# PIEJoin: Towards Parallel Set Containment Joins

Anja Kunkel*[1]   Astrid Rheinländer*[1]   Christopher Schiefer*[1]
Sven Helmer‡[2]   Panagiotis Bouros+[3]   Ulf Leser*[1]
*Humboldt-Universität zu Berlin, Germany    +Aarhus University, Denmark
‡Free University of Bozen-Bolzano, Italy
[1]{akunkel,rheinlae,schiefec,leser}@informatik.hu-berlin.de [2]shelmer@inf.unibz.it [3]pbour@cs.au.dk

## ABSTRACT

The efficient computation of set containment joins (SCJ) over set-valued attributes is a well-studied problem with many applications in commercial and scientific fields. Nevertheless, there still exists a number of open questions: An extensive comparative evaluation is still missing, the two most recent algorithms have not yet been compared to each other, and the exact impact of item sort order and properties of the data on algorithms performance still is largely unknown. Furthermore, all previous works only considered sequential join algorithms, although modern servers offer ample opportunities for parallelization.

We present PIEJoin, a novel algorithm for computing SCJ based on intersecting prefix trees built at runtime over the to-be-joined attributes. We also present a highly optimized implementation of PIEJoin which uses tree signatures for saving space and interval labeling for improving runtime of the basic method. Most importantly, PIEJoin can be parallelized easily by partitioning the tree intersection. A comprehensive evaluation on eight data sets shows that PIEJoin already in its sequential form clearly outperforms two of the three most important competitors (PRETTI and PRETTI+). It is mostly yet not always slower than the third, LIMIT+(opj) but requires significantly less space. The parallel version of PIEJoin we present here achieves significant further speed-ups, yet our evaluation also shows that further research is needed as finding the best way of partitioning the join turns out to be non-trivial.

## 1. INTRODUCTION

In its conventional form, the relational model expects attributes of a tuple to hold only a single value (or none) [3]. If the property of an entry allows multiple values, such as the qualifications of a person, the tags of a photo, or the outlinks of a webpage, the relational way of modelling is the use of a proper relation for the values connected by primary and foreign keys. However, many applications require operations on these sets, such as building the intersection of sets (all tags shared by two photos) or testing their containment (all persons more qualified as a given one). Formulating such tasks with SQL requires long queries leading to inefficient evaluation algorithms, which resulted in two lines of research: (1) Modelling languages which allow set-valued attributes, such as the object-relational model or NF2 models [18], and (2) specialized algorithms which take sets of sets as input and efficiently compute the desired operations (containment, similarity, equality etc.) [5].

A particular important operation on tuples with set-valued attributes is the containment join (or *Set Containment Join* (SCJ)): Given two relations $R$, $S$ with set-valued attributes $R.a$, $S.b$ respectively, SCJ returns the subset of the $R \times S$ cross-product with $r.a \subseteq s.b$. Set containment joins were extensively studied in the last two decades [5, 8, 11, 14, 15, 16]; in this context, a plethora of methods was proposed based on diverse base data structures, such as signature files, hashing, inverted files, or trees. For quite some time, the presumably fastest method was PRETTI [8], which indexes relation $R$ with a prefix tree and relation $S$ with an inverted index. The last years have brought a renewed interest in optimizing the computation of SCJ due to the ever increasing data set sizes. The two most notably novel contributions both build on top of PRETTI: PRETTI+ by Luo et al. [10] employs Patricia trees instead of prefix trees, while LIMIT and its variants by Bouros et al. [1] builds the prefix tree on $R$ only up to a predefined depth and interleaves index creation with the actual join computation. This predefined depth is an important data-dependent tuning parameter which influences the efficiency of the join and automatically setting it requires heuristics [1]. Note that none of these methods adequately exploits modern computer architectures with their ever increasing number of parallel threads.

In this paper, we present a new algorithm for computing set-containment joins called *PIEJoin* (Preorder Interval Encoded Set Containment Join). Conceptually, PIEJoin indexes both input relations using prefix trees and computes the SCJ by intersecting these trees. PIEJoins implementation is highly optimized to achieve space efficiency and high performance, replacing the traditional objects-and-pointers representation of prefix trees with a set of compact arrays and using preorder interval labeling to compute the otherwise costly look-ups of elements from $R$ in $S$ in constant time. As a result, PIEJoin requires significantly less space compared to the currently available implementations of PRETTI, PRETTI+, and the LIMIT methods. When run in sequential mode, PIEJoin also clearly outperforms both PRETTI and PRETTI+, achieving execution times

Prefix tree for R:

name:"" pre:0

name:A pre:1 • name:E key:{r5} pre:8

name:B key:{r1} pre:2

name:C pre:3 • name:D key:{r3} pre:6 • name:F key:{r4} pre:7

name:D pre:4

name:E key:{r2} pre:5

Relation R:

| Tuple ID | Attribute a |
|----------|-------------|
| r1 | {A,B} |
| r2 | {A,B,C,D,E} |
| r3 | {A,B,D} |
| r4 | {A,B,F} |
| r5 | {E} |

Prefix tree for S:

name:"" i:[0,13]

name:A key:{s1} i:[1,8] • name:B i:[9,11] • name:E i:[12,13]

name:B key:{s2} i:[2,6] • name:D i:[7,8] • name:C i:[10,11] • name:F key:{s7} i:[13,13]

name:C i:[3,5] • name:E key:{s4} i:[6,6] • name:E key:{s5} i:[8,8] • name:D key:{s6} i:[11,11]

name:D i:[4,5]

name:E key:{s3} i:[5,5]

Relation S:

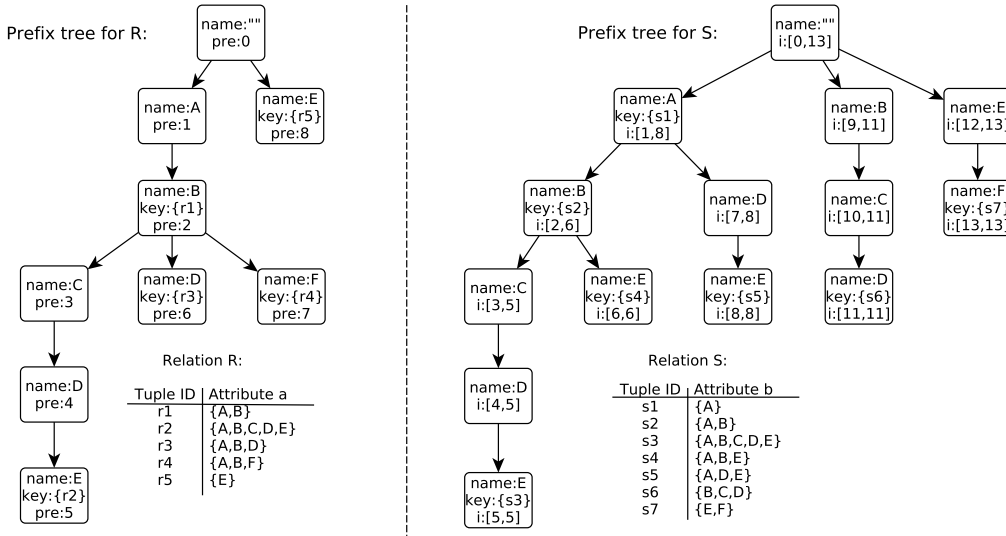| Tuple ID | Attribute b |
|----------|-------------|
| s1 | {A} |
| s2 | {A,B} |
| s3 | {A,B,C,D,E} |
| s4 | {A,B,E} |
| s5 | {A,D,E} |
| s6 | {B,C,D} |
| s7 | {E,F} |

Figure 1: Exemplary prefix trees for two set-valued relations in frequent sort order with preorder annotations (left) and preorder interval annotations (right).

close to the fastest variant from [1] termed LIMIT+(opj) without the need of any parameter tuning. Most importantly, the base paradigm of intersection two trees opens the door to a natural way of parallelizing the SCJ by assigning pairs of subtrees to be intersected to separate threads. We present initial methods for balancing the load generated by these pairs of (unevenly large) subtrees, leading to significant speed-ups on all data sets we considered. However, the scale-out is satisfying for only some of them, which calls for further research into parallelizing SCJs. More specifically, the contributions of our work the following:

- We propose PIEJoin, a novel SCJ algorithm based on prefix tree intersection, together with a highly efficient implementation using array-based tree signatures.

- We conduct the to date most comprehensive experimental analysis of SCJ algorithms involving eight real-world data sets each exhibiting different characteristics. We compare PIEJoin with its three most important competitors [1, 8, 10]. Note that this is the first analysis comparing both [1, 10] methods, which originally were presented almost at the same time. Furthermore, our evaluation also covers SCJ with two different relations, while almost all previous work only considered self-joins.

- We present initial results on the parallel computation of SCJ based on the PIEJoin algorithm. We perform an empirical investigation showing the potential of parallelization, yet also highlighting some pitfalls of such methods. To the best of our knowledge, this is the first paper addressing parallel SCJ algorithms.

**Outline.** The rest of the paper is organized as follows. Section 2 formally defines the problem of SCJ while Section 3 briefly outlines previous work. Section 4 details our novel PIEJoin algorithm while Section 5 presents our experimental evaluation on a single-threaded setup. Next, Section 6 introduces our parallel evaluation framework and presents our experimental analysis on a multi-threaded setup. Finally, Section 7 concludes the paper and discusses ideas for future work.

## 2. PRELIMINARIES

In this section, we introduce basic concepts and definitions relevant for the remainder of this paper. We study the efficient computation of set containment joins, which are defined as follows:

DEFINITION 1 (SET CONTAINMENT JOIN). *Given two relations $R, S$ with set-valued attributes $R.a$ and $S.b$, a set-containment join $R \bowtie_\subseteq S$ returns all pairs $(r, s), r \in R, s \in S$ where $r.a \subseteq s.b$.*

Note that Definition 1 primarily targets $RxS$-joins. In the self-join case on a relation R, the join condition for computing $R \bowtie_\subseteq R$ is adjusted to return all pairs $(r_1, r_2), r_1, r_2 \in R$ for which $r_1.a \subseteq r_2.a$ holds. We call the set of all distinct values stored in $R.a$ and $S.b$ domain.

Computing set SCJ can be accelerated by using proper index structures. PIEJoin and others employ prefix trees to index tuples from $R$ and/or $S$, which are defined as follows:

DEFINITION 2 (PREFIX TREE OVER SET $S$). *Let $V(T)$ denote a set of nodes, $E(T)$ denote a set of edges, and $v_0 \in V(T)$ denote the root node of tree $T(V, E, v_0)$. Let $S$ (short for $S.b$) be a set-valued attribute. We call $T(V, E, v_0)$ the prefix tree for $S$ iff the following holds:*

1. *Each node $v \in V(T)$ is assigned an ID $v.id$ and labeled with a name $v.name$ from the elements of $S$. The root node $v_0$ is labeled with the empty string. The set of a node $v$ is the set of names from $v$ and all its predecessors.*

2. *The labels of any two children $v_j, v_k$ of the same node $v$ are distinct.*

3. *For every set $s \in S$, there exists a node $v$ such that the set of $v$ equals $s$. We call any such $v$ a set node.*

**preorderToName:**

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| - | A | B | C | D | E | D | E | F |

**preorderToTupleID:**

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| - | - | r1 | - | - | r2 | r3 | r4 | r5 |

**preorderToChildIDs:**

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | - | - | - | - |
|   |   |   |   | 8 |   | 6 |   |   |
|   |   |   |   |   |   | 7 |   |   |

name:A
pre:1

name:B
key:{r1}
pre:2

Array- and map layout of R

**nameToPreorder:**

| A | B | C | D | E | F |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 13 |
| 9 | 10 | 7 | 6 |   |   |
| 11 | 8 |   |   |   |   |
| 12 |   |   |   |   |   |

**preorderToTupleID:**

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|
| - | s1 | s2 | - | - | s3 | s4 | - | s5 | - | - | s6 | - | s7 |

**preorderToMaxPreorder:**

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|
| 13 | 8 | 6 | 5 | 5 | 5 | 6 | 8 | 8 | 11 | 11 | 11 | 13 | 13 |

name:D
i:(7,8)

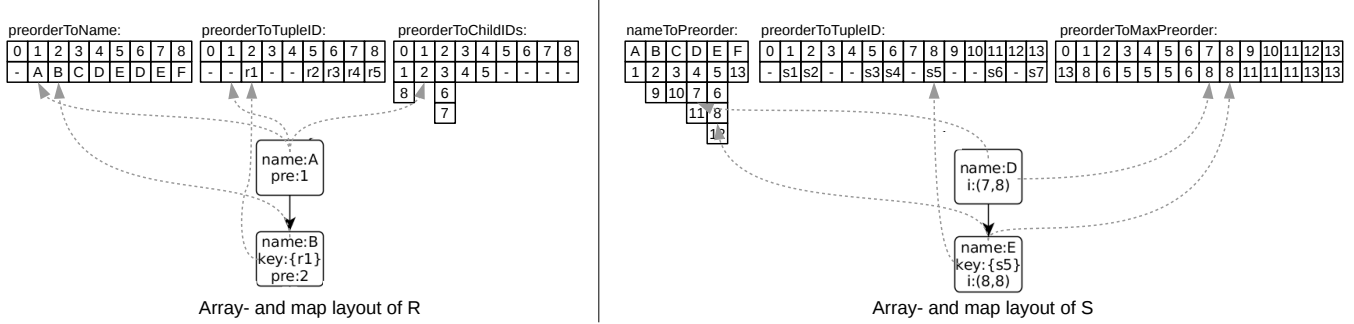name:E
key:{s5}
i:(8,8)

Array- and map layout of S

Figure 2: Array- and map based storage of prefix trees for $R$ (left) and $S$ (right).

*Set nodes are labeled with the IDs of the sets s they represent.*

*4. Every leaf of $T$ is a set node.*

We say that node $v$ is on level $l$ if the path from root to $v$ has length $l$ ($level(v) = l$). Furthermore, let $size(v)$ denote the total number of descendants of $v$. Figure 1 displays exemplary prefix trees for two relations $R$ (left) and $S$ (right). For example, the tuple $r2$ is represented by following the path from the root node to the leftmost leaf node in $T_R$.

When using prefix trees for indexing set-valued attributes, the definition of an appropriate sort order on all set items from $S$ is important. For SCJ, usually one of the following two sort orders is used. In the *frequent sort order*, elements are sorted in descending order of their overall frequency in $S$. In the *infrequent sort order*, elements are sorted in ascending order of their frequency. For example, we employ frequent sort order in Figure 1. Thus, the letter "A" as shown in the prefix trees corresponds to the most frequent and the letter "B" corresponds to the second most frequent set item of the entire domain of $R$ and $S$. Which of these two sort orders is the most efficient one depends on the actual data and the join algorithm being used (see Evaluation for details).

Tree signatures are an efficient means of representing tree data structures in a compact form [19]. In PIEJoin, we use preorder ranks and preorder intervals to linearize our data structures and to decide efficiently whether a certain set item is contained in some subtree. For any prefix tree $T$, the preorder sequence for all nodes $v \in V(T)$ is determined by traversing $T$ recursively in depth-first manner starting at $v_0$ from left to right. The increasing preorder rank $pre(v)$ is assigned to $v$ before $v$'s children are traversed.

DEFINITION 3 (PREORDER INTERVAL). *The preorder interval $i$ of a node $v \in V(T)$ is defined as $i(v) := [i_1(v), i_2(v)]$ with $i_1(v) := pre(v)$ and $i_2(v) := max(pre(c))$, where $c$ is a node in the subtree rooted at $v$.*

Using preorder intervals, we can now easily decide whether a name $a$ is contained in a sub-tree rooted at node $b$ because $a \in subtree(b) \Leftrightarrow i_1(b) \le i_1(a) \le i_2(b)$.

Consider the prefix tree for $S$ as shown in the right part of Figure 1 and the leftmost child of the root node, which is named "A" and labeled with the preorder interval $i = [1, 8]$. All nodes that are assigned a preorder value within this interval are contained in the sub-tree starting at node "A".

## 3. RELATED WORK

In the following we review the most important approaches for evaluating set containment joins, namely signature-, hash-, and tree-based algorithms and joins based on inverted indexes.

Signatures, also called superimposed coding [17], are a method to compactly represent sets using bit vectors. Instead of reserving one bit for every item of the domain, these algorithms use a much smaller vector, hashing each item of the domain to $k$ bits in this vector. A set is the bitwise-OR of the vectors of all its items. This enables to compare two sets very efficiently using bitwise operations. However, there is also a downside: this technique introduces false positives, which have to be filtered out in a second step. The main difference between signature-based algorithms is the way they store the signatures. Signature nested-loop and sequential signature file joins [6, 11] write them sequentially to a file, while signature tree joins utilize a tree structure [4].

Hash-based joins partition the tuples into different buckets using an appropriate hash function. The tuples found in corresponding buckets are then joined using one of the other techniques. The most prominent algorithms in this area are partitioned set join [16], adaptive pick and sweep join, divide-and-conquer set join, and adaptive divide-and-conquer set join [14, 15]. There is also some overlap with signature-based techniques. For instance, signature hash, extendable signature hashing, and lattice set join [5, 6, 13] use signatures for hashing the tuples into buckets.

Inverted indexes, also called inverted files, are an access method commonly used in information retrieval systems to index sets of keywords describing documents and retrieve supersets of keyword query sets [12]. They can also be adapted to other set query predicates and are used in the inverted file [5, 6], block nested-loop [11], and inverted file join [11] algorithms.

Finally, we come to the tree-based approaches. An early algorithm indexed the $R$ in a binary tree for faster lookup of items [5]. However, this simple technique is not faster than a nested-loop join. More sophisticated variants of this type of join, and closest to our own approach, are based on prefix trees. PRETTI uses a prefix tree to keep track of the tuples of $R$ and an inverted file index to keep track of those contained in $S$ [8]. The prefix tree is traversed in a depth-first manner and in each step a list of potential matching tuples (candidates) is pruned with data from the inverted file index on $S$, i.e., by intersecting the list of current candidates with posting lists. PRETTI+ is an extension employing a

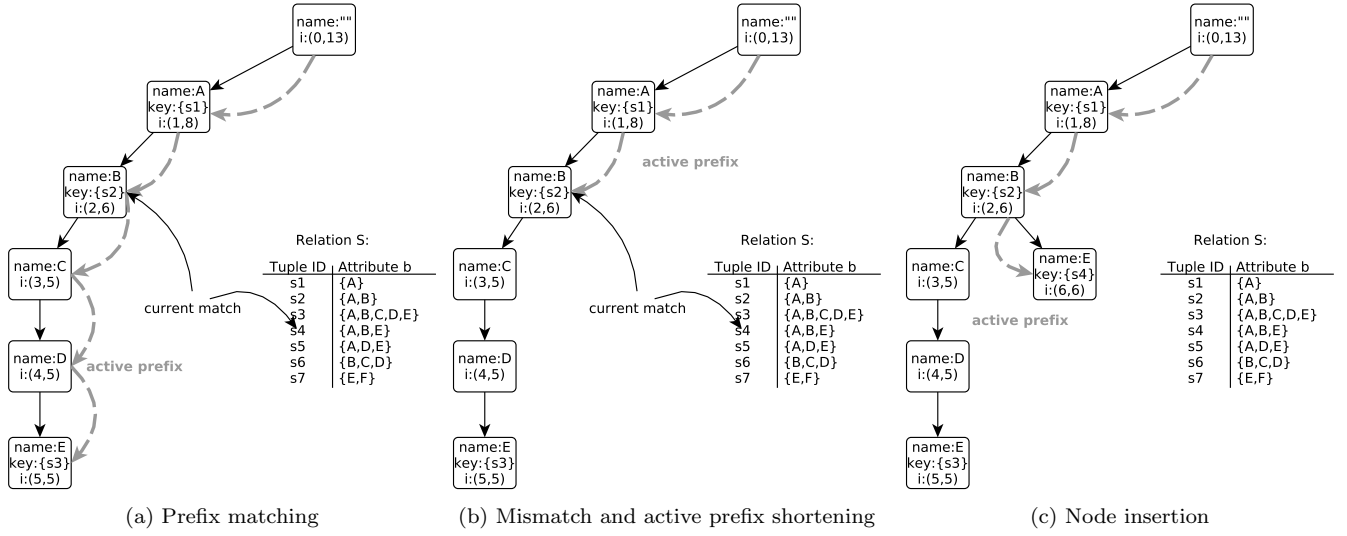(a) Prefix matching     (b) Mismatch and active prefix shortening     (c) Node insertion

Figure 3: Tuple insertion into $T_S$.

Patricia tree for implementing the prefix tree [10]. Bouros et al. [1] tackle two shortcomings of PRETTI: the prefix tree on $R$ can become quite large, making it infeasible to hold it in main memory, and the benefits of updating the candidates list via intersections with the posting lists has diminishing returns in later stages. The proposed LIMIT+ algorithm builds the prefix tree only up to a predefined parameter for maximum depth $\ell$ and employs a cost model to decide online whether to stop intersecting the candidates list with the posting lists. While this introduces false positives into the process, very often it is cheaper to verify candidates rather than going through all the steps. Bouros et al. also proposed a novel join paradigm termed *opj* based on the idea that the joining process can be interleaved with indexing which further reduces the space requirements, the list intersection cost and hence, the total execution time.

While PRETTI and PRETTI+ are tuning-free algorithms, LIMIT+(opj) greatly benefits from tuning $\ell$, the depth upon which the prefix tree is built. While Luo et al. [10] do not specify certain sort orders for set items for PRETTI+, Bouros et al. [1] report that infrequent sort order is beneficial for LIMIT+(opj). Note that since approximately 10 years, optimization of SCJ is carried out for the main-memory-based join part of the proposed algorithms. For large data sets and indexes, which do not fit into main memory entirely, Jampani and Pudi [8] proposed to horizontally partition the index and join partition pairs using one of the tree-based SCJ algorithms.

## 4. PIEJOIN

PIEJoin conceptually employs two prefix trees $T_R, T_S$ to compute SCJ between two set-valued attributes $R.a, S.b$. Let $p_r$ be a path in $T_R$ that represents all set items of a tuple $r \in R$ and let $p_s$ be a path in $T_S$ that represents all set items of a tuple $s \in S$. Recall that a set containment $r \subseteq s$ between $r$ and $s$ fulfills the requirements of the SCJ condition if all node names on $p_r$ are contained in the set of node names on $p_s$. In contrast to set equality joins, where all set items of $r$ and $s$ need to match, SCJ allows mismatching set items of $s$, since we only require $s$ to be a superset of the set items of $r$. Conceptually, we can find such pairs by computing the intersection of both trees with step-wise depth-first traversal of $T_R$ and $T_S$. A decision whether $r$ is contained in $s$ can only be taken if either a leaf node or a node $w \in V(T_S)$ is reached, whose name is greater than the largest node name on $p_r$. This often leads to long and unnecessary path traversals in $T_S$, since mismatching candidate pairs can only be excluded from the search space after many set items were compared. For example, when searching for the containment of tuple $r4 \in R$ from Figure 1 in $S$, we need to traverse the entire subtree starting at the leftmost child of the root node of $T_S$ to detect that $r4$ is not contained in any of the tuples of $S$.

To circumvent this drawback, PIEJoin uses preorder intervals of $T_R$ and $T_S$ to skip long paths in $T_S$ and access candidate nodes directly during tree traversal. In this section, we introduce data structures and present novel algorithms for index creation and join. Since space consumption of prefix-tree-based indexing methods is often large [9], we also give insights into implementation details of the PIEJoin to minimize index space consumption.

### 4.1 Data structures for space-efficient index creation

Before computing the actual join, PIEJoin creates a prefix tree index for each relation on the fly in main memory. To achieve space efficiency, all index structures are built using a few map data structures instead of creating object trees. For storing $T_R$, we use

- a map $preorderToName$, that assigns a node name to each preorder value,

- a map $preorderToTupleID$, that assigns a list of tuple IDs from $S$ to each preorder value, and

- a map $preorderToChildIDs$, that assigns a list of child node IDs (i.e., preorder values of child nodes) to each preorder value.

Similarly, we represent $T_S$ using

**Algorithm 1:** Index creation algorithm for $T_S$

---

**Input:** Sorted relation $S$
**Output:** PIETree $T_S$

1  $T_S \leftarrow$ new PIETree()
2  nodeID $\leftarrow 0$,
3  activePrefix $\leftarrow$ new Stack()
4  $T_S$.nameToPreorder[-1] $\leftarrow 0$
5  **foreach** $s \in S$ **do**
6     **for** $pos \leftarrow 0$ *to* $s.getSetLength() - 1$ **do**
7        item $\leftarrow s.getSet()$[pos]
8        **if** $activePrefix.size() > pos$ **and**
       $activePrefix.getName(pos) \neq item$ **then**
         /* Cut active prefix      */
9           **for** $i \leftarrow activePrefix.size()$ **to** $pos$ **do**
10             lastNode $\leftarrow$ activePrefix.pop()
11             $T_S$.preorderToMaxPreorder[lastNode] $\leftarrow$
            nodeID
12          **end**
13       **end**
      /* Insert node          */
14       **if** $\neg activePrefix.containsName(item)$ **and**
       $activePrefix.size() == pos$ **then**
15          nodeID $\leftarrow$ nodeID + 1
16          activePrefix.push([nodeID, item])
17          $T_S$.nameToPreorder[item] $\leftarrow$ nodeID
18       **end**
      /* Insert tuple ID      */
19       **if** $pos == s.getSetLength()-1$ **then**
20          $T_S$.preorderToTupleID[nodeID] $\leftarrow s.getID()$
21       **end**
22    **end**
23 **end**
24 **for** $i \leftarrow activePrefix.size()$ **to** $0$ **do**
25    last $\leftarrow$ activePrefix.pop()
26    $T_S$.preorderToMaxPreorder[last] $\leftarrow$ nodeID
27 **end**
28 $T_S$.preorderToMaxPreorder[0] $\leftarrow$ nodeID
29 **return** $T_S$

---

- a map *nameToPreorder* that assigns a preorder value to each node name,

- a map *preorderToTupleID*, similar to $T_R$, and

- a map *preorderToMaxPreorder*, which stores a preorder value $i$ and the maximum preorder value $i_{max}$ contained in the sub-tree starting at the node identified by $i$.

Storing preorder intervals for $S$ and preorder-child relationships for $R$ describes both sets uniquely. Figure 2 displays the map-based representation of the relations $R$ and $S$ introduced in Figure 1 by means of exemplary sub-trees. Dashed gray arrows indicate the corresponding map storage positions for the example nodes in each map.

Internally, PIEJoin uses native arrays and maps, as these data structures consume significantly less space compared to objects and references. Since preorder IDs are consecutive, the are represented by array indexes in many cases. Specifically, we can store mappings from preorder IDs to single items within a single array, and mappings from preorder IDs to lists of items can be stored using two arrays only. In the

first case, the maps *preorderToTupleID*, *preorderToName*, and *preorderToMaxPreorder* can be stored as plain arrays, where each array index corresponds to the preorder value that identifies a certain node in both prefix trees. In the latter case, we store the map *preorderToChildIDs* and *nameToPreorder* using two arrays for each map. The first array stores for each preorder value a reference to an index position in the second array where the actual data starts. Because of the consecutive order of preorder values, the next index position stored in the first array also determines the end position of the list. Consider the map *preorderToChildIDs* as shown in Figure 2. For storing the child node IDs for the preorder values 0 and 1, the first array stores the value 0 at index position 0, 2 at index position 1, and 3 at index position 2. Using this information, we can determine the actual IDs of child nodes from the second array by accessing the partial arrays $[0, 2[$ and $[2, 3[$, respectively.

In the next section, we explain index creation algorithms for $T_R$ and $T_S$ in detail using above-mentioned data structures.

## 4.2 Index creation

Index creation is conceptually similar for both relations $R$ and $S$, since in both cases, all tuples from the sorted relations $R$ and $S$ are inserted sequentially into $T_R$ and $T_S$ in $O(|R|)$ and $O(|S|)$ steps, respectively. All set items of one tuple are first sorted based on the frequent or infrequent sort order (cf. Section 2) and mapped to an alphabet, which is sorted lexicographically. This assures that all sets that share a common prefix in each relation are processed sequentially. For example, consider the tuples $s3, s4$ from relation $S$ in Figure 1. Set items are sorted in ascending lexicographical order and $s3$ precedes $s4$ since they share a common prefix $A, B$ and the mismatching infix $C$ is lexicographically smaller than $D$.

**Index creation for $S$.** Algorithm 1 displays the index creation algorithm for $S$. It is initialized with an empty prefix tree, a root node, and an empty prefix for all tuples of $S$ (lines 1–4). For each tuple $s \in S$, the algorithm first determines the common prefix with the previously considered tuple $s' \in S$ and stores it in an auxiliary data structure called *activePrefix*, which is initially empty. ActivePrefix stores the path from the root node in our implicit prefix tree to node $c$, which marks the current position in the tree $T_S$. Internally, activePrefix stores a list of pairs consisting of node ID and node name. In the following, we explain insertion of yet unseen tuples into $T_S$ based on the example shown in Figure 3. The right part of each insertion step shows the current processing state and the left part depicts the implicit prefix tree. As shown in the figure, we currently process tuple $(s3, \{A, B, E\})$ and the previous tuples $s0, s1, s2$ were already processed. Thus, the current activePrefix is $[A, B, C, D, E]$. First, the longest common prefix of tuple $s3$ and activePrefix is determined, which is $lcp = [A, B]$. Since both nodes $A$ and $B$ are already contained in $T_S$, these nodes do not need to be inserted again (cf. Figure 3(a)).

If the activePrefix is either longer than the length of the set item list in the current tuple or if items do not match, it needs to be shortened. After cutting these items (see lines 8–13), the corresponding tree nodes are finalized because of the initial sort order of our input relation. Consequently, no further child nodes are added to any of the finalized nodes and we store largest node ID occurring in the subtree start-

ing at our current node as the corresponding maximum preorder ID in the map $preorderToMaxPreorder$ (cf. line 11 and Figure 3(b)).

The activePrefix is too short if either the set item list of our current tuple is longer or if it contains different items than the activePrefix. Consequently, new nodes need to be inserted in $T_S$ (cf. lines 14–18). At the same time, new preorder values are generated by incrementing the $nodeID$ counter, the newly added nodes are added to the activePrefix, and the node name is stored in the array $nameToPreorder$. Figure 3(c) displays the creation of a new node in $T_S$. In this example, the new node $E$ is created as a child of $B$ and the activePrefix is extended to $[A, B, E]$.

Whenever a tuple has been processed completely, the map $preorderToTupleID$ is filled with the current preorder value $nodeID$ and a tuple ID (cf. lines 19–21). Since the new node $E$ is the last item of the current tuple in our example, it is annotated with the corresponding tuple ID $s4$. Once all tuples are processed, the current activePrefix needs to be reduced subsequently until empty by storing $maxPreorder$ values for each node on the corresponding path in $T_S$ (cf. lines 24–29).

**Index creation for $R$.** Although the index creation algorithm for $R$ is conceptually similar to Algorithm 1, we need modified data structures and algorithmic adjustments in some details. As explained in Section 4.1, preorder IDs are matched with corresponding node names and stored in an array $preorderToName$. Parent-child relationships are stored in a map structure $preorderToChildIDs$, which internally consists of two arrays. In contrast to accessing tuple IDs, it is not feasible to access parent-child relations in the order of storage in the array, since at least, the total number of child nodes needs to be known in advance before creating new nodes. This knowledge is not available when reading the input data sequentially and would require expensive look-ahead operations. Instead, we propose to use an auxiliary data structure during index construction(i.e., a list consisting of integer array lists), which is converted into the final $preorderToChildIDs$ map after reading all input data.

Let $r \in R$ be the a tuple that is inserted into the index. Similar to creating the PIETree for $S$, the common prefix of $activePrefix$ and $r$ can be ignored, since all nodes that correspond to the common prefix are already inserted into the tree. Similarly, if the $activePrefix$ is too long, unnecessary items are removed. Since $maxPreorder$ values are not saved in the index for $R$, newly created nodes do not need to be annotated with this information. If the $activePrefix$ is too short, new nodes are inserted into the tree together with the corresponding parent-child relationships, preorder values are generated, and the newly inserted node is marked as the new $activePrefix$. Once $r$ is completely inserted, a new entry in $preorderToTupleID$ is created consisting of the tuple ID of $R$ and the preorder value of the last created node.

## 4.3 Join Algorithm

Once index structures for $T_R$, $T_S$ are created in main memory, the PIEJoin algorithm is applied, which conceptually computes the intersection $T_R \cap T_S$ by simultaneous depth-first traversal of both prefix trees. Due to the compact storage of $T_R$ and $T_S$ in maps and arrays and preorder interval encoding of $T_S$, the search for descendants in $T_S$ breaks

---

**Algorithm 2:** PIEJoin algorithm.

**Input:** PIETree indexes $T_R$, $T_S$

1  search(0,0)
2  **Function** $search(preorder_v, preorder_w)$ **is**
3      lookForOutput($preorder_v$, $preorder_w$)
4      $children_v \leftarrow T_R$.getChildrenOf($preorder_v$)
5      **foreach** $v_i \in children_v$ **do**
6          $children_w \leftarrow T_S$.findPreorderIdsInScope( $T_R$.getName($v_i$), $preorder_w$)
7          **foreach** $w_j \in children_w$ **do**
8              search($v_i$, $w_j$)
9          **end**
10     **end**
11 **end**
12 **Function** $lookForOutput(preorder_v, preorder_w)$ **is**
13     $tuples_r \leftarrow T_R$.getTupleIDsByPosition($preorder_v$)
14     **if** $\neg\ tuples_r.isEmpty()$ **then**
15         $maxPreorder_w \leftarrow$ S.getMaxPreorder( $preorder_w$)
16         $tuples_s \leftarrow T_S$.getTupleIDsByPosition( $preorder_w$, $maxPreorder_w + 1$)
17         print $tuples_r \times tuples_s$
18     **end**
19 **end**

---

down to a binary search in sorted lists, which also allows us to prune large parts of the search space while computing the join. In the following, we explain the PIEJoin algorithm in detail, which is shown in Algorithm 2.

Matching pairs $(r, s)$ are determined by traversing $T_R$ in depth-first manner by invoking the $search$ function with the root nodes $v_0 \in V(T_R)$ and $w_0 \in V(T_S)$ of both trees (cf. line 1). The $search$ function first calls another function $lookForOutput$ (cf. lines 12–19) to determine whether matching tuples are found at the current nodes of $T_R$ and $T_S$ (cf. line 3).

Recall that for a positive join pair $(r, s)$, the set-valued attribute of $r \in R$ must be fully contained in the set-valued attribute of $s \in S$. Assume we found a pair of nodes $v \in V(T_R)$, $w \in V(T_S)$, where $v.name = w.name$ holds. In a first step, the function $lookForOutput$ determines whether node $v$ is annotated with a list of tuple IDs, indicating that when reaching $v$, we have at least seen one complete set of a tuple $r \in R$, which is contained in all tuples of $S$ stored in the sub-tree starting at node $w$ (cf. line 13). Next, PIEJoin evaluates the preorder interval annotated at $w$ and retrieves all tuples stored within this interval from the arrays $preorderToMaxPreorder$ and $preorderToTupleID$ (cf. lines 15–16). Finally, it returns the cross-product of both tuple ID lists as valid join pairs (line 17).

Subsequently, tree traversal is continued. Since we are interested in containments of the form $r \subseteq s$, all direct child nodes $v_i$ of $v$ need to be considered for finding join pairs, whereas child nodes of $w$ may possibly be skipped. PIEJoin recursively traverses $T_R$ (cf. lines 5 - 10) and retrieves for each child node $v_i \in V(T_R)$ of $v$, all nodes $w_j \in V(T_S)$, where $v_i.name = w_j.name$ holds and which are a descendant of $w$ (cf. line 6). These nodes $w_j$ occur in the subtree starting at $w$ and can be retrieved efficiently by first applying binary search to the array $nameToPreorder$ to find all preorder values for nodes with the given name $v_i.name$. Sub-

| Data set | Domain size | No. of tuples | Max. set size | Avg. set size | Join cardinality | Size in MB |
|----------|------------|---------------|---------------|---------------|------------------|------------|
| BMS | 1,657 | 515,597 | 164 | 6.53 | $3.2 * 10^9$ | 11.2 |
| Flickr | 810,660 | 1,680,490 | 102 | 9.78 | $1.6 * 10^9$ | 79.6 |
| Flickr-LC | 618,971 | 3,546,729 | 1,230 | 5.36 | $6.3 * 10^9$ | 132.9 |
| Kosarak | 41,270 | 990,002 | 2,497 | 8.10 | $5.5 * 10^{10}$ | 31.6 |
| Netflix | 17,770 | 480,189 | 17,653 | 209.25 | $1.6 * 10^8$ | 426.4 |
| Orkut | 15,293,693 | 1,853,285 | 2,958 | 57.16 | $1.9 * 10^6$ | 881.7 |
| Twitter | 1,318 | 371,586 | 687 | 65.96 | $1.2 * 10^8$ | 82.3 |
| Webbase | 15,146,263 | 168,707 | 3,842 | 463.64 | $2.3 * 10^7$ | 709.6 |

Table 1: Key figures of data sets used in the evaluation.

sequently, these nodes are filtered by preorder values that lie within the boundaries of the preorder interval defined at $w_j$ (i.e., all descendants of $w_j$). For each of these pairs $v_i, w_j$, PIEJoin determines the corresponding preorder interval, calls the function $lookForOutput$ to emit possible join pairs, and recursively repeats the procedure.

Consider the example from Figure 1 and Figure 2 and assume that the node $v$ with name "A" is reached in in $T_R$ (cf. Figure 1, left side). This node $v_1$ has only one child with name "B", which is also annotated with the key $r_1$. In $T_S$ (cf. Figure 1, right side), we also reached the node $w$ named "A", which is annotated with the key $s_1$. The algorithm now retrieves all descendants of $w$ in $T_S$, which are also labeled with the letter "B". Since only one such node $w_1$ exists (cf. Figure 1, leftmost branch in the prefix tree on the right side), PIEJoin now determines the corresponding preorder interval $i(w_1) = [2, 6]$. Subsequently, the function $lookForOutput$ is called, tuple IDs stored in the interval $i(w_1)$ are retrieved (i.e., $s2$, $s3$, and $s4$) and the join pairs $(r_1, s_2), (r_1, s_3), (r_1, s_4)$ are returned. Subsequently, the algorithm continues to recursively traverse $T_R$ in depth-first manner and retrieve matching nodes from $T_S$.

## 5. EVALUATION

We compare PIEJoin to the three currently best main-memory-based SCJ algorithms, namely PRETTI [8] in the version provided by Luo et al. [10], PRETTI+ introduced by Luo et al. [10], and LIMIT+(opj) introduced by Bouros et al. [1]. Since LIMIT+(opj) was only available as a C++ library, we reimplemented the algorithm in Java. Parameter tuning (parameter $\ell$ for maximal tree depth) was carried out prior to our evaluation manually and individually for each data set. All algorithms were implemented in Java and compiled to Oracle Java SE 7. Experiments were carried out on a multi-core server with 1 TB RAM and 4 Intel R Xeon R E7-4870 CPUs, each equipped with 20 threads. If not stated otherwise, each experiment was repeated at least 3 times and we report the average of all runs.

**Datasets.** We evaluated PIEJoin and its competitors on eight different data sets (see Table 1). Specifically, we evaluated all algorithms on

- BMS, a collection of click-stream data [20],

- Flickr, a collection of tags and titles of pictures taken in London over a period of two years,

- Flickr-LC, a modified version of Flickr that models low set cardinalities,

- Kosarak, a collection of click-stream data taken from a Hungarian news portal,

- Netflix, a collection of user ratings on movie titles collected over a period of seven years,

- Orkut, a collection of network memberships with at least 10 set items per tuple in an online community,

- Twitter, a data set partitioning the Twitter graph based on node neighborhood information, and

- Webbase, a collection of web pages with at least 200 hyperlinks taken from the Stanford WebBase project [7].

The data sets Flickr-LC, Orkut, Twitter, and Webbase were obtained from [10], while BMS, Flickr, Kosarak and Netflix were obtained from [1]. Executables, source code for all evaluated algorithms, and all experimental data can be obtained from our website under the address https://www.informatik. hu-berlin.de/de/forschung/gebiete/wbi/resources/piejoin.

### 5.1 Join performance

We evaluated the performance of the algorithms on all data sets using both frequent and infrequent sort order. Indexes were created on the fly in main memory and time for index creation is therefore included in all measurements. Results are shown in Figure 4. In summary, PIEJoin is faster than PRETTI and PRETTI+ in seven our of eight data sets, while LIMIT+(opj) is faster than PIEJoin in 11 out of 16 runs (data set plus sort order). Contradicting results from [10], which claim PRETTI+ to be faster than PRETTI (evaluating on data sets Flickr-LC, Orkut, Twitter, and Webbase with unknown sort order), our results indicate that PRETTI is actually faster than PRETTI+ for most runs. The impact of sort order is notable for all four methods.

In detail, PIEJoin and LIMIT+(opj) benefit from infrequent sort order in regards of execution time for most of the data sets. This is most evident for the Netflix data set, as both methods execute the join 13 times faster compared to frequent sort order. While PIEJoin performs up to 2 times faster on two sets (Twitter, Webbase) and, remarkably, 5 times faster on the Kosarak set, LIMIT+(opj) needs more execution time on all of the data sets when using frequent order. The importance of choosing the best sort order can be seen in the measurements for PRETTI, since its performance drops significantly on all data sets when using infrequent sort order.
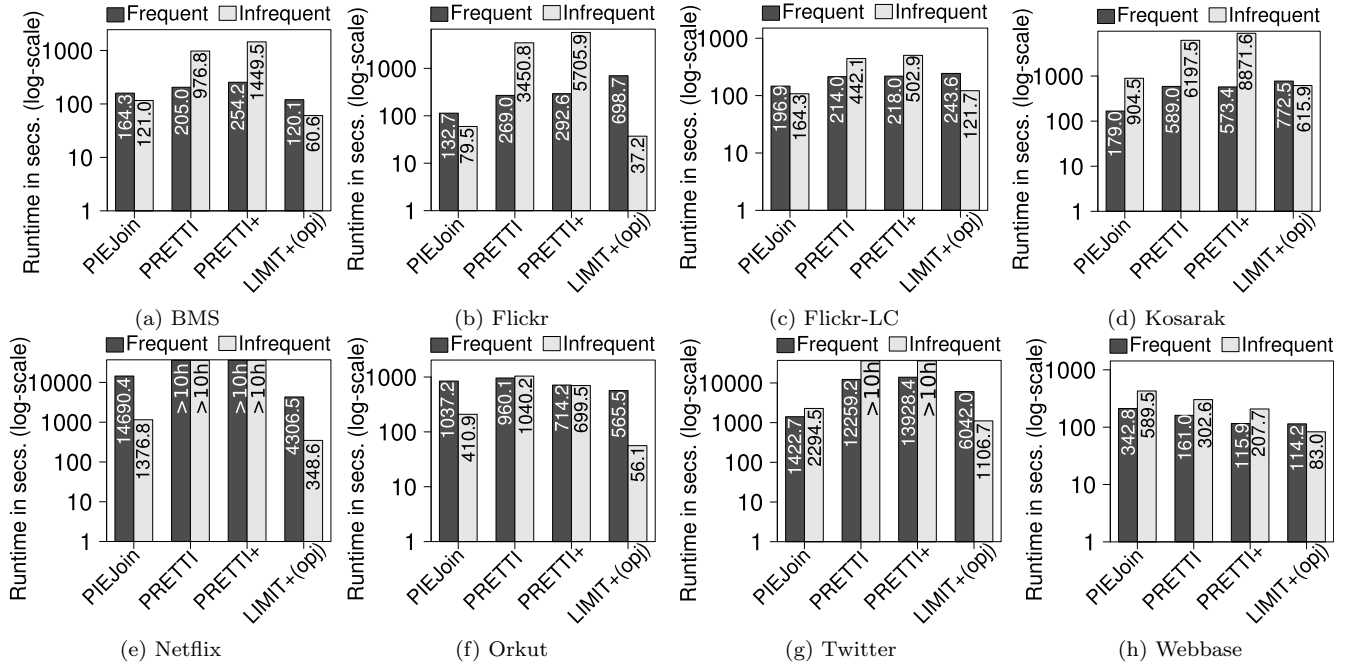
Figure 4: Average execution time in seconds (log-scale) of different SCJ algorithms on different data sets for two sort orders.

Analyzing results for the infrequent sort order more deeply, we see that LIMIT+(opj) finishes fastest with only one exception, while PIEJoin is slightly faster on Flickr-LC. PIE-Join achieves execution times very close to LIMIT+(opj) on three data sets (BMS, Flickr, Kosarak) and is up to 5 times slower for the remaining data sets. PRETTI and PRETTI+ show competitive results only on the Webbase data set. For BMS, Flickr, Flickr-LC, Kosarak, and Orkut, PRETTI and PRETTI+ compute the join slower by a factor of up to 38 and even more strikingly, both algorithms were not able to finish on the Netflix and Twitter data sets within 10 hours for infrequent sort oder (note that [10] probably evaluated on a different sort order).

Results for frequent sort order differ noticeably. PIEJoin is fastest on four data sets (Flickr, Flickr-LC, Kosarak, Twitter) and LIMIT+(opj) is the fastest algorithm on the remaining data sets BMS, Netflix, Orkut, and Webbase. PRETTI and PRETTI+ achieve considerably better results compared to infrequent sort order and their execution times are much closer to PIEJoin and LIMIT+(opj) compared to infrequent sort order. For frequent sort order, only a few significant differences between the methods stand out. For example, LIMIT+(opj) is slower by a factor of 6 and 4 for the Flickr and Twitter data compared to PIEJoin, but 3 times faster for the Netflix data set, for which PRETTI and PRETTI+ were aborted after 10 hours without terminating. PRETTI and PRETTI+ achieve competitive results for five sets (Kosarak, Flickr, Flickr-LC, Webbase, Orkut), yet still are always outperformed by either LIMIT+(opj) or PIEJoin. On most data sets, however, results for the infrequent sort order are considerably faster.

We found that set sizes are strongly connected to the performance of each algorithm on frequent sort order. LI-MIT+(opj) benefits from data sets where tuple are rather large on average (i.e., is fastest with the Netflix, Webbase and Orkut), while PIEJoin, PRETTI and PRETTI+ are faster for smaller sets (Kosarak, Flickr, Flickr-LC, Twitter). This observation can not be easily transferred to the results for infrequent sort order, since the differences between the algorithms are more consistent over all data sets in this case.

## 5.2 RxS joins

Next to self-joins, we evaluated all algorithms regarding their performance on $RxS$ joins, which were not considered in [1] and [10]. Therefore, we split one of the larger data set, Netflix, using random sampling into two subsets $R$, $S$ using the following configurations:

- 10-90: $R$ holds 10%, $S$ holds 90% of the tuples,
- 30-70: $R$ holds 30% of the tuples, $S$ holds 70%,
- 50-50: $R$ and $S$ both hold 50% of the tuples,
- 70-30: $R$ holds 70% of the tuples, $S$ holds 30%, and
- 90-10: $R$ holds 90%, $S$ holds 10% of the tuples from the original data set.

Figure 5 displays results for the $RxS$ join using infrequent sort order, which proves to be beneficial for all algorithms. LIMIT+(opj) outperforms all the other methods without exception for each set cardinality, with factors of 4 to 7 compared to PIEJoin, and 6 to 114 in comparison to PRETTI and PRETTI+. The differences between the algorithms change quite notably with varying set cardinality. LIMIT+(opj) and PIEJoin behave in a similar manner by performing fastest for the 90-10 and 10-90 splits and slower for the other distributions by a small factor. PIEJoin proves to be stable regarding changing set sizes, with the fastest and slowest execution times differing only by a factor of 1.7. LIMIT+(opj) is slightly behind with a factor of 3.2. In contrast, the performance of PRETTI and PRETTI+ drops significantly with increasing sizes of $R$.
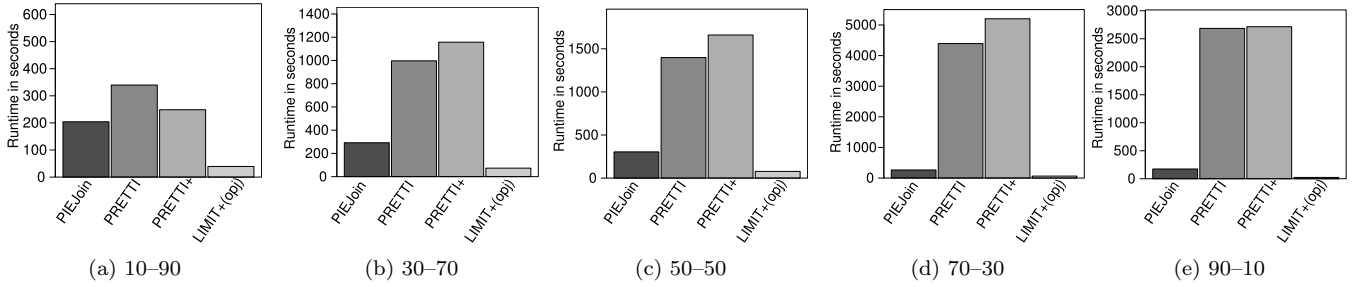
| (a) 10–90 | (b) 30–70 | (c) 50–50 | (d) 70–30 | (e) 90–10 |

Figure 5: Total average execution time in seconds of different SCJ algorithms for $RxS$ joins on different configurations.

## 5.3 Index space consumption

We evaluated the index space consumption including space overhead for join processing of all methods on all of our data sets on both two sort orders, yet show only a fraction of the results here for space constraints. Note that space consumption is highly implementation-dependent. Thus, we do not claim that any of the measured algorithms is conceptually more or less space demanding than the others; instead, we can only compare the available implementations, which took different efforts for reducing space (or none at all). Clearly, the implementation of PIEJoin was highly engineered to reduce space and to fit more data into main memory to be usable for larger data sets. This effort yields in very good results regarding space (see below), yet we believe that the tree representation used in PIEJoin would also greatly reduce the space requirements of the other methods.

Exemplary results for two data set are displayed in Figure 6 (please note the log-scale). For each algorithm, we found that indexes built using frequent sort order consume less space than indexes built using the infrequent sort order. For LIMIT+(opj), the difference between sort orders is most striking as it consumes between 2 (Netflix) and up to 7.6 times (Twitter) more space when using infrequent sort order. The only exception is the Webbase data set, where index space consumption remains stable for LIMIT+(opj) regardless of sort order. For all other algorithms, space consumption is less affected by sort order, i.e., both the PIEJoin and PRETTI index on infrequent sort order consumes at most 2.3 and 2.5 times more space on the Twitter data set, respectively. PRETTI+ is not influenced by changes in item sort order, we only observed minor size differences, which can be attributed to measuring inaccuracies.

Regarding total index space consumption, we observe that PIEJoin requires the least amount of space on five data sets regardless of sort order. Compared to PRETTI, PIEJoin requires significantly less space with factors between 4.7 (Webbase) and 12.3 (BMS) using frequent, and factors between 6.2 (Webbase) and 12.3 (Flickr-LC) using infrequent sort order for all data sets. PRETTI+ requires less space than PRETTI, but still, PIEJoin is substantially more space efficient (factors between 2.1 and 6.7 on frequent sort order, and 1.3 and 6.4 on infrequent sort order) in all cases. The space requirements of PIEJoin compared to LIMIT+(opj) strongly depend on the characteristics of the indexed data sets. PIEJoin is advantageous in terms of index space consumption for data sets with rather small set sizes (avg. set size $< 60$, max. set size $< 3000$), whereas LIMIT+(opj) has slight advantages on data sets with rather large set sizes. On data sets with small set sizes (BMS, Flickr, Flickr-LC,
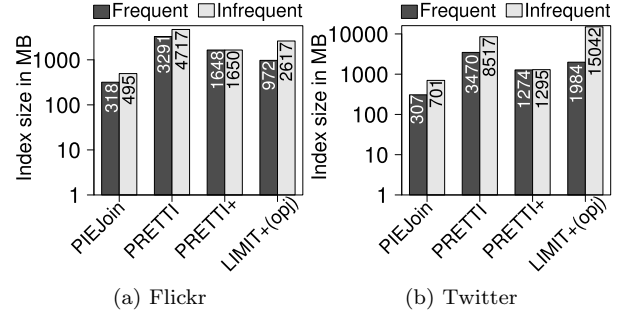


| (a) Flickr | (b) Twitter |

Figure 6: Index space consumption in MB (log-scale) of different SCJ algorithms for two sort orders.

Kosarak, Twitter), PIEJoin needs significantly less space than LIMIT+(opj) with factors between 3.1 (Kosarak) and 6.5 (Twitter) using frequent, and with factors between 5.3 (Flickr) and 21.4 (Twitter) using infrequent sort order. On the data sets Netflix, Orkut, and Webbase, LIMIT+(opj) needs between 12% (Webbase) and 56% (Netflix) less space than PIEJoin on frequent sort order and between 11% (Netflix) and 14% (Orkut) on infrequent sort order.

## 6. TOWARDS PARALLEL SCJ

The single-threaded join execution time of all algorithms, as evaluated in the previous chapter, easily reaches several hours when faced with giga-byte size data sets, which are more and more common in today's applications. Although previous works [1, 10] briefly mentioned the potential of parallelization for speeding-up the respective algorithm, we are not aware of any previous implementation or evaluation of parallel SCJs.

In this section, we highlight important aspects regarding parallelization of the PIEJoin algorithm, present an initial parallelization strategy, and report on evaluation results using 1 to 64 parallel threads. In particular, we discuss index partitioning and task composition to ensure a roughly even work distribution among parallel threads with the aim to avoid stragglers.

### 6.1 Preliminaries

Before deciding upon a parallelization strategy for PIE-Join, we first analyzed which phases of the algorithm contribute to what extent to the total single-threaded execution time. We found that for all data sets, almost 95% of the total execution time of PIEJoin is spent during join computation, whereas only 5% of the time is needed for creating index

structures. Analyzing the join phase more deeply, we observed that the time needed to process one level of the index tree is not distributed evenly across the different tree levels. For the data sets BMS, Flickr-LC, and Kosarak, PIEJoin spends most time on processing the upper levels of the trees, whereas more than 65% of the join phase is spent on processing tree levels deeper than $l = 20$ for data sets with an average set size of more than 50 items (Netflix, Orkut, and Webbase). For these reasons, we apply parallelization only to the join phase and ensure that tree partitioning starts at level 1 and is also carried out for deeper levels of the tree.

Ideally, parallelization reduces the time needed for computing SCJ proportional to the number of available threads such that no threads run idle and the number of tasks is appropriate to the degree of parallelization and workload. This optimum is in practice hardly reachable, since overhead is introduced by partitioning the entire workload, thread management, and skew in the frequency distributions of the data sets to be joined. We consider different parallelization strategies, namely (a) computing one tree traversal step per task, (b) processing one fixed sub-tree per task, and (c) processing groups of nodes per task.

Applying option (a) to PIEJoin creates one task for each recursive call of the *search* function, which optimally partitions the entire workload, but introduces an enormous overhead for task management. Consequently, the number of tasks is identical to the number of recursive calls of the search function and ranges between millions and billions of tasks depending on the data set. When partitioning the prefix trees for $R$ and $S$ on the first level and assigning each pair of sub-trees to a task (option (b)), the total number of tasks and the overhead necessary for task management decreases, but the size of each task increases significantly. Moreover, partitioning the index on level 1 only creates tasks of unequal size, since none of the data sets available for evaluation is equally distributed and the size and number of matches across the sub-trees differs heavily. Plus, this partitioning scheme still creates many tasks depending on alphabet size, e.g., more than 95 million tasks are created for the Netflix data set, which still induces a significant task management overhead. Therefore, we deem option (c), processing groups of nodes using an adaptive partitioning scheme to create a limited number of tasks of similar size, as the most promising alternative.

**Task size estimation for skewed distributions.** For all available data sets, we analyzed the frequency distributions and found that in each data set, neither set size nor set item frequency is equally distributed. Particularly, infrequent set items are only contained in a few tuples and thus, sub-trees starting at such an item are comparatively small, whereas frequent set items in the upper levels of the tree contain sub-trees of all those tuples that do not contain rarer set items. Thus, sub-trees starting at nodes that represent frequent set items potentially contain more tuples and nodes. Moreover, nodes representing frequent set items in $T_R$ have many matches in $T_S$ in self-joins, which requires an enormous amount of execution time even if mismatches occur at an early stage of processing. This leads to search spaces of greatly varying size and makes index partitioning and task size estimation increasingly difficult.

To estimate the execution time needed to join a node in $T_R$ with $T_S$, we found that the frequency of a node label in the data set is a suitable measure for many data sets, which

roughly adhere to a Zipfian or binomial distribution. As shown in Figure 7, we empirically observed a linear correlation between the frequency of a node name and execution time needed to compute the partial SCJ for a sub-tree at level 1 of $T_R$ with this label for the cause that the distribution of node labels is even for $T_R$ and $T_S$ (e.g., in self-joins). Using node label frequency and a targeted number of tasks, we introduce a parallelization strategy for PIEJoin in the next section.

## 6.2 Parallel SCJ by dynamic range partitioning

---

**Algorithm 3:** Parallelization of PIEJoin using dynamic node ranges and splitting tasks.

---

**Input:** $T_R$, node name frequencies $frequencyMap$ for $S$, scheduler $sched$, $v_0 inR$, $w_0 inS$

1  $nodes_R \leftarrow$ R.getChildrenOf($v_0$)
2  idealRangeSize $\leftarrow \frac{R.level1FrequencySum}{sched.rangeFactor \times sched.nThreads}$
3  $currentRangeStart \leftarrow 0$
4  $currentRangeFreqs \leftarrow 0$
5  **for** $i \leftarrow 0$ **to** $|nodes_R|$ **do**
6     $n_R \leftarrow nodes_R[i]$
7     $nname_R \leftarrow$ R.getNameOf($n_R$)
8     $freq \leftarrow$ frequencyMap.get($nname_R$)
9     **if** $currentRangeFreqs + freq < idealRangeSize$
   **then**         // add nodes to partition
10       $currentRangeFreqs \leftarrow currentRangeFreqs + freq$
11     **else**
12       **if** $currentRangeStart == i$ **then**
13         sched.addSplittingTask($n_R$) // split large
14         $currentRangeFreqs \leftarrow 0$        // nodes
15         $currentRangeStart \leftarrow$ i+1
16       **else**         // start a new range
17         sched.addSearchTask(
18            child nodes of $v_0$ in the interval $[currentRangeStart, i - 1]$,
19            matches of $v_0$ in the sub-tree $w_0$)
20         $currentRangeFreqs \leftarrow freq$
21         $currentRangeStart \leftarrow$ i
22       **end**
23     **end**
24  **end**
25  **if** $currentRangeStart \neq |nodes_R|$ **and** $|nodes_R| > 0$
   **then**
26     sched.addSearchTask(    // Finalize last range
27       child nodes of $v_0$ in the interval $[currentRangeStart, |nodes_R| - 1]$,
28       matches of $v_0$ in the sub-tree $w_0$
29     )
30  **end**
31  lookForOutput($v_0$, $w_0$)

---

We parallelize PIEJoin using a shared-memory model, where multiple tasks work on the same index simultaneously. Task synchronization is not necessary, since all tasks access the index in a read-only manner. A central scheduler maintains a task queue and the available threads. Read-only access is performed on all index structures, write access is only necessary for collecting the partial result sets after each
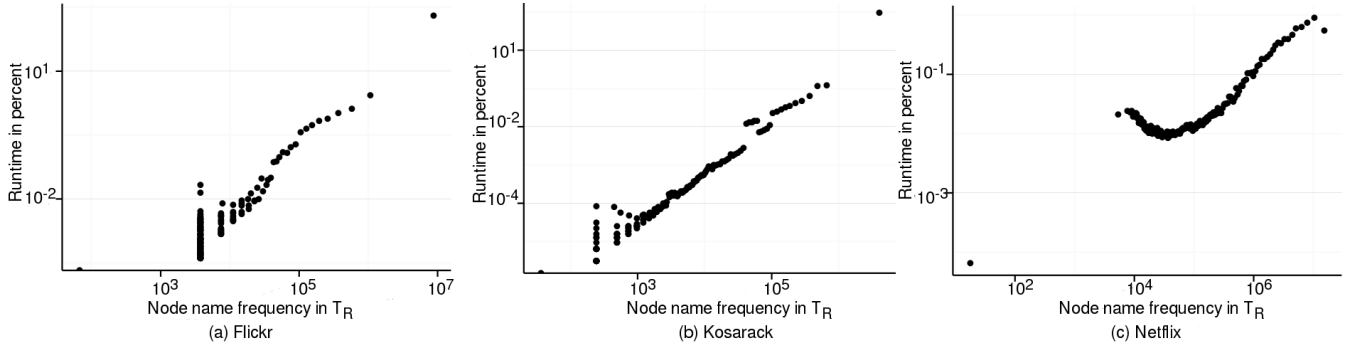
Figure 7: Percentage-wise execution time of nodes of $T_R$ relative to the node name frequency.

task is finished. The initial task, which processes the root nodes of $R$ and $S$, is passed to the queue. The scheduler distributes the tasks from the queue to the various existing threads. Each tasks processes its assignment and possibly generates new tasks, which are added to the queue for later processing. The search phase of PIEJoin ends if the queue is empty and if all running tasks are finished.

When parallelizing PIEJoin, we aim at generating partitions of similar execution time while not creating excessively many tasks. To accomplish this, we create node ranges dynamically based on a parameter $rangeFactor$, which defines a factor of the minimum number of necessary partitions based on the given degree of parallelism. For example, a $rangeFactor$ of 5 and a degree of parallelism of 10 indicates that at least 50 partitions are created, which are subsequently processed by the different threads.

In Algorithm 3, partitioning is carried out greedily based on the order in which tree nodes are stored in the arrays. Note that our partitioning scheme cannot guarantee optimality in all cases, since finding optimal partitions is NP-hard [2]. Nevertheless, it is efficiently computable and empirically creates partitions of similar workload in many cases. During initialization, the algorithm starts at the root node of $R$ and groups node ranges to individual partitions, which are processed by the available threads. Computation of ranges is carried out based on node name frequency (cf. line 8), the targeted number of partitions, and partition size (cf. line 2). The list of all nodes is traversed and the first range partition is created at the first node. For each visited node $v_i$, the algorithms tests whether the current range partition is still smaller than the optimal size and, if successful, adds the node to the current partition. If adding $v_i$ to the current partition exceeds the optimal partition size, the currently created partition is finalized, handed to the scheduler for processing, and a new partition is started at $v_i$. Using this approach, we create at least as many partitions as targeted initially, but often, some additional partitions are created, which never exceed the targeted size. This prevents the creation of very large tasks, which potentially slow down the entire algorithm.

Due to the skewed distributions of our real-world data sets, it might happen that single nodes in $R$ already contain very large and computational-intensive sub-trees, i.e., all nodes of $R$ on level 1, which form a separate partition because of their name frequency. Therefore, we additionally split such large tasks for nodes $v_j$ on level 2 and create partitions in the same manner as explained previously by grouping child nodes into ranges, creating new partitions, and finally, to process $v_j$ (Algorithm 3, lines 12 – 15). In PIEJoin, we use splitting tasks only on level 2, since splitting tasks are predominantly useful for separating tree traversal from collecting matching tuples for $v_j$. Since the number of matches for each tuple of $R$ decreases with the depth in the index tree, we do not employ splitting tasks on any level deeper than 2. We leave a more adaptive strategy to future work.

## 6.3 Evaluation

We evaluated the parallel version of PIEJoin regarding speed up on all data sets using varying range factors and degrees of parallelism on the same platform as for single-threaded execution. Compared to single-threaded execution, we observed speed ups between 2.7 (Kosarak) and 18.3 (Netflix) using 64 threads for range factors between 1 and 10 compared to the single-threaded version.

Figure 8 displays the speed up of PIEJoin for different range factors and varying degrees of parallelism between 4 and 64 for the data sets BMS, Kosarak, and Netflix. We clearly see that parallelization does not pay off equally well for each data set and that choosing an appropriate range factor is essential in most of our evaluation data sets. The only data set standing out of this observation is Netflix, where significant speed ups were reached independent of the chosen range factor. For the other data sets, however, we observed that smaller range factors, which create fewer partitions, are often beneficial. One possible explanation for this behavior can be found in the distribution of the set items in the different data sets. Netflix corresponds rather to a Poisson distribution, whereas all other data sets rather approach a Zipfian distribution. Another explanation for low speed ups is the existence of straggling tasks. Although splitting tasks diminished the size of individual partitions, we still found sub-tasks that dominated the total runtime of PIEJoin and prevent higher speed ups for increasing degrees of parallelism. For example, the Kosarak data set contains a singular task, which dominates the entire execution time, and which could not be reduced significantly even by employing splitting tasks. Compared to the single-threaded LIMIT+(opj), the parallel version of PIEJoin achieves speed-ups 28% (BMS) and 583% (Netflix).

## 7. CONCLUSIONS

We presented PIEJoin, a novel algorithm for fast computation of memory-based set containment joins together with
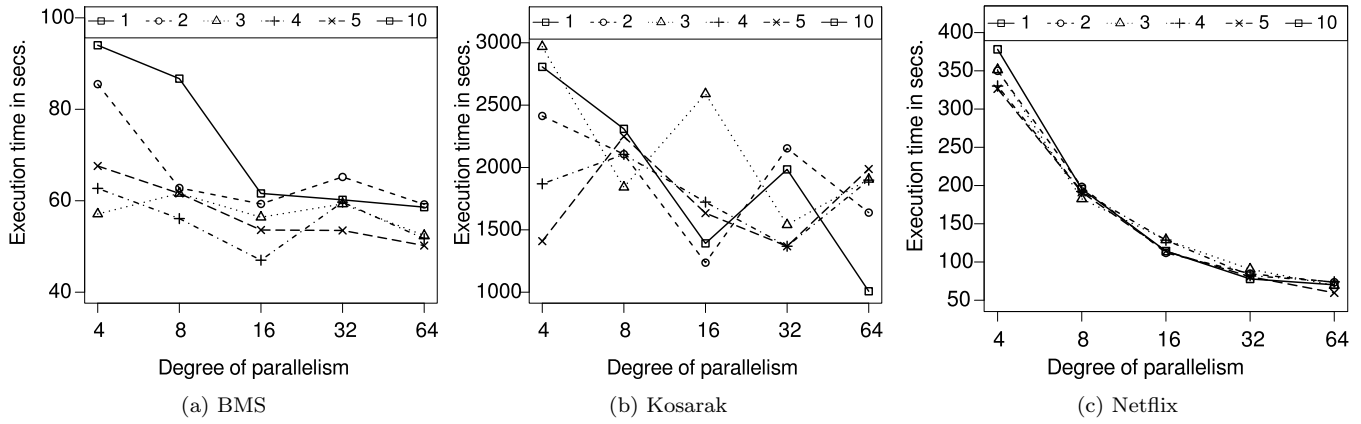
Figure 8: Average parallel execution time of PIEJoin for varying degrees of parallelism and range factors.

a highly space efficient implementation. PIEJoin is considerably faster than two out of three of its main competitors as evaluated on eight different data sets, while comparison the third method, LIMIT+(opj), leads to mixed results with a slight edge for the latter. Most notably, PIEJoin's basic structure opens the door for efficient parallel implementations of SCJ. In this regard, we presented the, to our knowledge, first empirical study on the potential and pitfalls of parallel SCJ. Our results show the importance of the trade-off between strategies that try to accurately estimate runtimes of sub-tree intersections – which can be costly to compute – or that try to be fully adaptive – which leads to increased administration overhead at runtime. Clearly, our results in this regard are preliminary and call for further research into this timely topic.

# 8. REFERENCES

[1] P. Bouros, N. Mamoulis, S. Ge, and M. Terrovitis. Set containment join revisited. *Knowledge and Information Systems*, pages 1–28, 2016 (to appear).

[2] S. Chopra and M. R. Rao. The partition problem. *Math. Program.*, 59(1):87–115, 1993.

[3] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[4] U. Deppisch. S-tree: A dynamic balanced signature index for office retrieval. In *Proc. of the 9th Ann. Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 77–87, 1986.

[5] S. Helmer and G. Moerkotte. Evaluation of main memory join algorithms for joins with set comparison join predicates. In *Proc. of the 23rd Int. Conf. on Very Large Data Bases*, pages 386–395, 1997.

[6] S. Helmer and G. Moerkotte. A performance study of four index structures for set-valued attributes of low cardinality. *VLDB Journal*, 12(3):244–261, 2003.

[7] J. Hirai, S. Raghavan, H. Garcia-Molina, and A. Paepcke. Webbase: A repository of web pages. *Computer Networks*, 33(1-6), 2000.

[8] R. Jampani and V. Pudi. Using prefix-trees for efficiently computing set joins. In *Proc. of the 10th Int. Conf. on Database Systems for Advanced Applications*, pages 761–772, 2005.

[9] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *Proc. of the 2013 IEEE Int. Conf. on Data Engineering*, pages 38–49, 2013.

[10] Y. Luo, G. H. L. Fletcher, J. Hidders, and P. D. Bra. Efficient and scalable trie-based algorithms for computing set containment relations. In *Proc. of the 31st IEEE Int. Conf. on Data Engineering*, pages 303–314, 2015.

[11] N. Mamoulis. Efficient processing of joins on set-valued attributes. In *Proc. of the 2003 ACM SIGMOD Int. Conf. on Management of Data*, pages 157–168, 2003.

[12] C. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval.* Cambridge University Press, 2008.

[13] S. Melnik and H. Garcia-Molina. Divide-and-conquer algorithm for computing set containment joins. Technical Report, Stanford Infolab, http://ilpubs.stanford.edu:8090/502/, 2001.

[14] S. Melnik and H. Garcia-Molina. Divide-and-conquer algorithm for computing set containment joins. In *Advances in Database Technology - EDBT 2002, 8th International Conference on Extending Database Technology, Proceedings*, pages 427–444, 2002.

[15] S. Melnik and H. Garcia-Molina. Adaptive algorithms for set containment joins. *ACM Trans. Database Syst.*, 28(1):56–99, 2003.

[16] K. Ramasamy, J. M. Patel, J. F. Naughton, and R. Kaushik. Set containment joins: The good, the bad and the ugly. In *Proc. of the Int. Conf. on Very Large Databases*, pages 351–362, 2000.

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

[18] M. Stonebraker and D. Moore. *Object Relational DBMSs: The Next Great Wave.* Morgan Kaufmann Publishers Inc., 1995.

[19] P. Zezula, G. Amato, F. Debole, and F. Rabitti. Tree signatures for xml querying and navigation. In *Database and XML Technologies*, volume 2824 of *LNCS*, pages 149–163. 2003.

[20] Z. Zheng, R. Kohavi, and L. Mason. Real world performance of association rule algorithms. In *Proc. of the 7th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 401–406, 2001.