

# Using Prefix-trees for Efficiently Computing Set Joins

Ravindranath Jampani

Vikram Pudi

Center for Data Engineering,  
International Institute of Information Technology, Hyderabad, India  
ravi@students.iiit.net, vikram@iiit.net

**Abstract.** Joins on set-valued attributes (set joins) have numerous database applications. In this paper we propose PRETTI (PREfix Tree based seT joIn) – a suite of set join algorithms for containment, overlap and equality join predicates. Our algorithms use prefix trees and inverted indices. These structures are constructed on-the-fly if they are not already precomputed. This feature makes our algorithms usable for relations without indices and when joining intermediate results during join queries with more than two relations. Another feature of our algorithms is that results are output continuously during their execution and not just at the end. Experiments on real life datasets show that the total execution time of our algorithms is significantly less than that of previous approaches, even when the indices required by our algorithms are not precomputed.

## 1 Introduction

Set-valued attributes are a natural and concise representation for many real life models. Object Oriented and Object Relational DBMS require the support of set-valued attributes. Efficient execution of queries involving these attributes is therefore an important problem [14]. Set join [11, 10, 7, 6, 13] is perhaps the most important operator on set-valued data since it is useful in several real-world problems, while at the same time being difficult to compute [2]. A set join between two relations  $R$  and  $S$ , retrieves pairs of records  $(t_R, t_S)$ ,  $t_R \in R$  and  $t_S \in S$ , for which  $t_R.p \theta t_S.q$  returns true, where  $p$  and  $q$  are set-valued attributes and  $\theta$  can be any boolean valued function over two sets. Examples of such functions include set containment, set equality and set overlap.

As an example of a set containment join, consider the join of a relation *students* with a relation *courses*, where *students* has a set-valued attribute *coursesTaken*, and *courses* has a set-valued attribute *prerequisites*. This join finds all students eligible for taking each course when the predicate is  $courses.prerequisites \subseteq students.courses$ . Overlap joins can be very useful in any match-making domain. One example is to find pairs of customers of amazon.com who purchase at least 5 books in common.

A number of partition based [11, 10] and inverted index based approaches [7, 6, 13] have been proposed for set joins. These studies were a welcome first step in addressing the problem of efficient computation of set joins. The partition based approaches do not require any precomputed index, but they are in general not as efficient as the inverted index based approaches [7]. The state-of-the-art inverted index based approaches, while being more efficient, have the drawback of requiring a precomputed index – this slows down database updates.

Our contributions in this paper are as follows:

1. We propose PRETTI (PREfix Tree based seT joIn) – a suite of novel algorithms for set containment, overlap and equality joins.
2. Our algorithms use prefix trees [4] in addition to inverted index structures. This helps utilize overlaps in records resulting in less rework.
3. Our algorithms build the required prefix trees and inverted indices “on-the-fly”. This makes them usable for relations without indices and when joining intermediate results during join queries with more than two relations.
4. Computing indices on-the-fly instead of precomputing them means that there is no maintenance cost in terms of disk-space or time for updates.
5. Our algorithms can beneficially utilize precomputed indices if they are available. We note that our algorithms out-perform previous approaches even when our algorithms build the indices on-the-fly.
6. Join results are output continuously during the execution of the algorithm and not just at the end. In experiments on real life datasets where the join results are *huge* the (initial) response was almost instantaneous.
7. The output of our algorithms is organized without special effort in the following fashion: the output corresponding to records (in one of the relations) that have the same set contents are clumped together. Further, the output corresponding to “prefixes” of a record appear just before the output corresponding to that record. This organization of output helps in making sense of large join results.

We assume a nested representation of the data [11]. In this representation all the set elements are stored at the same place, facilitating efficient joins on them.

*Organization* In Section 2, we formally define the set join problem. Next, in Section 3, we motivate the use of prefix tree and inverted index structures for the computation of set joins and their on-the-fly construction. In Section 4, we present the proposed set join algorithms. Related work is described in Section 5. The performance of the set join algorithms is evaluated in Section 6. Finally, in Section 7, we summarize the conclusions of our study.

## 2 Problem Definition

**Definition 1 (set join).** *The set join of two relations  $R$  and  $S$  is defined as  $R \bowtie_{p\theta q} S$  where  $p$  and  $q$  are set-valued attributes, and  $\theta$  is a join condition. A pair of records  $t_R \in R$  and  $t_S \in S$  will be present in the join result, if  $\theta$  is satisfied for  $t_R.p$  and  $t_S.q$ .*

For pedagogical reasons, we assume that relations  $R$  and  $S$  contain only two attributes: (1) the set-valued attribute that they are joined on, and (2) the record identifier (*rid*). Hence, each record  $t$  contains a set, which we refer to as  $t.set$ . For convenience, we use  $t_R$  to represent  $t_R.p$  and  $t_S$  to represent  $t_S.q$ .

In this paper we study set *containment*, *equality* and *overlap* joins. The set containment join retrieves all pairs of records  $(t_R, t_S)$ , for which  $t_R \subseteq t_S$ . In equality join,  $(t_R, t_S)$  is in the result iff  $t_R$  and  $t_S$  are exactly the same. Set overlap join retrieves all pairs  $(t_R, t_S)$ , for which  $t_R$  and  $t_S$  has at least one common element. We also handle the more general overlap join called as  $\epsilon$ -overlap, where  $t_R$  and  $t_S$  should have at least  $\epsilon$  common elements.

### 3 Index Structures for Set Joins

In this section, we motivate the use of prefix trees [4] and inverted indices [5] for the computation of set joins. We also discuss how they can be computed efficiently on-the-fly, instead of precomputing them. As mentioned in the Introduction, this is useful because no disk-space is reserved for the index, and database updates require no extra work in keeping the index up-to-date.

#### 3.1 Prefix Trees

Prefix trees have been used in [4] to store sets for the purpose of mining frequent itemsets. This has resulted in several elegant algorithms [4, 3] for that task. Prefix trees can similarly be used to store sets for set-join applications. Since prefix trees store (ordered) sequences and sets are unordered, an ordering is imposed upon the set-elements based on their frequency of occurrence (in a given relation).

Each node  $n$  in the tree (except root) holds a set-element (referred to as  $n.element$ ). The node  $n$  also represents a set (referred to as  $n.set$ ) – containing the elements stored in  $n$  and its ancestors. We also store in  $n$ , an *rid-list* containing the rids (record identifiers) of all records whose content is the same as  $n.set$ . We refer to this list as  $n.ridlist$ . The structure is *compact* because a common prefix of several sets is represented only once. Ordering the set-elements based on their frequency helps in identifying more and longer common prefixes.

In addition to saving space, a prefix tree also saves on time because tasks that need to be performed over all the sets can be performed just once for each common prefix. Thus *prefix trees help avoid redundant work*. Our algorithm to construct prefix trees (adapted from [4]) is as follows:

First the root node of the prefix tree is created. Each record  $t$  of the relation is then inserted into the tree as follows: The elements of  $t$  are first sorted in decreasing order of their frequency in the relation. Starting from the root of the tree, we follow a path  $P$  as long as the sequence of elements in the nodes of  $P$  is a *prefix* of the sorted record  $t$ . We finally reach a node  $n$  such that  $n.set$  is the longest prefix of  $t$  currently represented in the tree. Then we add a path  $P'$  of nodes, as descendants of  $n$ , to hold the remaining elements of  $t$ . The last node in the entire path  $P + P'$  from the root now represents the newly inserted record  $t$ . The rid of  $t$  is appended to the *ridlist* of this node.

The above algorithm is sufficient if the entire prefix tree fits in main memory. Otherwise, we logically divide the database into *horizontal* partitions such that the prefix tree built on that partition fits in main memory. This reduces the efficiency of the above approach because common prefixes between records *across partitions* are not used.

#### 3.2 Inverted Indices

Inverted indices were first used in [6] for the computation of set joins. A recurring task in set join computation is to find all the records that contain a given set. It is possible to do this efficiently using an inverted index, in which for each element in the domain  $D$  a list of record-identifiers (rids) of records having that element is maintained. Thus, the

records containing a given set can be found by just intersecting the lists corresponding to each element in the set.

The inverted index can be constructed by making a single pass over the data. We build a list for each set-element  $x$  containing the rids of records containing  $x$ . We refer to this list as the *inverted list* of  $x$  and represent it by  $x_{list}$ . Using the inverted index, the list of records that contain a given set  $X$  (referred to as the inverted list of  $X$ , or  $X_{list}$ ) can be computed by simply *intersecting* the inverted lists of each element in  $X$ .

As noted earlier, if the entire index does not fit in main memory, the database is logically divided into *horizontal* partitions such that the index constructed over that partition fits in main memory. Note that this partitioning of the database differs from the *vertical* partitioning in [7]. There, the set-elements were divided into disjoint partitions, whereas here, the records are divided into disjoint partitions.

*Precomputing Indices* Building the above index structures (both prefix trees and inverted indices) on-the-fly has a cost of  $\Theta(N)$ , where  $N$  is the total number of elements in all records of the relation. The constant factor involved is reasonably small. For prefix trees, the constant factor would include  $\log(k)$  – the additional cost for sorting elements of each record of length  $k$ . A similar cost (with a still smaller constant factor) would be incurred for simply reading a precomputed index from disk. We feel that in most cases, the additional cost for building indices on-the-fly is justified by the index maintenance costs in terms of disk space and update time. However, we note that *if* a pre-computed index is available, our algorithms described in the next section can make use of it directly instead of building it on-the-fly.

## 4 The PRETTI Algorithms

In this section we present the PRETTI (PREfix Tree based seT JoIn) suite of algorithms for set containment, overlap and equality joins between two relations  $R$  and  $S$ . These algorithms use a prefix tree on  $R$  and an inverted index on  $S$ , unless otherwise specified. For equality join we give an additional algorithm using prefix trees on both  $R$  and  $S$ .

### Algorithm PRETTI: Nested Loop( $R, S$ )

```

1  for each partition  $P_R$  in  $R$ 
2       $R_{PT} = \text{build\_prefixTree}(P_R)$ 
3      for each partition  $P_S$  in  $S$ 
4           $S_{IL} = \text{build\_invertedList}(P_S)$ 
5          Set Join( $R_{PT}, S_{IL}$ )

```

**Fig. 1. Partition Nested Loop Set Join**

If the main memory is insufficient, a *nested loop join* is used, as shown in Figure 1. In the nested loop join, each relation is partitioned *horizontally* so that indices constructed on each pair of partitions  $(P_R, P_S)$ ,  $P_R \in R$  and  $P_S \in S$ , fit in main memory. The required join algorithm is performed on each such pair. Therefore, without loss of generality, we assume that the relations to be joined can be processed in main memory.

## 4.1 Set Containment Join

A set containment join, represented by  $R \bowtie_{p \subseteq q} S$ , is one of the important operators among set joins. It was shown in [7] that set containment joins using inverted indices on  $S$  are more efficient than signature-based and partition-based approaches. In our approach, we retain this idea of using an inverted index on  $S$  (represented as  $S_{IL}$ ) and in addition use a prefix tree on  $R$  (represented as  $R_{PT}$ ).

We need to find pairs of records  $(t_R, t_S)$  from  $R$  and  $S$  such that  $t_R \subseteq t_S$ . That is, for each node  $n$  of  $R_{PT}$ , we need to find  $n.set_{list}$  – the records in  $S$  that contain  $n.set$ . To do this we need to intersect the inverted lists (from  $S_{IL}$ ) of all elements in  $n.set$ . Now, it is clear that  $n.set = \{n.element\} \cup m.set$  where  $m$  is the parent of  $n$ . Therefore,  $n.set_{list} = n.element_{list} \cap m.set_{list}$ . It follows that for each node  $n$ , we can compute  $n.set_{list}$  by processing the nodes of  $R_{PT}$  in a depth-first traversal, since in such a traversal the parent  $m$  of  $n$  is visited before  $n$ .

The pseudo-code of the above algorithm is shown in Figure 2. The function **Set Containment** is a recursive implementation of depth-first traversal over  $R_{PT}$ . Initially it is called separately for each child  $n$  of the root of  $R_{PT}$  with the following arguments – (1)  $n$  itself, and (2)  $n.set_{list}$  (initially equal to  $n.element_{list}$  from  $S_{IL}$ ). It first outputs pairs of rids of records  $(rid_R, rid_S)$  such that  $rid_R \in n.ridlist$  and  $rid_S \in n.set_{list}$  (lines 1–3 of Figure 2). Then, for each child  $c$  of  $n$ , it computes  $c.set_{list}$  by intersecting  $n.set_{list}$  with  $c.element_{list}$ , which is obtained from  $S_{IL}$  (lines 5–7).

**Algorithm PRETTI: Set Containment**( $n, n.set_{list}$ )

```

1  for each  $rid_R$  in  $n.ridlist$  do
2    for each  $rid_S$  in  $n.set_{list}$  do
3      output(  $rid_R, rid_S$  )
4
5  for each child  $c$  of  $n$  do
6     $c.set_{list} = n.set_{list} \cap c.element_{list}$ 
7    Set Containment( $c, c.set_{list}$ )

```

**Fig. 2. Set Containment Join**

## 4.2 Set Overlap Join

The set overlap join retrieves all pairs of records  $(t_R, t_S)$  from relations  $R$  and  $S$ , for which  $|t_R \cap t_S| \geq 1$ . A more general form is  $\epsilon$ -overlap where  $|t_R \cap t_S| \geq \epsilon$ , where  $\epsilon$  is a user-specified parameter. We again use a prefix tree on  $R$  and an inverted index on  $S$ . However, for pedagogical reasons, we first explain the algorithm without using a prefix tree on  $R$ .

**Without Prefix Trees** For each record  $t_R$  in  $R$ , we need to determine all the records  $t_S$  in  $S$  such that  $t_R$  and  $t_S$  share  $\epsilon$  elements. To do this, we build an array  $Count_S$  that holds for each record in  $S$ , the number of elements it contains in common with  $t_R$ .

This array can be built by simply scanning each rid in the inverted lists (in  $S_{IL}$ ) of all elements in  $t_R$  and incrementing its counter in  $Count_S$ . If and when the count of an rid in  $Count_S$  reaches  $\epsilon$ , that rid along with the rid of  $t_R$  is output because then they share  $\epsilon$  elements in common.

The above approach does a lot of redundant work – if two records are identical, the above operation can be performed just once for both records. Even if the two records are not exactly equal, but share several elements in common, much of the work can be reused. Using a prefix tree on  $R$  can help identify such common elements between transactions and avoid redundant work. We now describe this approach.

**With Prefix Trees** An immediate benefit that is obtained by using a prefix tree is that the above operation of building the  $Count_S$  array, etc. needs to be done *only once* for all records in the *ridlist* of each node. In addition, since the set corresponding to a node  $n$  shares all the elements of its parent node  $m$ , we can reuse the  $Count_S$  array of  $m$  for processing  $n$  – this  $Count_S$  array is up-to-date with respect to the inverted lists of all elements in  $m$ . We only need to update the  $Count_S$  array by scanning each rid in  $n.element_{list}$  and incrementing its counter. We note that partial results are output as soon as the value of a counter reaches  $\epsilon$ , instead of waiting till  $Count_S$  is updated completely for node  $n$ .

In the above procedure, we observe that if the  $Count_S$  array for node  $m$  has an entry whose count equals or exceeds  $\epsilon$ , then it would automatically equal or exceed  $\epsilon$  even for  $n$ . Therefore, we maintain the rids corresponding to such entries of  $Count_S$  separately in an array called  $Cur_{sol}$ . Then, while processing node  $n$ , we output the pairs of rids in  $Cur_{sol}$  and  $n.ridlist$ , without further processing.

The above paragraphs describe how the  $Count_S$  array of a parent node can be reused at a given node. In a depth-first traversal of the prefix tree, a node  $n$  and all its children are visited before the next sibling of  $n$  is visited. In order to ensure that the  $Count_S$  array is usable by the next sibling of  $n$ , we need to *undo* changes made to it while processing  $n$  (and its children). This is achieved by simply scanning each rid in  $n.element_{list}$  and *decrementing* its counter, after the processing over  $n$  has been completed. Finally, if this decrement operation causes an entry in  $Count_S$  to fall below  $\epsilon$ , the corresponding rid needs to be removed from  $Cur_{sol}$ .

The pseudo-code of the algorithm described above is shown in Figure 3. Like the set containment join (Figure 2), the overall structure of this algorithm is a recursive implementation of a depth-first traversal over  $R_{PT}$ . It is initially called with the root of  $R_{PT}$  as the first argument. The  $Count_S$  array described above is the second argument and is initialized to zeros before calling the function for the first time. The third argument to the function is  $Cur_{sol}$ , which as described above, is an array of rids corresponding to the entries of  $Count_S$  whose values equal or exceed  $\epsilon$ . It is initially empty.

### 4.3 Set Equality Join

The set equality join retrieves all pairs of records  $(t_R, t_S)$  from relations  $R$  and  $S$ , for which  $t_R == t_S$ . We present two different algorithms for set equality join. The first algorithm is a variant of the set containment join presented in Section 4.1 and therefore

**Algorithm** Set Overlap ( $n, Count_S, Cur_{sol}$ )

```

1  for each child  $c$  of  $n$  do
2    for each  $rid_S$  in  $c.element_{list}$ 
3       $Count_S[rid_S]++$  // increment
4      if  $Count_S[rid_S] == \epsilon$ 
5        append  $rid_S$  to  $Cur_{sol}$ 
6        for each  $rid_R$  in  $c.ridlist$  do
7          output( $rid_R, rid_S$ )
8
9    Set Overlap ( $c, Count_S, Cur_{sol}$ )
10
11   for each  $rid_S$  in  $c.element_{list}$ 
12      $Count_S[rid_S]--$  // decrement
13     if  $Count_S[rid_S] == \epsilon - 1$ 
14       delete  $rid_S$  from  $Cur_{sol}$ 

```

**Fig. 3. Set Overlap Join**

uses a prefix tree on relation  $R$  and an inverted index on  $S$ . The second algorithm uses prefix trees on both relations and no inverted index.

**Using a Prefix Tree and an Inverted Index** The set equality join can be considered to be a special case of the set containment join – we first find all pairs  $(t_R, t_S)$  from  $R$  and  $S$  such that  $t_R \subseteq t_S$  and among these pairs, we output only those for which  $|t_R| == |t_S|$ . We can obtain  $|t_R|$  from the depth of the node corresponding to  $t_R$  in the prefix-tree. The value of  $|t_S|$  can be precomputed and stored while building the inverted index on  $S$  – this results in a small memory overhead.

**Using Two Prefix Trees** An equality join can be computed efficiently when both relations  $R$  and  $S$  are *sorted* on the join attribute. We achieve this by constructing prefix trees  $R_{PT}$  and  $S_{PT}$  on  $R$  and  $S$ , respectively. A depth-first traversal on  $R_{PT}$  or  $S_{PT}$  yields the sets stored in it in a sorted lexicographic order, as explained below.

As mentioned in Section 3.1, the individual set elements are ordered based on their frequency in each relation. Here, we order them based on their *total* frequency in both relations. We then define the lexicographic ordering of sets with respect to this frequency-based ordering of individual set elements. Note that prefix tree construction is more efficient than generic sorting since it requires only  $\Theta(N)$  time (see Section 3).

The equality join algorithm then merely consists of a simultaneous depth-first traversal over both  $R_{PT}$  and  $S_{PT}$ . Let the current nodes during the traversals be  $n_R$  and  $n_S$  in  $R_{PT}$  and  $S_{PT}$ , respectively. If  $n_R.set < n_S.set$ , then the traversal over  $S_{PT}$  is *suspended* until  $n_R.set == n_S.set$ . Similarly, if  $n_R.set > n_S.set$ , then the traversal over  $R_{PT}$  is *suspended*. As long as  $n_R.set == n_S.set$ , the pairs of rids in the ridlists of  $n_R$  and  $n_S$  are output.

Note that for each node  $n$  in a prefix tree, we do *not* store  $n.set$  in the node. Instead, this is computed on the fly during the depth-first traversal by forming the union of the *element* fields stored at  $n$  and its ancestors.

## 5 Related Work

Set join operators received significant attention recently. In [2], the authors showed that set-joins are one of the hardest operators to optimize. Several nested loop join techniques were evaluated in [5] and *signature-hash join* was found to be the best among them. A recent work [13] studied more complex varieties of similarity joins on set-valued data. Applicability of set division operator for containment join on set-data in first normal form is discussed in [12]. The Apriori algorithm [1] for mining frequent itemsets has been suggested for containment joins since it counts the occurrences of “candidate itemsets” in set-data.

Several partition based approaches for set joins have been proposed such as PSJ [11], APSJ, DCJ and ADCJ [9, 10]. In these approaches, the relations are partitioned based on hash functions such that pairs of records in the output fall in the same partition. Although faster than signature based methods, their performance heavily depends on the number of partitions and the hash function used. A bad partitioning can make these approaches perform near the worst case quadratic time complexity due to false drops. Though adaptive approaches [10] have been proposed to overcome the first drawback, the problem of false drops still remains. Also, most adaptive approaches perform better than PSJ only in cases of very large average set cardinality [9].

Block Nested Loop Join (BNL) was proposed in [7]. It first constructs an inverted index  $S_{IL}$  over the relation  $S$ . Then, for all elements in each record  $t_R \in R$ , the corresponding inverted lists are intersected to get the records in  $S$  that contain  $t_R$ . Since the complete relation  $S$  may not fit in main memory  $S_{IL}$  is *vertically* partitioned into a number of blocks such that each block fits in main memory. Each partition has inverted lists of a subset of the total elements in the domain.

In BNL, instead of loading  $R$  record by record, a page of records is read. For each page of  $R$  all blocks of  $S_{IL}$  are loaded one by one and processed. Since all elements in a record of  $R$  need not belong to a single block of  $S$ , *temporary files* are used to store the partial results for each block.

The major drawbacks of BNL (w.r.t. PRETTI) are: (1) In BNL, overlaps between records are not taken into account. (2) Due to the vertical partitioning approach, BNL needs to maintain temporary files. The sizes of these files can be of the order of output size, which can be quadratic over the size of the relations. (3) To build a complete vertical partition, the entire database needs to be scanned. This excludes the possibility of constructing a vertical partition on-the-fly.

## 6 Experiments

In this section we compare our proposed algorithms with BNL [7] and partition based PSJ [11] and APSJ [10] algorithms. We mainly compare our approach with BNL since it was shown [7] to outperform partition based PSJ. In this section we always perform

self-joins, i.e.  $R = S$ . We also assume that  $R$  is the outer relation and  $S$  is the inner relation. All the experiments are performed on a 2.6 GHz Celeron PC with 256 MB main memory, running Red Hat Linux 2.4.20-8. An illusion of limited main memory is created by limiting the buffer size and also ensuring that Linux does not cache  $S$  during nested loop joins – we made several copies of  $S$  and used a different one for each iteration of the nested loop join.

For comparison with APSJ, we used the Set Containment Join Testbed [8]. We implemented the BNL suite of algorithms as described in [7] in which we incorporated the functionalities such as compression, pipelining and pruning using set cardinalities. We study these algorithms for varying buffer and relation sizes. In experiments where we do not vary buffer sizes, we fix it to 25% of the corresponding relation size.

For the datasets used in our experiments, the output of set joins are *huge*. Writing this to disk would over-shadow the actual join processing cost. To avoid this, we only count the number of pairs in the solution, instead of writing them to screen or disk.

BMS Dataset	Dom. Size	Avg. Set Card.	Relation Card.	Max. Set Card.	Set Card. $\geq 10$	Set Card. $\geq 20$	Set Card. $\geq 50$	Set Card. $\geq 100$	Rel. Size
WebView1	497	2.5	59,602	267	2098	411	72	34	588KB
WebView2	3340	5	77,512	161	10971	2574	160	8	1.5MB
POS	1657	6.5	515,597	164	128098	29934	955	37	11MB

Fig. 4. Dataset Characteristics

For our experiments, we used the real life datasets BMS-POS, BMS-WebView1 and BMS-WebView2 from Blue Martini Software [15]. These datasets originated from a dot-com company called Gazelle.com, a leg-wear and leg-care retailer and contains several months of click-stream data. Figures 6 shows the characteristics of these datasets.

The major criteria [11] to test join algorithms are their scalability with increasing relation cardinality, domain cardinality and record length. BMS WebView1 has a small domain cardinality but some records are very long. BMS WebView2 has a large domain cardinality of 3340. Both the domain size and relation cardinality are large in BMS POS.

## 6.1 Set Containment Join

In this section we compare the performance of PRETTI with BNL, PSJ and APSJ for set containment joins. The first experiment, Figure 5a, tests the scalability of the algorithms w.r.t. relation cardinality. Note that the y-axis is shown in *log-scale*. The dataset  $R$  is constructed by taking random samples from BMS-POS of increasing cardinalities. The buffer size was set to 25% of the size of  $S$ . We see that the response time of PRETTI increases very slowly for larger relations. This can be attributed to the overlaps between new records and old records in  $R$ .

On the other hand, we see that the performance of BNL deteriorates significantly as the relation cardinality increases. The major reason for this is its inability to exploit overlaps between records in  $R$ . Another reason is that as the relation cardinality increases, inverted lists become longer, resulting in fewer lists loaded into memory each time, which in turn results in large temporary files.

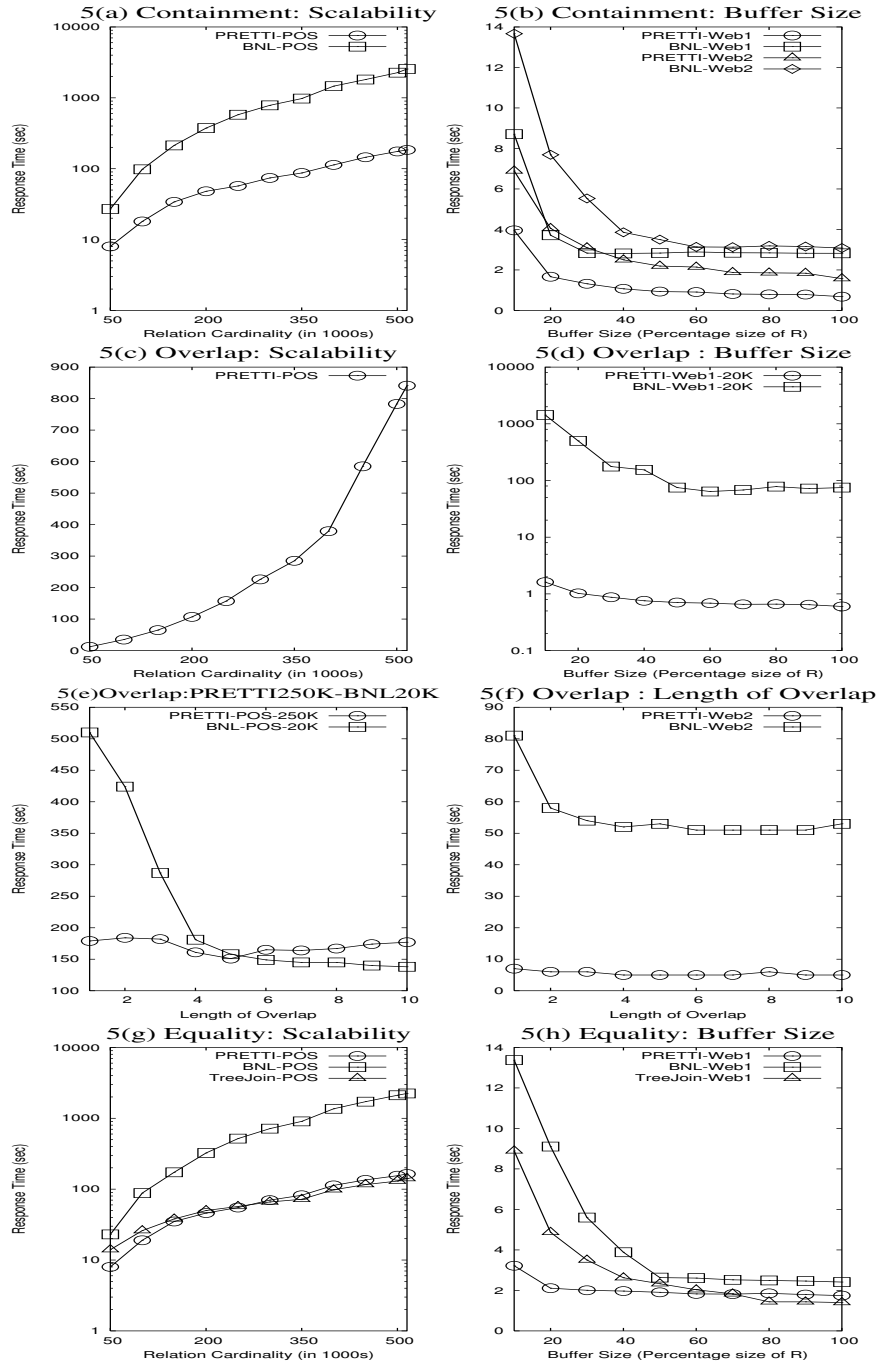


Fig. 5. Experimental Results

Figure 5b shows the running times of PRETTI and BNL for increasing buffer sizes on WebView1 and WebView2 datasets. We see that PRETTI consistently outperforms BNL. The response times for WebView2 show that PRETTI is well suited for datasets with large domains. We also see that as the buffer size decreases, BNL performs much worse since it needs to rely more on large temporary files.

Set Card.	10	20	40	60	80	100
PRETTI	10s	20s	39s	66s	84s	123s
APSJ	977s	812s	276s	158s	156s	172s
PSJ	479s	714s	1139s	1095s	-	-

**Table 1. Containment Join (Set Card. Vs Resp. Time)**

Table 1 compares PRETTI, PSJ and APSJ on a 100K record dataset generated by the testbed used in [8]. We see that PRETTI outperforms PSJ and APSJ significantly.

## 6.2 Set Overlap Join

Set overlap join is the most time-consuming operation among the three join types studied in this paper. Figure 5c shows the performance of PRETTI on the POS dataset. We see that PRETTI can handle large datasets even for overlaps. Due to the very large temporary files, BNL could not be run on this dataset.

Figure 5d (y-axis in *log-scale*) shows the running times of the algorithms for increasing buffer sizes on the WebView1 dataset. Figure 5e shows the running time of PRETTI for increasing values of  $\epsilon$  for a relation of 250K records. We see that PRETTI is scalable for high values of  $\epsilon$ . The difference in the response time is due to the varied cost in maintaining  $C_{ur_{sol}}$  for different  $\epsilon$ . To compare PRETTI and BNL, we show the response time for BNL on 20K records. Figure 5f shows the performance of the algorithms for varying overlap sizes on the WebView2 dataset.

In these graphs, PRETTI clearly outperforms BNL. For each record of  $R$ , BNL computes the union of inverted lists of all its elements. Since long records are common in real datasets, the resulting list explodes and can reach the worst case size (the relation cardinality). Further, the sizes of temporary files needed to eliminate duplicates (and hence the time to process them) can be quadratic on the relation cardinality.

## 6.3 Set Equality Join

For equality join, we compare the two PRETTI algorithms in Section 4.3 with BNL. We refer to the PRETTI algorithm that uses two prefix trees as “tree-join”. Figure 5g (y-axis in *log-scale*) shows the algorithms’ scalability with increasing relation cardinality.

As expected, we see that the response time of PRETTI and BNL is similar to their set containment join counter-parts. Surprisingly, we find that PRETTI outperforms tree-join in most cases. This is due to the maintenance of two prefix trees in main memory. Each node occupies three times space compared to that of a single element. This results in more partitions on  $R$  and  $S$ , which increases the number of iterations in the join. This experiment also shows that PRETTI outperforms BNL by a large margin.

Figure 5h shows response times of these algorithms on WebView1 for increasing buffer sizes. The rapid increase in response time of BNL as the buffer size decreases can be attributed to large temporary files.

## 7 Conclusions

In this paper we proposed the PRETTI suite of algorithms for set containment, overlap and equality joins. We investigated the use of prefix trees and inverted indices for performing set joins efficiently. Our algorithms do not require these structures to be stored on the disk, but instead build them on the fly. This property makes them useful in computing joins of intermediate results (which have no indices) in large join queries. Our results show that our algorithms significantly outperform previous approaches.

## References

1. R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of Intl. Conf. on Very Large Databases (VLDB)*, September 1994.
2. J. Cai, V.T. Chakaravarthy, R. Kaushik, and J.F. Naughton. On the complexity of join predicates. In *ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, 2001.
3. G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI)*, 2003.
4. J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, 2000.
5. S. Helmer and G. Moerkotte. Evaluation of main memory join algorithms for joins with set comparison join predicates. In *Proc. of Intl. Conf. on Very Large Databases (VLDB)*, 1997.
6. S. Helmer and G. Moerkotte. A study of four index structures for set-valued attributes of low cardinality. Technical report, University of Mannheim, 1999.
7. N. Mamoulis. Efficient processing of joins on set-valued attributes. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, 2003.
8. S. Melnik. Set containment joins: Testbed. <http://www-db.stanford.edu/~melnik/scj>.
9. S. Melnik and H. Garcia-Molina. Divide-and-conquer algorithm for computing set containment joins. In *Intl. Conf. on Extending Database Technology*, 2002.
10. S. Melnik and H. Garcia-Molina. Adaptive algorithms for set containment joins. *ACM Transactions on Database Systems (TODS)*, 28(2), 2003.
11. K. Ramasamy, J.M. Patel, J.F. Naughton, and R. Kaushik. Set containment joins: the good, the bad and the ugly. In *Proc. of Intl. Conf. on Very Large Databases (VLDB)*, 2000.
12. R. Rantau. Processing frequent itemset discovery queries by division and set containment join operators. In *8th ACM SIGMOD DMKD Workshop*, 2003.
13. S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, 2004.
14. M. Stonebraker. *Object-relational DBMS: The Next Great Wave*. Morgan Kaufmann, 1996.
15. Z. Zheng, R. Kohavi, and L. Mason. Real world performance of association rule algorithms. In *Intl. Conf. on Knowledge Discovery and Data Mining (KDD)*, 2001.