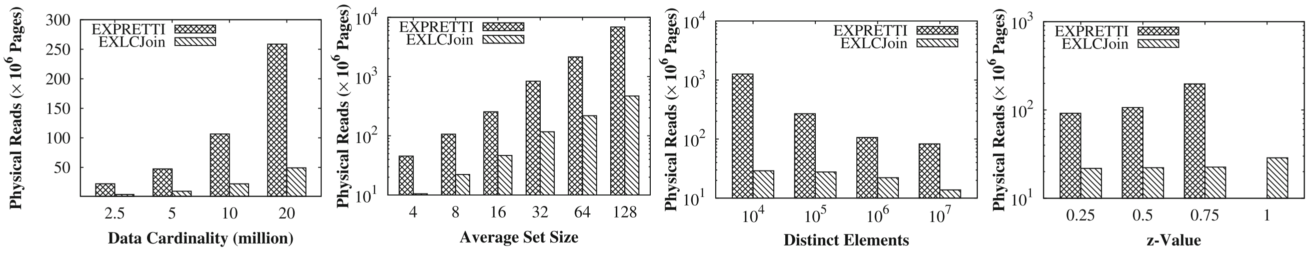
run time in all cases. For example, on AOL dataset, when the memory size was 2^4%, the I/O counts for EXPRETTI and EXLCJoin were 88.7 million and 0.6 million, respectively, while the run time was 13228s and 124s, respectively. The speedup of EXLCJoin over EXPRETTI is 100×. The main reason is that EXLCJoin excludes the "irrelevant" sets in $\mathbb{S}$ that must not be a superset of any set in the in-memory partition in advance, and as a result, the number of *target-sets* that are required to be loaded from the disk for local inverted index construction is effectively reduced. Moreover, with the help of incremental data loading, which can fully leverage the available memory by union as many small partitions as possible, the cost of constructing local inverted index is well amortized among these small partitions when batch processing them together. In contrast, EXPRETTI cannot make full use of the available memory space, and the I/O cost of loading inverted lists can only be shared among sets with common prefixes. For the rest suffixes, it has to repeatedly load the corresponding inverted lists to perform list intersections and therefore lead to a large number of disk reads. We also observed that EXLCJoin scaled much better than EXPRETTI. For example, on FLICKR dataset, when memory size increased form 2^{-3} to 2^4%, the run time for EXPRETTI decreased by 1.3× (from 4273 to 3349 s), while the run time for EXLCJoin decreased by 23.6× (from 874 to 37 s). This is due to the fact that the I/O cost of loading a candidate set S can be amortized by more sets in $\mathbb{R}$ with the increase in memory budget. Note that the performance of EXPRETTI remains stable when the memory size exceeds a threshold, where for arbitrary set R, all its intermediate intersection results can be cached in memory. Additionally, we also observed that the run time of EXLCJoin decreased faster than the I/O counts when the memory size increased. For example, on DBLP dataset, when memory size increased form 2^{-3} to 2^4%, the I/O count for EXLCJoin decreased by 5.9× (from 7.34 million to 1.24 million), while the run time decreased by 11.3× (from 1822.8 to 161.5 s). The reason is that when a larger memory budget is imposed, we need to load more sets in $\mathbb{S}$ to build the local inverted index for a partition, which would result in smaller average distances of fetched sets in $\mathbb{S}$ on the disk. On the other hand, the average disk scan cost of each set S is proportional to the distances between them.

7.2.4 Evaluation on synthetic datasets

In this section, we evaluate the performance of the two methods on synthetic datasets. In all cases, the memory budget was set to 2% of the dataset size. Figures 14a and 15a show the results on different data cardinalities. We observed that the performance gap between EXLCJoin and EXPRETTI increased with the dataset size. This is because EXLCJoin takes advantage of incremental data loading to maximize the usage of available memory budget that increases with the data cardinality, and in cooperation with the method of batch processing sets in memory, it results in low amortized I/O cost. However, for EXPRETTI, it cannot make good usage of the increasing memory budget, and has to scan longer inverted lists to process each set R when the data cardinality is larger. Figures 14b and 15b depict the total I/O cost and run time when varying the average set size. Note the logarithmic vertical scale. We can see that EXLCJoin scaled well with the increasing set size, while EXPRETTI favored small set size and its performance decreased sharply with the increase in set size. This is due to the fact that the I/O cost of most list intersections cannot be shared under a large set size. For example, when the average set size was 4, EXPRETTI had 4.3× more I/O counts than EXLCJoin. However, when the set size increased to 128, the gap between I/O counts became 14.6×. Figures 14c and 15c show the results on various element domain size. As expected, EXPRETTI preferred a larger element domain size because of the shorter length of inverted lists, and it needed less I/O cost to load them. We also observed that EXLCJoin had steady performance regardless of the number of distinct elements, and was still significantly superior to the EXPRETTI algorithm. This is because our common prefix-based data partition can effectively reduce the *target-sets* in $\mathbb{S}$ that are required to be loaded from disk to build the local inverted index. Figures 14d and 15d show the effect of different data skewness. Note that EXPRETTI failed to return the result in reasonable time when the z-value was 1. The reason is that for many sets in $\mathbb{R}$, the inverted list of the smallest element of R cannot be cached in memory due to the high skewness of elements; thus, EXPRETTI had to load these especially long inverted lists repeatedly. The performance of EXLCJoin was less affected with the increase in z-value, and it was up to one order of magnitude faster than EXPRETTI.

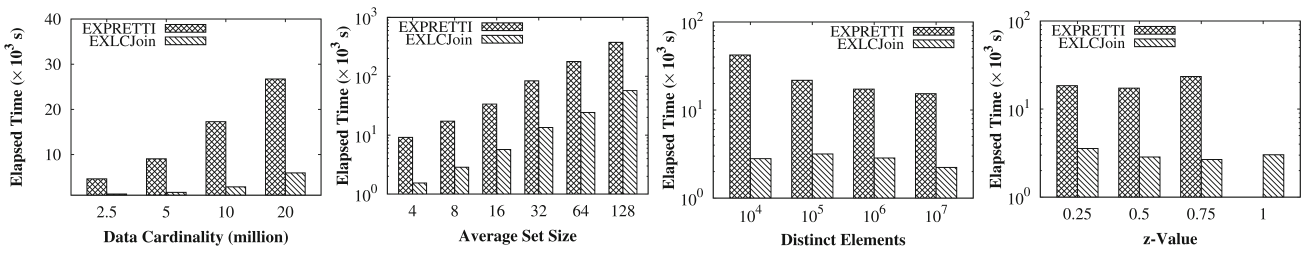
8 Related work

Containment joins. Data produced in the form of set-values are omnipresent [44–46]. Containment joins on set-valued data have become an important building block for today's knowledge and data management due to its wide range of applications [5]. 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 [16] proposed to use the signature-hash join (SHJ) to address the set containment join problem. SHJ first uses a signature structure [36] 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. [35] proposed the partitioned set join (PSJ) method, while Melnik et al. [30] developed the divide-and-conquer set join



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

Fig. 15 I/O counts of EXLCJoin and the baseline on synthetic datasets



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

Fig. 16 Run time of EXLCJoin and the baseline on synthetic datasets

(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. [31] 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. [27] further improved these methods by using a Patricia tree to reduce the signature enumeration cost. Yang et al. [58,59] 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 [5,58,59], most union-oriented methods are not competitive with intersection-oriented methods. Mamoulis [28] 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 [19] improved BNL by using a prefix tree to share the computation across the sets in $\mathbb{R}$. Luo et al. [27] 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. [5] 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. [21] 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. Additionally, with the broad application of distributed computing systems [51], Yang et al. [59] extended set containment join system to a distributed environment to improve the scalability. Specifically, they proposed a signature-based distribution mechanism, which can achieve good load balance as well as low communication cost.

Containment queries. In addition to the set containment join problem, the set containment query problem has also been extensively studied. Given a query set Q and a collection $\mathbb{S}$ of sets, the set containment query asks for all the sets in $\mathbb{S}$ that contain the query set Q. Helmer and Moerkotte [17] conducted an experimental study on the efficiency of various index structures for set containment queries. The experimental results indicate that the inverted index exhibits the best overall performance. Terrovitis et al. [47] proposed to combine the prefix tree and inverted index to support efficient set containment queries. The prefix tree indexes the top-k most frequent elements of the dataset and is placed in main memory, whereas the inverted index is placed in second storage. However, the size of some inverted lists might be especially large when the distribution of elements is skewed. To address this issue, Terrovitis et al. [43] proposed the ordered inverted file (OIF), which employs B-trees to order the sets in an inverted file by their set values. The OIF is capable of reducing the number of disk pages that need to be read from the second storage. Ibrahim and Fletcher [18] extended the prefix tree-based method for containment queries on nested sets, which can have both atomic and set-valued objects as elements. Yang et al. [60] proposed a prefix tree-based sampling approach for selectivity estimation on set containment queries.

Similarity joins. Another relevant line of research is the similarity join. A similarity join takes two collections of objects (such as sets and sequences) and identifies all the object pairs that are similar enough with regard to a given similarity function and threshold. There are a few experimental studies and survey for this topic [14,20,29,49,61]. More specifically, Deng et al. [9] proposed a partition-based method for set similarity joins under Jaccard similarity constraints. A similar approach is designed by Li et al. [24] for edit distance similarity join. Bayardo et al. [4], 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. [57] improved the prefix filter with a position filter and extended the technique to address top-k queries [56] and for edit-distance constraints [55]. Wang et al. [50] proposed AdaptJoin algorithm, which uses a longer prefix to filtering more dissimilar pairs. Wang et al. [52] combined the ideas of token universe partitioning and prefix filtering for the local similarity search problem, which identifies partially replicated text among a large number of documents. Further, Agrawal et al. [1] proposed parallel approaches to speed up the local similarity search. Wang et al. [53,54] proposed to group related sets into blocks in the index and explore possible computational cost sharing within the same block when processing set similarity joins. Qin et al. [34] proposed a new pigeonring principle, which organizes the boxes in a ring and constrains the number of items in multiple adjacent boxes, thus yielding stronger conditions compared with the basic pigeonring principle. Further, in cooperation with the token universe partitioning method, the new principle was utilized to effectively find the candidate sets whose distances to the query set might fall below a given threshold. Deng et al. [10] proposed a size-aware method for the overlap set similarity join problem, which finds all the set pairs with a sufficient large overlap size. Deng et al. [7] proposed an efficient method to find related sets using two-tier similarity functions. Li et al. [23] proposed a partition-based method for string similarity joins based on edit distance constraints, which find every string pair from two large sets of strings whose edit distance is within a given threshold. To further improve the scalability, a number of solutions [3,8,11–13,15,25,26,32,37–42,48] have been proposed to perform the set similarity join on MapReduce [6] or Spark [62]. Agrawal et al. [2] proposed to solve the error-tolerant set containment join problem. Li et al. [22] developed algorithms to solve the T-occurrence problem. These methods can be adapted to solve our set containment join problem. However, as shown in [59], they did not perform well when applied to our problem.

9 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 and apply different approaches to each partition. Furthermore, we proposed an adaptive and I/O efficient data partition method to support large-scale datasets that cannot be kept in memory. The experimental results demonstrated the superiority of our proposal.

References

1. Agrawal, M., Manchanda, K., Soni, R., Lal, A., Chowdary, C.R.: Parallel implementation of local similarity search for unstructured text using prefix filtering. In: International Conference on Parallel and Distributed Computing, Applications and Technologies, pp. 98–103 (2017)
2. Agrawal, P., Arasu, A., Kaushik, R.: On indexing error-tolerant set containment. In: SIGMOD, pp. 927–938 (2010)
3. Baraglia, R., Morales, G.D.F., Lucchese, C.: Document similarity self-join with mapreduce. In: ICDM, pp. 731–736 (2010)
4. Bayardo, R.J., Ma, Y., Srikant, R.P: Scaling up all pairs similarity search. In: WWW, pp. 131–140 (2007)
5. Bouros, P., Mamoulis, N., Ge, S., Terrovitis, M.: Set containment join revisited. Knowl. Inf. Syst. **49**(1), 375–402 (2016)
6. Dean, J., Ghemawat, S.: Mapreduce: a flexible data processing tool. Commun. ACM **53**(1), 72–77 (2010)
7. Deng, D., Kim, A., Madden, S., Stonebraker, M.: Silkmoth: an efficient method for finding related sets with maximum matching constraints. PVLDB **10**(10), 1082–1093 (2017)

8. Deng, D., Li, G., Hao, S., Wang, J., Feng, J.: Massjoin: a mapreduce-based method for scalable string similarity joins. In: ICDE, pp. 340–351 (2014)
9. Deng, D., Li, G., Wen, H., Feng, J.: An efficient partition based method for exact set similarity joins. PVLDB **9**(4), 360–371 (2015)
10. Deng, D., Tao, Y., Li, G.: Overlap set similarity joins with theoretical guarantees. In: SIGMOD, pp. 905–920 (2018)
11. Ding, X., Yang, W., Choo, K.R., Wang, X., Jin, H.: Privacy preserving similarity joins using mapreduce. Inf. Sci. **493**, 20–33 (2019)
12. do Carmo Oliveira, D.J., Borges, F.F., Ribeiro, L.A., Cuzzocrea, A.: Set similarity joins with complex expressions on distributed platforms. In: ADBIS, pp. 216–230 (2018)
13. Elsayed, T., Lin, J.J., Oard, D.W.: Pairwise document similarity in large collections with mapreduce. In: ACL, pp. 265–268 (2008)
14. Fier, F., Augsten, N., Bouros, P., Leser, U., Freytag, J.: Set similarity joins on mapreduce: an experimental survey. PVLDB **11**(10), 1110–1122 (2018)
15. Gavagsaz, E., Rezaee, A., Javadi, H.H.S.: Load balancing in join algorithms for skewed data in mapreduce systems. J. Supercomput. **75**(1), 228–254 (2019)
16. Helmer, S., Moerkotte, G.: Evaluation of main memory join algorithms for joins with set comparison join predicates. In: VLDB, pp. 386–395 (1997)
17. Helmer, S., Moerkotte, G.: A performance study of four index structures for set-valued attributes of low cardinality. VLDB J. **12**(3), 244–261 (2003)
18. Ibrahim, A., Fletcher, G.H.L.: Efficient processing of containment queries on nested sets. In: EDBT, pp. 227–238 (2013)
19. Jampani, R., Pudi, V.: Using prefix-trees for efficiently computing set joins. In: DASFAA, pp. 761–772 (2005)
20. Jiang, Y., Li, G., Feng, J., Li, W.: String similarity joins: an experimental evaluation. PVLDB **7**(8), 625–636 (2014)
21. Kunkel, A., Rheinländer, A., Schiefer, C., Helmer, S., Bouros, P., Leser, U.: Piejoin: towards parallel set containment joins. In: SSDBM, pp. 11:1–11:12 (2016)
22. Li, C., Lu, J., Lu, Y.: Efficient merging and filtering algorithms for approximate string searches. In: ICDE, pp. 257–266 (2008)
23. Li, G., Deng, D., Feng, J.P.: A partition-based method for string similarity joins with edit-distance constraints. ACM Trans. Database Syst. **38**(2), 9:1–9:33 (2013)
24. Li, G., Deng, D., Wang, J., Feng, J.: PASS-JOIN: a partition-based method for similarity joins. PVLDB **5**(3), 253–264 (2011)
25. Li, R., Ju, L., Peng, Z., Yu, Z., Wang, C.: Batch text similarity search with mapreduce. In: 13th Asia-Pacific Web Conference, pp. 412–423 (2011)
26. Liu, W., Shen, Y., Wang, P.: An efficient mapreduce algorithm for similarity join in metric spaces. J. Supercomput. **72**(3), 1179–1200 (2016)
27. Luo, Y., Fletcher, G.H.L., Hidders, J., Bra, P.D.: Efficient and scalable trie-based algorithms for computing set containment relations. In: ICDE, pp. 303–314 (2015)
28. Mamoulis, N.: Efficient processing of joins on set-valued attributes. In SIGMOD, pp. 157–168 (2003)
29. Mann, W., Augsten, N., Bouros, P.: An empirical evaluation of set similarity join techniques. PVLDB **9**(9), 636–647 (2016)
30. Melnik, S., Garcia-Molina, H.: Divide-and-conquer algorithm for computing set containment joins. In: EDBT, pp. 427–444 (2002)
31. Melnik, S., Garcia-Molina, H.: Adaptive algorithms for set containment joins. ACM Trans. Database Syst. **28**, 56–99 (2003)
32. Metwally, A., Faloutsos, C.: V-smart-join: a scalable mapreduce framework for all-pair similarity joins of multisets and vectors. PVLDB **5**(8), 704–715 (2012)
33. Newman, M.E.J.: Power laws, Pareto distributions and Zipf's law. Contemp. Phys. **46**, 323–351 (2005)
34. Qin, J., Xiao, C.: Pigeonring: a principle for faster thresholded similarity search. PVLDB **12**(1), 28–42 (2018)
35. Ramasamy, K., Patel, J.M., Naughton, J.F., Kaushik, R.P: Set containment joins: the good, the bad and the ugly. In: VLDB, pp. 351–362 (2000)
36. Roberts, C.: Partial-match retrieval via the method of superimposed codes. Proc. IEEE **67**(12), 1624–1642 (1979)
37. Rong, C., Lin, C., Silva, Y.N., Wang, J., Lu, W., Du, X.: Fast and scalable distributed set similarity joins for big data analytics. In: ICDE, pp. 1059–1070 (2017)
38. Rong, C., Lu, W., Wang, X., Du, X., Chen, Y., Tung, A.K.H.: Efficient and scalable processing of string similarity join. IEEE Trans. Knowl. Data Eng. **25**(10), 2217–2230 (2013)
39. Sarma, A.D., He, Y., Chaudhuri, S.: Clusterjoin: a similarity joins framework using map-reduce. PVLDB **7**(12), 1059–1070 (2014)
40. Silva, Y.N., Reed, J.M.: Exploiting mapreduce-based similarity joins. In: SIGMOD, pp. 693–696 (2012)
41. Sun, J., Shang, Z., Li, G., Bao, Z., Deng, D.: Balance-aware distributed string similarity-based query processing system. PVLDB **12**(9), 961–974 (2019)
42. Sun, J., Shang, Z., Li, G., Deng, D., Bao, Z.: Dima: a distributed in-memory similarity-based query processing system. PVLDB **10**(12), 1925–1928 (2017)
43. Terrovitis, M., Bouros, P., Vassiliadis, P., Sellis, T.K., Mamoulis, N.: Efficient answering of set containment queries for skewed item distributions. In: EDBT, pp. 225–236 (2011)
44. Terrovitis, M., Liagouris, J., Mamoulis, N., Skiadopoulos, S.: Privacy preservation by disassociation. PVLDB **5**(10), 944–955 (2012)
45. Terrovitis, M., Mamoulis, N., Kalnis, P.: Privacy-preserving anonymization of set-valued data. PVLDB **1**(1), 115–125 (2008)
46. Terrovitis, M., Mamoulis, N., Kalnis, P.: Local and global recoding methods for anonymizing set-valued data. VLDB J. **20**(1), 83–106 (2011)
47. Terrovitis, M., Passas, S., Vassiliadis, P., Sellis, T.K.: A combination of trie-trees and inverted files for the indexing of set-valued attributes. In: CIKM, pp. 728–737 (2006)
48. Vernica, R., Carey, M.J., Li, C.: Efficient parallel set-similarity joins using mapreduce. In: SIGMOD, pp. 495–506 (2010)
49. Wandelt, S., Deng, D., Gerdjikov, S., Mishra, S., Mitankin, P., Patil, M., Siragusa, E., Tiskin, A., Wang, W., Wang, J., Leser, U.: State-of-the-art in string similarity search and join. SIGMOD Record **43**(1), 64–76 (2014)
50. Wang, J., Li, G., Feng, J.: Can we beat the prefix filtering?: An adaptive framework for similarity join and search. In: SIGMOD, pp. 85–96 (2012)
51. Wang, L., von Laszewski, G., Younge, A.J., He, X., Kunze, M., Tao, J., Fu, C.: Cloud computing: a perspective study. New Gener. Comput. **28**(2), 137–146 (2010)
52. Wang, P., Xiao, C., Qin, J., Wang, W., Zhang, X., Ishikawa, Y.: Local similarity search for unstructured text. In: SIGMOD, pp. 1991–2005 (2016)
53. Wang, X., Qin, L., Lin, X., Zhang, Y., Chang, L.: Leveraging set relations in exact set similarity join. PVLDB **10**(9), 925–936 (2017)
54. Wang, X., Qin, L., Lin, X., Zhang, Y., Chang, L.: Leveraging set relations in exact and dynamic set similarity join. VLDB J. **28**(2), 267–292 (2019)
55. Xiao, C., Wang, W., Lin, X.: Ed-join: an efficient algorithm for similarity joins with edit distance constraints. PVLDB **1**(1), 933–944 (2008)
56. Xiao, C., Wang, W., Lin, X., Shang, H.: Top-k set similarity joins. In: ICDE, pp. 916–927 (2009)
57. Xiao, C., Wang, W., Lin, X., Yu, J.X.: Efficient similarity joins for near duplicate detection. In: WWW, pp. 131–140 (2008)
58. Yang, J., Zhang, W., Yang, S., Zhang, Y., Lin, X.: Tt-join: efficient set containment join. In: ICDE, pp. 509–520 (2017)
59. Yang, J., Zhang, W., Yang, S., Zhang, Y., Lin, X., Yuan, L.: Efficient set containment join. VLDB J. **27**(4), 471–495 (2018)

60. Yang, Y., Zhang, W., Zhang, Y., Lin, X., Wang, L.: Selectivity estimation on set containment search. In: DASFAA, pp. 330–349 (2019)
61. Yu, M., Li, G., Deng, D., Feng, J.: String similarity search and join: a survey. Front. Comput. Sci. **10**(3), 399–417 (2016)
62. Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J., Ghodsi, A., Gonzalez, J., Shenker, S., Stoica, I.: Apache spark: a unified engine for big data processing. Commun. ACM **59**(11), 56–65 (2016)