Set Containment Joins: The Good, The Bad and The Ugly

Karthikeyan Ramasamy[*] *UW-Madison* *karthik@cs.wisc.edu*

Jignesh M Patel[†] *UM-Ann Arbor* *jignesh@eecs.umich.edu*

Jeffrey F Naughton *UW-Madison* *naughton@cs.wisc.edu*

Raghav Kaushik *UW-Madison* *raghav@cs.wisc.edu*

April 4, 2000

Abstract

Efficient support for set-valued attributes is likely to grow in importance as object-relational database systems, which either support set-valued attributes or propose to do so soon, begin to replace their purely relational predecessors. One of the most interesting and challenging operations on set-valued attributes is the set containment join, because it provides a concise and elegant way to express otherwise complex queries. Unfortunately, evaluating these joins is difficult, and naive approaches lead to algorithms that are very expensive. In this paper, we develop a new partition based algorithm for set containment joins: the Partitioning Set Join Algorithm (PSJ), which uses a replicating multi-level partitioning scheme based on a combination of set elements and signatures. We present a detailed performance study with a complete implementation in the Paradise object-relational database system. Our results show that PSJ outperforms previously proposed set join algorithms over a wide range of data sets.

1 Introduction

The data modeling community has long realized that set valued attributes provide a concise and natural way of modeling complex data [RKS98]. Recently, there has been a resurgence of interest in set-valued attributes from two different perspectives. First, commercial O/R DBMS [Sto96] are beginning to support set-valued attributes, which is likely to lead to their use in "real" applications. Second, the rise of XML as an important data standard increases the need for set-valued attributes, since it appears that set-valued attributes are key for the natural representation of XML data in

[*] Portions of this research were done while the author was at NCR Corporation, Madison, WI

[†] Portions of this research were done while the author was at NCR Corporation, Madison, WI

relational systems [SHT+99]. Unfortunately, although sets have been fairly well studied from a data-modeling viewpoint [Zan83], very little has been published about the efficient implementation of operations on set-valued attributes. In this paper, we consider the implementation of a particularly challenging operation over set-valued attributes, the set-containment join.

Many real world queries can be easily expressed using set containment joins. For example, consider two relations:

STUDENT (sid, {courses})
COURSES (cid, pre-requests)

where the attributes {courses} and {pre-requests} are set-valued attributes. A frequently asked query on this data set may be to find the courses that the students are eligible to take. Such a query can be easily expressed as a set containment join using the following query:

```
SELECT s.sid, c.cid
FROM STUDENT s, COURSES c
WHERE c.pre-requests ⊆ s.courses
```

Another motivating example arises from the web. Consider a simple relation that describes a document and set of hyper-links that point to it.

DOCUMENT(did, {hyper-links-in}, actual-document)

As an another motivating example, consider a simple relation that describes a document and set of hyper-links that point to it.

DOCUMENT(did, {hyper-links-in}, actual-document)

Suppose document d_1 is more important than d_2 if d_1 is linked-to by a superset of the documents that link to d_2. We can find pairs of documents d_1 and d_2 where d_1 is more important than d_2 with the following query:

```
SELECT d1.did, d2.did
FROM DOCUMENT d1, DOCUMENT d2
WHERE d2.hyper-links-in ⊂ d1.hyper-links-in
```

The algorithms available for implementing set-containment joins depend upon how set-valued attributes are stored in the database. As described in [RNM99], sets can be stored in the *nested internal representation* (set elements are stored together along with the rest of the attributes) or the *unnested external representation* (set elements are scattered and stored in a separate relation). To the best of our knowledge, current commercial O/R DBMS use the unnested external representation. Since the unnested external representation reduces to standard SQL2 relations under the covers, set containment joins on the unnested external representation can be evaluated by rewriting the queries into SQL2 (with no sets) and evaluating these rewritten queries. On the other hand, with the nested internal representation, the most obvious algorithm for evaluating set-containment joins is nested loops. Two questions immediately arise: (1) Are there better algorithms than nested loops? (2) How do these algorithms compare in efficiency with the rewrite in SQL2 approach that is most logical for the unnested external representation?

This paper attempts to answer these questions by proposing a new partition-based join algorithm for set containment joins, which we call PSJ. Partition-based algorithms certainly dominate join algorithms in scalar and spatial domains, so it is natural to suspect that a partition-based algorithm will be the algorithm of choice for set-containment joins.

This paper makes two main contributions. First, it presents the new algorithm PSJ for set containment joins. Second, it includes an extensive performance study of three set containment algorithms: the traditional SQL approach on the unnested external representation, and signature nested loops and PSJ on the nested internal representation. Our experience with an implementation in the Paradise object-relational database system [PYK+97] shows that PSJ yields significant speedup over both the SQL-based approach and signature nested loops. An added benefit of this algorithm is that, like all partition-based algorithms, it is trivially parallelizable. Finally, our results present a strong case for storing sets in the nested internal form, since PSJ and even signature nested loops outperform the rewritten queries over the unnested external representation.

1.1 Related Work

Joins have been studied extensively in relational [MK76], [Bra84], [DKO+84], [DNS91] and spatial domains [LR96], [PD96]. Pointer joins for efficiently traversing path expressions in object-oriented databases has also been studied extensively [DLM93], [SC90]. However, there is very little previous work on set containment joins. The only reported work of which we are aware is the work by Helmer and Moerkotte [HM96], [HM97]. These papers investigate nested loops algorithms for computing a set containment join and propose a new signature based hash join. We discuss these algorithms in Sections 3 and 4.4 Signatures have been studied in detail in [FC84] and [ZMR98] under the context of fast retrival of documents matching a query predicate. [IKO93] studies the application of signatures for evaluating conjuntive and disjunctive predicates over set-valued attribute.

1.2 Paper Organization

The rest of the paper is organized as follows. Section 2 defines the problem of set containment and the notation used in the paper. Various storage representations for sets, the SQL approach and signature nested loops joins are explained in detail in Section 3. The need for partition based algorithms is justified in Section 4. The partition based set join algorithm is outlined in Section 4.2. Section 5 presents a detailed performance study of all the algorithms. The conclusions and future work are presented in Section 6.

2 Problem Definition and Notations

For the rest of the paper, we consider the two relations $R(a, \{b\})$ and $S(c, \{d\})$ containing the set valued attributes $\{b\}$ and $\{d\}$ respectively. Since set is a type constructor, attributes b and d can be of an arbitrary type and we assume that these types provide an equality predicate that compares the equivalence of two set elements. Also we do not assume any order among the set elements. The

set containment join $R \bowtie_{\{b\} \subseteq \{d\}} S$ pairs tuples in relation R and S such that $\{b\}$ is subset of $\{d\}$. Table 1 describes the notation used in the rest of the paper.

$\| R \|$	Number of pages of R	$\| S \|$	Number of pages of S
$\mid R \mid$	Relation cardinality of R (# of tuples)	$\mid S \mid$	Relation cardinality of S (# of tuples)
k_R	Average set cardinality of R	k_S	Average set cardinality of S
r_R	Replication factor of R	r_S	Replication factor of S
σ	Selectivity of $R \bowtie_{\{b\} \subseteq \{d\}} S$	f	False drops as a percent of $\sigma \mid R \| S \mid$
TID_s	Size of an rid (bytes)	P_S	Size of the data page (bytes)
h_c	CPU cost of hash computation	s_c	CPU cost of comparing signatures
IO_{seq}	Cost of a sequential I/O	IO_{rand}	Cost of a random I/O

Table 1: Description of Notation Used

3 Previously Proposed Algorithms

Options for algorithms for set containment joins heavily depend on how the set valued attributes are stored in the database. In order to make this paper self-contained, we briefly discuss the options for storing set-valued attributes.

3.1 Storage Representations for Sets

Various representations for sets are possible depending on the following two characteristics: **nesting** (set elements are clustered or scattered) and **location** (set elements are either stored with the rest of the attributes internally or vertically partitioned and stored externally). As outlined in [RNM99], the three main representations for sets are:

- **Nested Internal**: Here the set elements are grouped together and stored with the rest of the attributes in the tuple.
- **Unnested External**: In this representation, the set-valued attribute is stored in a separate relation. For each set-valued attribute in a relation, two relations are created: (1) A base relation that stores the other non set-valued attributes and an identifier, and (2) An auxiliary relation that stores each element of the set-valued attribute as a tuple. Consider relation $R(a, \{b\})$. This relation is decomposed into two relations: $R_B(i, a)$ and $R_S(i, b)$, where R_S contains all the set elements. The elements and the base tuple are related by the key attribute i. During query processing, a join is used to associate the base tuple and set-elements.
- **Nested External**: This is similar to nested internal except that set valued attribute is vertically decomposed and stored in an external relation. Again, consider a relation $R(a, \{b\})$. This relation is decomposed into two relations: $R_B(i, a)$ and $R_S(i, \{b\})$, where R_S contains all the set elements. The elements and the base tuple are related by the key attribute i. The

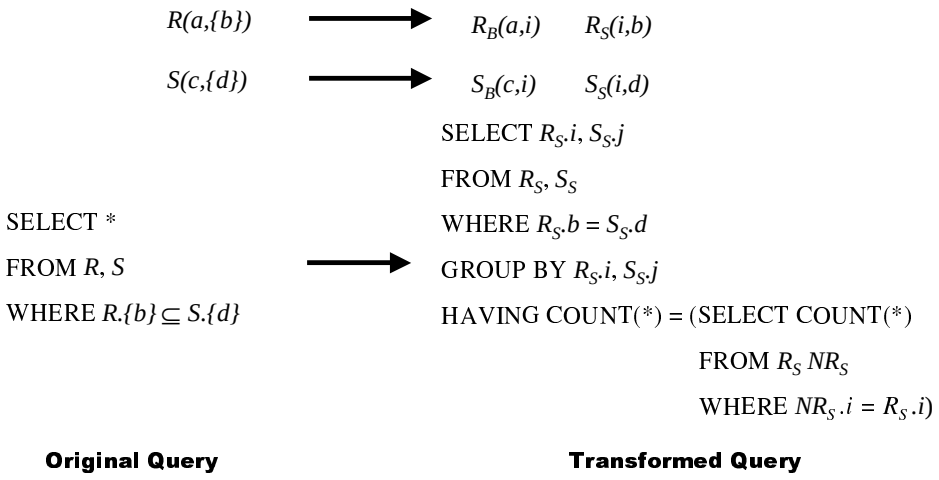

Figure 1: Original and Transformed SQL Queries (excluding final joins for a and c)

number of tuples in R_B and R_S are the same as in the original relation R. As in unnested external, a join is used to associate each base tuple with its set.

As shown in [RNM99], for selections on set-valued attributes the performance of the nested internal and the nested external representations are very similar. We expect the same result to hold for set containment joins, since any algorithm on the nested internal representation can be easily translated into an equivalent algorithm on the nested external representation with just one additional key-key scalar join. Consequently, in this paper, we focus only on set containment algorithms for the unnested external and the nested internal representations.

3.2 Join Algorithms for Unnested External Representation

If sets are stored in the unnested external representation, set-containment joins can be expressed and evaluated using standard SQL2 constructs. This approach is important to study, because (a) it is the simplest to add to any RDBMS, and (b) perhaps because of (a), to our knowledge the commercial O/R DBMSs all use this approach. As we discussed in Section 3.1, in this representation, a relation with a set-valued attribute is decomposed into two relations. A set containment operation can then be expressed using SQL over these decomposed relations. If R and S are the two relations being joined, and R_S and S_S are the corresponding decomposed auxiliary set relations, then the original and transformed queries are shown in Figure 1.

The rewritten query joins the set relations and groups the pair of sets that have at least one element in common. Then, for each group, it checks whether the size of the group (which is the number of elements in common between the R set and the S set) is the same as the cardinality of the set in R. This query, as written, returns in the answer tuples only the pair of set ids that satisfy the containment. If any other attributes were included in the answer, additional joins would be required to extract them from the input relation. The main problem with this approach is the efficiency of evaluation of this query. Since this is a correlated nested query, the nested query must

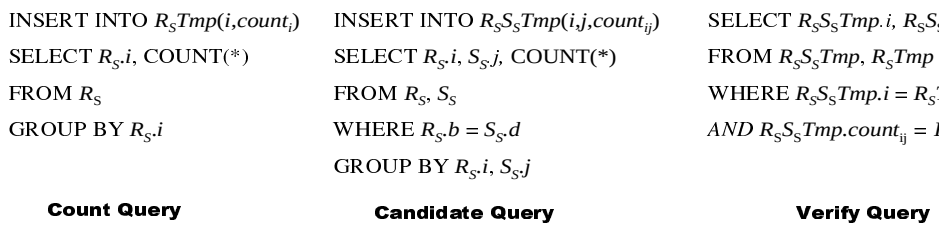

Figure 2: Magic Sets Rewriting

be evaluated for each group. Since the number of groups is proportional to $\mid R \mid \times \mid S \mid$, the cost of evaluating the nested query naively can be prohibitively large. A possible optimization is to use magic-sets rewriting [SPL96] and transform the original query into the set of queries shown in Figure 2, thus evaluating the inner query only once (as opposed to once for every tuple produced by the outer block). Even after this transformation, each set element in R_S is compared with every other set element in S_S. Hence the evaluation of the block (join followed by group-by) is likely to be the dominating cost.

Our experiments show empirically that even this approach performs very poorly unless the set sizes and relation sizes are small; in fact, in many cases, it is so bad that the algorithm can arguably be called "ugly".

3.3 Signature Nested Loops Algorithm for Nested Internal Representation

Nested loop algorithms for set containment fall into two broad categories: n̂aive nested loops and *signature nested loops*. In naive nested loops, the set containment predicate is evaluated on pairs of sets for the entire cross product of R and S. As shown in [HM97], naive nested loops performs poorly, as it is very expensive to compute the set containment predicate for every pair of tuples. Hence we do not consider naive nested loops for the remainder of this paper.

The signature nested loops algorithm proposed by [HM97] attempts to reduce the cost of evaluating the containment predicate by approximating sets using signatures and evaluating the join predicate by comparing these signatures. A signature is a fixed length bit vector that is computed by applying a function M iteratively to every element e in the set and setting the bit determined by $M(e)$. If the containment predicate $s \subseteq t$ is to be satisfied for two signatures s and t, then the following condition is necessary: *For all bit positions that are set to 1 in signature s, the corresponding bits in signature t should be set to 1.* However, this condition is not sufficient since signatures are only an approximate representation for the set (unless the signature length is equal to the size of the domain of the set). Hence using signatures to evaluate a predicate will yield false drops. That is, two sets may have signatures that indicate containment, while the actual sets do not really satisfy the containment predicate. The actual sets must be examined to eliminate these false drops.

We are now ready to describe the signature nested loops algorithm. This algorithm operates in three phases: the *signature construction phase*, the *probing phase*, and the *verification phase*. During the signature construction phase, the entire relation R is scanned, and for every tuple $t_i \in R$,

a signature s_i is constructed. A triplet (c_i, s_i, OID_i) is computed and stored in an intermediate relation R_{sig}; here c_i is the set cardinality and OID_i is the physical record identifier (rid) of the tuple. The same process is repeated for the relation S and an intermediate relation S_{sig} is created. Next, the algorithm proceeds to the probing phase, where the tuples of R_{sig} and S_{sig} are joined. For every pair $(c_i, s_i, OID_i) \in R_{sig}$ and $(c_j, s_j, OID_j) \in S_{sig}$, two conditions must be verified (i) $c_i \leq c_j$ and (ii) $s_i \wedge s_j = s_i$, where the wedge represents the bit-wise and of the two signatures. If both the conditions are satisfied, then the pair (OID_i, OID_j) is a possible candidate for the result. During the final verification phase, the tuples referred to in the candidate (OID_i, OID_j) pairs are fetched and the subset predicate is evaluated on the actual set instances, producing the final result.

The main issue in the signature nested loop join algorithm is reducing the number of false drops to minimize the cost of the verification phase. The false drop probability depends on the number of bits used in constructing the signature. The greater the signature length, the smaller will be the false drop probability. However, larger signatures lead to more bit comparisons per signature, thereby increasing the execution time of the probing phase. Hence, it is necessary that the chosen signature size be such that further increases in the number of bits do not significantly reduce the false drop probability.

3.3.1 Estimation of Signature Size

Estimating the signature size requires the computation of false drop probability. The definition of false drop probability P_{FD} as defined in [IKO93] is given by

$$P_{FD} = \frac{falsedrops}{N - actualdrops} \tag{1}$$

where N is the total number of comparisons and $actualdrops$ is the total number of qualified pair of tuples including the false drops. Equation (1) can be rewritten as

$$P_{FD} = \frac{falsedrops}{N - resultsize - falsedrops} \tag{2}$$

Now $resultsize$ can be expressed as $\sigma \mid R \mid\mid S \mid$ where σ is the join selectivity. If $falsedrops$ are expressed as a percentage of $resultsize$ as $f\sigma \mid R \mid\mid S \mid$, where f is the tolerable false drop percentage. Now P_{FD} can be expressed as

$$P_{FD} = \frac{f\sigma \mid R \mid\mid S \mid}{\mid R \mid\mid S \mid - \sigma \mid R \mid\mid S \mid - f\sigma \mid R \mid\mid S \mid} \tag{3}$$

The false drop probability P_{FD} of a subset predicate between two sets of size k_R and k_S is derived in [IKO93] and is given by

$$P_{FD} = (1 - e^{-k_S/F_{SNL}})^{k_R} \tag{4}$$

Using equations (3) and (4), we can determine the the optimal signature length (F_{SNL}) as

$$F_{SNL} = \frac{-k_S}{ln\left(1 - \left(\frac{f\sigma}{1-\sigma(1+f)}\right)^{1/k_R}\right)} \tag{5}$$

Note that even with the signatures of an ideal length, this algorithm compares signatures for every pair of tuples in the cross product of R and S. If R and S each has one million tuples, there are one trillion comparisons. This is discouraging enough to be considered "bad."

4 Partition Based Algorithms

In this section, we propose a new algorithm for the nested internal representation that is based upon partitioning. In general, partition based algorithms for joins (scalar and spatial) attempt to optimize join execution by partitioning the problem into multiple smaller subproblems using a partitioning function. First, the relation R is partitioned into k partitions, $R_1, R_2, \ldots, R_k$. Similarly, the relation S is partitioned into $S_1, S_2, \ldots, S_k$ using the same function. Note that we are using a generalization of the classical definition of partitioning in that one tuple may be mapped to multiple partitions. An ideal partitioning function has the following characteristics:

- Each tuple r of relation R falls exactly in one of the partitions R_i $(1 \leq i \leq k)$
- Each tuple s of relation S falls exactly in one of the partitions S_i $(1 \leq i \leq k)$
- The join can be accomplished by joining only R_i with S_i $(1 \leq i \leq k)$

It is hard and expensive to satisfy the three conditions in non-scalar domains (spatial and set domains) as evident from the theory of indexability [HKP97].

4.1 Will Partitioning Improve Speedup?

Partitioning algorithms have been shown to outperform other algorithms in scalar and spatial domains. In this section, we argue how partitioning speeds up set containment join. We analyze the speedup using a very simple analytical model. Let us assume that both the relations R and S have N tuples and joins within the partitions are done using signature nested loops. (We relax this assumption later in this section). Also consider the class of partitioning functions in which only tuples of S are replicated. If r_S is the replication factor and P is the number of partitions, then the overall cost of the partition-based algorithm, ignoring constants, is given by

$$\sum_{P} \left(\frac{N}{P}\right) \times \left(\frac{r_S N}{P}\right) \tag{6}$$

On the other hand, the cost of the nested loop algorithm, again ignoring constants, is N^2. Hence the speedup of the partition-based algorithm is proportional to P/r_S, where r_S depends upon P.

To investigate this dependence, we make the assumption that the set instances draw their elements from the domain uniformly. Furthermore, we assume that the partitioning algorithm works by partitioning the elements of the domain from which the set is drawn. If a set has an element e, and e maps to a partition p, then the set itself must be mapped to partition p. From statistics [Fel57], assuming a large domain size (greater than set cardinality), the expected number

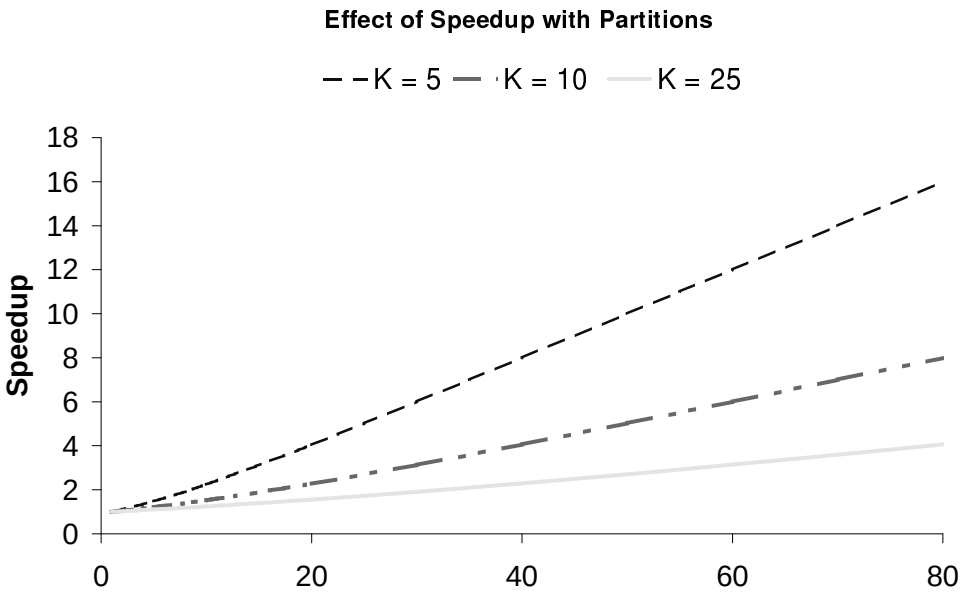

Figure 3: Variation of Speedup with Increase in Partitions

of partitions to which a tuple is replicated (essentially r_S) when its set instance has cardinality of k is then given by

$$k - k(1 - 1/k)^P \tag{7}$$

According to equation (7), when the number of partitions is increased, the expected number of partitions approaches the value of k asymptotically (thereby bounding the replication to k). Hence the speedup of partition-based algorithm can be rewritten as

$$Speedup = \frac{P}{k - k(1 - 1/k)^P} \tag{8}$$

If k_S is the average set cardinality of relation S, the speed up can be easily rewritten as

$$Speedup = \frac{P}{k_S - k_S(1 - 1/k_S)^P} \tag{9}$$

A plot of equation (9) for various values of k_S is shown in Figure 3. The graph shows that as P increases the speedup increases. Consider the individual effects of the two terms in equation (9): P and $k_S - k_S(1 - 1/k_S)^P$. Increasing P tends to increase the speedup. However, increasing P also increases $k_S - k_S(1 - 1/k_S)^P$, which has the effect of decreasing the speedup. Since the rate of increase is greater than the rate of decrease of the denominator, overall the speedup increases. Replication is bounded as a tuple cannot be replicated to more partitions than its set cardinality. nce the replication has reached the maximum of k_S, the rate of increase is purely dominated by increase in P. Also observe that for a given speedup to occur, higher cardinality sets require large number of partitions since the decreasing effect of replication is extended in large sets. In order to counteract this effect, more partitions are required.

The number of partitions is bounded by the domain size. Hence the speed up is bounded by $\mid D \mid /k$. Of course, this analysis is greatly oversimplified and in practice such a speedup is

not attainable, because increasing the number of partitions causes overhead of its own. However, the intuition from this simple model is valid - because there is a bound on the replication factor, increasing the number of partitions beyond a certain level will not cause any more replication.

Finally, we return to the assumption that each pair of partitions is joined using nested loops. Relaxing this assumption and considering faster algorithms for joining partitions does change the predicted speedup factors, but it does not affect the general conclusion of the model (that partitioning is beneficial).

4.2 Partitioned Set Join Algorithm (PSJ)

Now we are ready to describe the Partitioned Set Join Algorithm (PSJ), uses a two level partitioning scheme. It operates in three phases:

- **Partitioning Phase**: Each tuple of R is sent to exactly one partition based on the first level partitioning function h. Each tuple of S, in general, is replicated across multiple partitions using (the same) h.
- **Joining Phase**: Each partition of R is joined with its counterpart in S using a second level partitioning function that operates on signatures. Hence false drops are possible.
- **Verification Phase**: The tuple pairs that the join phase indicates could join, are compared to remove any false drops.

The subsequent sections decribe each of the phases in detail.

4.3 Partitioning Phase

This phase uses a partitioning function h that operates on the set elements. The partitioning phase begins by reading the relation R. For each tuple r of R, the following steps are executed

1. A 3-tuple (c_i, s_i, OID_i) is computed, where c_i is the set cardinality, s_i is the signature of the set instance, and OID_i is the OID of the tuple.
2. A random element e_R is picked from $r.\{b\}$.
3. The 3-tuple is sent to the partition determined by $h(e_R)$.

Observe that the 3-tuple for each tuple of R is sent only to one partition. Now the relation S is read. For each tuple s of S, the following steps are executed

1. A 3-tuple (c_i, s_i, OID_i) is computed.
2. *For each* element $e_S \in s.\{d\}$, the 3-tuple is sent to the partition determined by $h(e_S)$.

Note that if $r.\{b\} \subseteq s.\{d\}$ then the partition determined by $h(e_R)$ will contain the 3-tuples corresponding to r and s. Hence the algorithm computes containment correctly.

4.4 Joining Phase

During the joining phase, each partition of R is joined with its counterpart in S. There are various algorithms that could be used in this phase. However, at this point, the tuples in each partition do not carry the actual set instances since they are approximated by signatures. Hence the join algorithm in this phase has to operate directly on signatures. In this phase, we use a partition based in-memory algorithm using signatures.

The joining algorithm works in two steps: the *build step* and the *probe step*. In the build step, an array A of size equal to the number of bits in the signature is constructed. Now the partition R_i is scanned and each 3-tuple (c_i, s_i, OID_i) is read. A bit position m that is set to 1 is chosen randomly from the signature. The 3-tuple is inserted into $A[m]$. At the end of first step, the signatures from partition R_i have been partitioned.

During the probe step, partition S_i is scanned. For each 3-tuple (c_j, s_j, OID_j) the chain of signatures in $A[n]$ is examined whenever bit n is set to 1 in s_j. The containment predicate is evaluated (as in Section 3.3) for each signature encountered in the chain and the candidate pairs (OID_i, OID_j) are inserted into a temporary relation. These candidate pairs potentially satisfy the containment relationship.

This phase of the algorithm is similar to signature hash join (SHJ) proposed in [HM97]. We use a single bit in the signature to determine the array index for R. SHJ in general uses more bits (a partial signature) to determine the array index. For S, SHJ requires all possible subset signatures to be enumerated for a given partial signature to determine the chains to be probed. This enumeration is exponential.

4.5 Verification Phase

In the verification phase, we examine the actual R and S tuples to determine whether they satisfy the join condition. The main issues involved in this phase are speeding up set containment verification and avoiding random seeks while fetching the tuples.

The set elements in the tuples of R and S retrieved from the storage manager are stored in the nested internal representation (as described in Section 3.1). Such a storage representation is not very efficient for evaluating the containment predicate for the oid pair (OID_{iR}, OID_{jS}). This is because it requires examining all the set elements of the tuple corresponding to OID_{iS} for every set element of the tuple corresponding to OID_{iR}. In order to speed up the containment verification, the set elements of each tuple of R are inserted into a hash table to facilitate faster probing. Observe that each tuple has a hash table of its own. The set instances of each tuple in S can either be directly scanned in the nested internal representation sequentially or can be converted into an in-memory array representation and accessed. The former approach is expensive since each set element has to be converted into an in-memory representation before probing into the hashed set instances of R. Hence it is not efficient if the same tuple is accessed repeatedly. On the other hand, the array representation is advantageous since it amortizes the cost of conversion into an array over multiple accesses, thereby improving the overall time. Using these combinations, we get the best speedup

for set containment checking. A detailed study of the options for speeding up various pair-wise set operations can be found in [RPN].

In order to minimize the disk seeks we employ a strategy described in [Val87]. The OID pairs relation is already in sorted order with OID_{iR} as the primary key. This makes the access to R tuples sequential. Now as many S tuples as possible are fetched such that the available memory holds (i) tuples of R and its hashed set instances, and (ii) tuples of S and its set instances in array form and (iii) the corresponding array of (OID_{iR}, OID_{jS}) pairs. The OIDs of R are swizzled to point to the R tuples in memory and then the array is sorted on OID_{jS} so that the access to the tuples of S is sequential. The nested set instances present in the swizzled tuples of R are converted into efficient hashed representations in memory. Then the S tuples are read sequentially into memory and their nested set valued attribute is converted into an array representation. The join condition is evaluated for each OID pair by scanning the set instances in S in array representation and probing into the hashed instances of R till all set elements of R are accounted for. We chose to build a hash table for set instances of R rather than S since they are smaller, reducing the cost of building the hash table.

4.5.1 Estimation of Number of Partitions

As seen in Section 4.1, the number of partitions has a critical impact on the performance of PSJ. The desirable number of partitions depends on two parameters: the average set cardinality and the relation cardinality of the relations involved. Even though the speedup is expected to increase as the number of partitions is increased, in practice, the overhead associated with each partition prevents such unbounded speedup.

In order to estimate the desired number of partitions (P_{PSJ}) and the signature size (F_{PSJ}), we employed a detailed analytical model to predict the execution time of the algorithm. Using the model, the cost of each phase is calculated.

Cost of Partitioning Phase: The cost of partitioning includes the cost of reading the relations, cost of determining the partitions and the partitioning overhead. It can be calculated as
Cost of partitioning Phase

$\| R \| IO_{seq} + \| S \| IO_{seq} +$	I/O cost of reading the relations R and S
$P_{PSJ} \left\lceil \frac{\|R\| \times T}{P_S \times P_{PSJ}} \right\rceil IO_{rand} +$	I/O cost of writing the partitions of R
$P_{PSJ} \left\lceil \frac{r_S \|S\| \times T}{P_S \times P_{PSJ}} \right\rceil IO_{rand} +$	I/O cost of writing the partitions of S
$h_c \mid R \mid + h_c k_S \mid S \mid$	CPU cost of computing the hash function

where T is the size of 3-tuples. It is equal to $(TID_s + \lceil F_{PSJ}/8 \rceil + i)$ where i is the number of bytes requires to store the set cardinality. The cost of partitioning assumes that at least a page of each partition fits in memory.

Cost of Joining Phase: The cost of the joining phase is a summation of the joining cost of individual partitions. The joining cost in turn depends on the cost of reading the partitions,

number of signature comparisons and writing the result. Estimating the total number of signature comparisions depends on (i) length of the chain and (ii) expected number of chains examined for each signature of a partition of S. Assuming a uniform distribution, one can determine the length of the chain C_L as

$$C_L = \frac{\mid R \mid}{F_{PSJ} P_{PSJ}} \tag{10}$$

Now we have to determine the expected number of chains examined for a signature from a tuple of S. It depends on the number of bits set to 1. The probability that a bit position b is set to 1 is given by $1/F_{PSJ}$. If m is the expected number of bits set to 1, then we have

$$m = F_{PSJ} \times Prob\{b_t^i = 1\} \tag{11}$$

Now m can be rewritten as

$$m = F_{PSJ} \left(1 - \left(1 - \frac{1}{F_{PSJ}} \right)^{k_S} \right) \tag{12}$$

Since m also equals the number of chains examined, the total number of comparisons N_p required for each partition is given by

$$N_p = \frac{m \mid R \mid\mid S \mid}{F_{PSJ} P_{PSJ}^2} \tag{13}$$

Now the total cost of joining can be calculated as
Cost of joining phase =

$P_{PSJ} \left\lceil \frac{\mid R \mid \times T}{P_S \times P_{PSJ}} \right\rceil IO_{seq} +$	I/O cost of reading the partitions of R
$P_{PSJ} \left\lceil \frac{r_S \mid S \mid \times T}{P_S \times P_{PSJ}} \right\rceil IO_{seq} +$	I/O cost of reading the partitions of S
$\frac{m \mid R \mid\mid S \mid s_c}{F_{PSJ} P_{PSJ}} +$	CPU cost of comparing the signatures
$\left\lceil \frac{\sigma \mid R \mid\mid S \mid V}{P_S} \right\rceil IO_{seq}$	I/O cost of writing the result

where V is the size of the OID pairs which is $2 \times TID_s$.

Cost of Verification Phase: The cost of verification includes the I/O cost of fetching the tuples and CPU cost of evaluating the containment. It is calculated as
Cost of verification phase =

$2 \left\lceil \frac{\sigma \mid R \mid\mid S \mid V}{P_S} \right\rceil IO_{seq} +$	I/O cost of sorting the result
$\sigma \mid R \mid\mid S \mid e_c log(\frac{MP_S}{V}) +$	CPU cost of sorting the result
$2\sigma \mid R \mid\mid S \mid IO_{seq} +$	I/O cost of fetching the tuples in the worst case
$\sigma \mid R \mid\mid S \mid h_c(k_R + k_S)$	CPU cost of verifying the set containment

The overall cost of the algorithm is the sum of the cost of these individual phases. The cost of the algorithm is minimum when the number of partitions is optimal. In order to derive an equation for the partitions, we differentiate the total cost with respect to P_{PSJ} and equate to zero and solve for P_{PSJ}. However, if we differentiate directly the overhead for fragmentation in the partitions will not be taken into account. This is because of the presence of the ceiling functions in various

phases. Fragmentation and other partitioning overhead heavily depends on the implementation of the system. They depend on how much is size of the page, the overhead per tuple, time to create and delete partitions, the cost of pinning and unpinning a page in the partition.

In order to model the partitioning overhead for our system, we tried a few experiments. Based on these observations we modeled the overhead as follows:

- Approximate the fragmentation effect as a quadratic function in P_{PSJ} as $0.5P_{PSJ}^2 IO_{rand}$ in the partitioning phase for each relation and $0.5P_{PSJ}^2 IO_{seq}$ for reading the partitions again in the joining phase.
- The overhead of creating and destroying partitions is again approximated by a quadratic function in P_{PSJ} and modeled by a term of the form HP_{PSJ}^2 where H is a system depend constant.

Even though the replication factor of S relation is related to P_{PSJ} as described Section 4.1, it is substituted by its average set cardinality for simplification. After these additions and substitution for m, the overall cost of the algorithm is differentiated with respect to P_{PSJ}. We get
$dC/dP_{PSJ} =$

$$2P_{PSJ}(IO_{rand} + IO_{seq} + H) - \mid R \mid\mid S \mid \left(1 - \left(1 - \frac{1}{F_{PSJ}}\right)^{k_S}\right) s_c / P_{PSJ}^2$$

Setting $dC/dP_{PSJ} = 0$ and solving for P_{PSJ} gives

$$P_{PSJ} = \left(\frac{\mid R \mid\mid S \mid \left(1 - \left(1 - \frac{1}{F}\right)^{k_S}\right)}{Z}\right)^{1/3} \tag{14}$$

where $Z = 2IO_{rand} + 2IO_{seq} + H$

The fudge factor H accounts for various system dependent factors including cost of creating and destroying partitions and other overheads. The fudge factor is likely to vary across systems. For a given system, H can be determined as follows: for a set of sample data, run PSJ for various number of partitions and measure the partition creation and deletion times. For each of the time, divide by P_{PSJ}^2 for various number of partitions and calculate the average value of H.

If the number of buffer pool pages available is less than the estimated value, whatever is available is committed to the algorithm. As shown in Section 4.1 even though the speedup will still occur but it is not necessarily be the maximum. Hence PSJ is easily adaptable to multiuser environment.

4.5.2 Estimation of Signature Size

The performance of PSJ depends on the size of the signature used for approximating sets. Since partitioning avoids many redundant comparisons, one can expect the signature size (F_{PSJ}) to be lower (when compared to Sig-NL). Also, as the number of partitions is increased the signature size is expected to get lower. We derive an equation for approximately estimating the signature size

based on desirable number of false drops. In order to compute the total number of false drops, first the false drops per partition has to be determined. The false drop probability equation (1) can be rewritten as

$$P_{FD_p} = \frac{falsedrops_p}{N_p - resultsize_p - falsedrops_p} \tag{15}$$

where P_{FD_p} is the false drop probability per partition, $falsedrops_p$ is the number of false drops per partition, $resultsize_p$ is the size of the result after joining corresponding partitions of R and S and N_p is the total number of signature pair comparisons for joining the partition. N_p is given by equation (11).

Letting the false drops f_p to be a percentage of result size of the partition and substituting for m, the equation (15) can be rewritten as

$$P_{FD_p} = \frac{f_p \sigma_p \mid R \mid r_S \mid S \mid}{\mid R \mid\mid S \mid \left(1 - \left(1 - \frac{1}{F_{PSJ}}\right)^{r_S}\right) - \sigma_p \mid R \mid r_S \mid S \mid - f_p \sigma_p \mid R \mid k_S \mid S \mid} \tag{16}$$

where σ_p is the selectivity of the join per partition. Assuming uniform distribution, the partition selectivity can be computed by observing that the total result is the summation of the results from individual partitions.

$$\sigma \mid R \mid\mid S \mid = P_{PSJ} \mid R \mid k_S \mid S \mid \sigma_p / P_{PSJ}^2 \tag{17}$$

Rearranging we get

$$\sigma_p = \sigma P_{PSJ} / r_S \tag{18}$$

Because of uniform distribution we can further assume that $f_p = f$ and $P_{FD_p} = P_{FD}$ where f is the overall percentage of false drops and P_{FD} is the overall false drop probability. Now the equation can be rewritten as

$$P_{FD} = \frac{f \sigma P}{\left(1 - \left(1 - \frac{1}{F_{PSJ}}\right)^{k_S}\right) - \sigma P - f \sigma P} \tag{19}$$

Combining equations (4) and (19) we get

$$(1 - e^{-k_S / F_{PSJ}})^{k_R} - \frac{f \sigma P}{\left(1 - \left(1 - \frac{1}{F_{PSJ}}\right)^{k_S}\right) - \sigma P - f \sigma P} = 0 \tag{20}$$

We use bisection method to solve this equation. There is a cyclic dependency between equations (14) and (20) Hence both the equations have to be solved simultaneously. We use these equations to determine the appropriate combination of partitions and signature size in our experiments for PSJ. As we shall see in Section 5.8 and Section 5.9, fortunately the performance curves as a function of the number of partitions and signature size are rather flat. So these equations do not have to be exact for reasonable performance.

5 Performance Evaluation

In this section, we evaluate the performance of the three set containment algorithms: the SQL approach for the unnested external representation (**SQL**), and the signature nested-loops (**Sig-NL**) and **PSJ** algorithms for nested internal. As a special case, we also ran PSJ with one partition which we call **PSJ-1**. The special case of one partition is important when applicable, because it has no partitioning overhead. We first describe our implementation of these algorithms and then present results from various experiments designed to investigate the performance of these algorithms under various conditions.

5.1 Implementation

Paradise is a shared nothing parallel object-relational system developed at the University of Wisconsin-Madison [PYK+97]. We implemented sets using the ADT mechanism in Paradise. The set ADT implements a number of set-oriented methods, including: create-iterator, which returns an iterator over the elements of the set; and set operators which are implemented by type specific methods invoked by the query engine when comparison and assignment are performed on sets. For more details on the implementation, refer to [RNM99], [RPN].

We implemented signature-nested loops (Sig-NL) and PSJ as join algorithms in the system, and extended the optimizer to recognize set containment join operations in queries. For the SQL approach, magic set optimization was used to rewrite the correlated nested query as shown in Section 3.2. In order to ensure that the optimizer did not choose bad plans, optimal physical plans for each query were fed into the system rather than the queries themselves. The plans are shown in Figure 4.

5.2 Experimental Setup and Data Generation

In our experiments, the total size of the non set-valued attributes in a tuple was 68 bytes. The average size of each set element was 30 bytes. We ran the experiments on an Intel 333 MHZ Pentium processor with 128MB of main memory running Solaris 2.6. We used a 4GB disk for storing the database volume. The disk was mounted as a raw device. It provided an I/O bandwidth of 6 MB/sec. Paradise was configured with a 32MB buffer pool. Though this buffer pool size may seem small compared to current trends in memory, we used this value since we wanted to test data sets that were much larger than the buffer pool. As will be seen in the following sections, with this buffer pool size, some experiments take many days to run. Each experiment was run against a cold buffer pool to eliminate the effect of file caching. The data generator for the BUCKY benchmark [CDN+97] was modified to generate data synthetically. The data generator takes as input the cardinality of the relations R and S, the average cardinality of the set valued attributes in the two relations, the size of the domain from which the set elements are drawn, and a correlation value. For each tuple, the set-valued attribute is generated as follows. First, the data generator divides the entire domain into 50 smaller sub-domains. The set elements are drawn from these

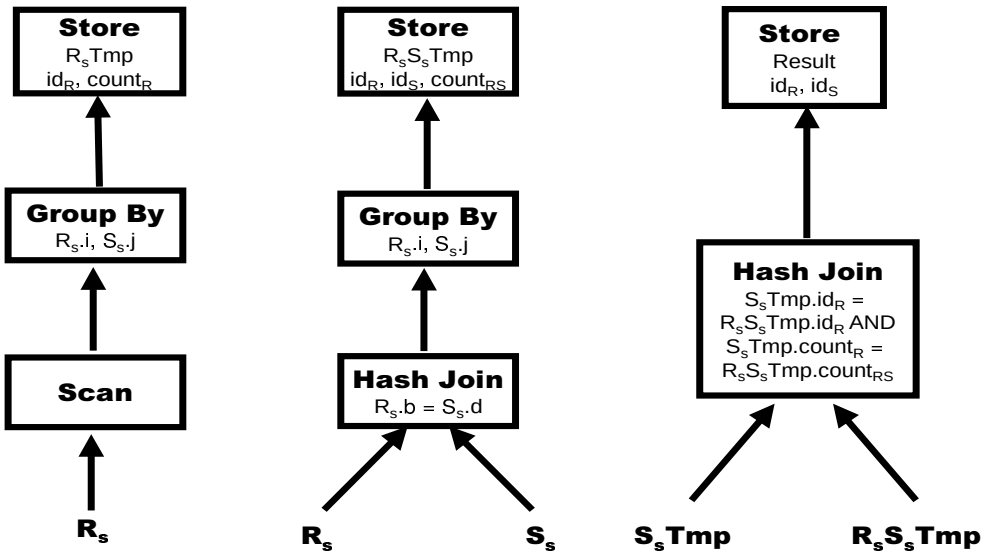

Figure 4: Physical Plans for Count, Candidate and Verify Queries Respectively

sub-domains. Set elements are correlated if they are drawn from the same sub-domain. Correlation of a set instance is defined as the percentage of the set elements that are drawn from a single sub-domain. For example, if the set cardinality is 10, a correlation of 90% implies that 9 set elements are picked from one sub-domain and 1 element is randomly chosen from one of the remaining 49 sub-domains. All the experiments used a correlation of 10% unless otherwise specified. Joining tuples were generated such that every R tuple joins with exactly one S tuple. Finally, we chose the response time as our performance metric.

5.3 Set Distributions

There are many distributions involving set valued attributes because there are many degrees of freedom:

- Average set cardinality of relation R and S
- Relation cardinality of R and S
- Size of domain from which the set elements are drawn
- Degree of correlation among the elements.

Each parameter can influence the performance of the containment algorithm. In an effort to reduce the problem space, we restricted ourselves to varying the relation and set cardinalities. Based on these two parameters we have four possible quadrants as shown in Figure 5.

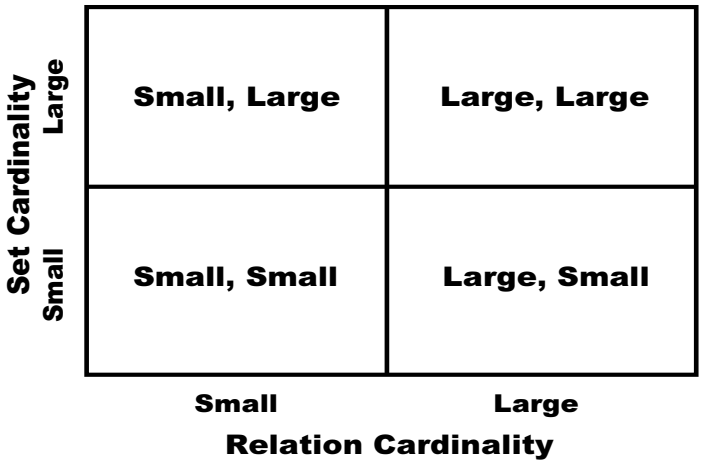

Figure 5: Taxonomy of Set Distributions

Here we give an example for each quadrant. If sets are mainly used as a logical collection (e.g set of courses, set of pre-requisites, set of hobbies and set of outgoing links in a web page) the average set cardinality is small typically in the range of 5-10 however the relation cardinality might be potentially large (number of students, number of employees). On the other hand, if each set instance is considered as a relation (department and set of employees working for it), then the average set cardinality can be pretty large however the relation cardinality might be small (number of department in a company). In the XML world since anything can be represented as sets with it is possible that the average set cardinality and relation cardinality is pretty large (number of documents).

5.4 Experiment 1: Varying Relation Cardinality

In this set of experiments, we investigated the effect of varying the relational cardinality. The domain size was fixed at 10000. Since the join was not symmetric, we further refined the experiments based on different cardinalities of R and S :

- The relation cardinalities of R and S are varied together and the values are kept the same.
- The relation cardinality of S is kept constant at a large value and that of R is varied.
- The relation cardinality of R is kept constant at a large value and that of S is varied.

5.4.1 Varying Relation Cardinalites of R and S

In this experiment, the relation cardinality was varied for two values of set cardinality: 20 and 120. The results of these experiments are plotted in Figure 6. The numbers for the SQL approach for relation cardinalities greater than 20000 are not included in the figure since these runs took more than 24 hours. The main observation is that PSJ outperforms (or performs as well as) other algorithms consistently over the entire space of relation cardinality. On the other hand, the SQL

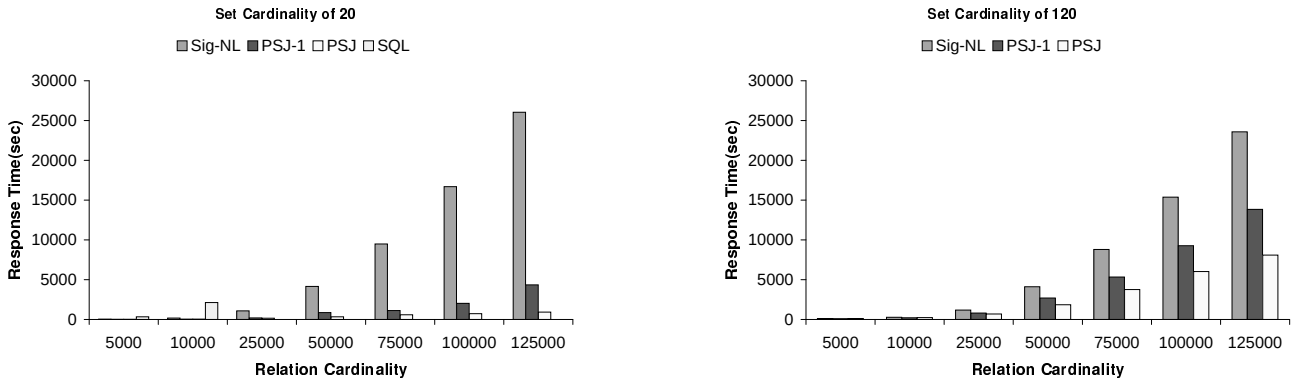

Figure 6: Sig-NL vs PSJ for Set Cardinalities of 20 and 120

approach starts getting worse from 10000 onwards. Section 5.5 discusses why the SQL approach performs poorly. Sig-NL and PSJ are analyzed in Section 5.6.

5.5 Performance of the SQL Approach

As seen from Figure 6, the SQL approach performs reasonably well at very small relation and set cardinalities. However, as the relation sizes increase (note the peak at 10000), the response time increases rapidly. The cost breakdown of the SQL approach is shown in Figure 7. The figure shows the times taken for running each of the component queries. It is evident that most of the time is dominated by the candidate generation query. The candidate generation query is expensive because of the following reasons:

- The input to the joins are two large set relations R_S and S_S. These relations suffer from cardinality explosion (their cardinality is the product of average set cardinality and relation size of the base relations). Such an explosion makes the join expensive.
- The number of intermediate tuples generated as a result of the join is also large. The output of the join generates a tuple for every element in R_S and its intersecting set in S_S. Essentially, it is computing an intersection which is a superset of the actual result. Note that the probing phase of the join and the aggregate phase were run in pipelined fashion so that there is no intermediate I/O cost.
- The number of groups generated from the aggregate operator is also large. The number of groups is proportional to $\mid R \mid \times \mid S \mid$ and it is equal to the number of set pairs that have at least one element in common. The number of groups actually generated is plotted against number of tuples generated, in Figure 8. The figure confirms the enormity of the number of tuples (groups) generated. This number is large even for smaller set and relation cardinalities.

Because of the aforementioned problems and consequent performance degradation, the SQL approach is not considered in the remaining sections.

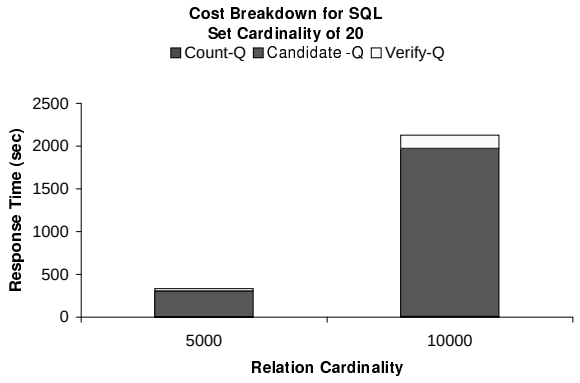

Figure 7: Cost Breakdown for SQL Approach

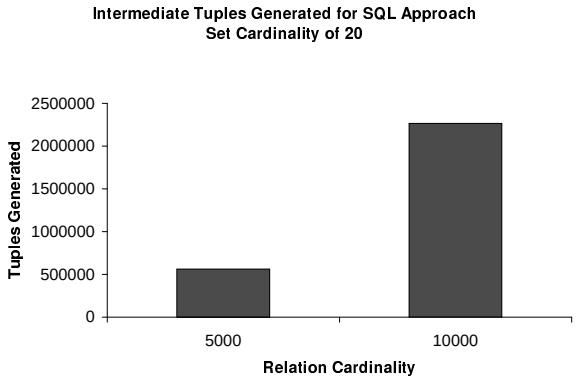

Figure 8: Intermediate Tuples Generated

5.6 Sig-NL Vs PSJ

The individual cost breakdown of these algorithms is shown in Figure 9, Figure 10 and Figure 11.

In general, the cost of these algorithms consists of three components: partitioning cost, comparison cost and verification cost. Sig-NL and PSJ-1 do not have any partitioning cost. The comparison cost is high in Sig-NL. It decreases in PSJ-1 and is least in PSJ.

The first observation is that PSJ outperforms PSJ-1 and Sig-NL consistently as seen from Figure 6. The basic insight is that if PSJ is to perform well, the reduction in the number of comparisons should be significant and the partitioning cost should not be too high. The reduction in number of comparisons is dominant at higher relation cardinalities as seen in Figure 10 and Figure 11. Hence PSJ consistently performs better at higher relation cardinalities. For lower relation cardinalities, the cost gained by avoiding unnecessary comparisons is not high. The gap between PSJ and the rest is smaller for set cardinality of 120. This is because the partitioning cost is higher. In addition, the comparison cost also increases because of replication.

Another contributing factor is the requirement of large signature sizes for lower set cardinalities of R. This might seem counterintuitive. However, a closer look at the false drop probability equation (4) reveals the following characteristics:

- For a given number of false drops and for a constant set cardinality of S, as the average set cardinality of R increases, the size of the signature decreases. For a given signature size, when the set cardinality increases, more bits get turned to 1 in R tuple signature. Hence the probability of a false drop is now reduced since for a tuple to match this signature it must match with the signature in all these bit positions in S tuple signature. As the false drops are kept constant, the effect of increasing cardinality decreases the signature size.

- For a given number of false drops and for a constant set cardinality of R, as the average set cardinality of S increases, the size of the signature increases. As more bits are turned to 1 in a S tuple signature, the probability of a false drop is increased. Because it highly likely that

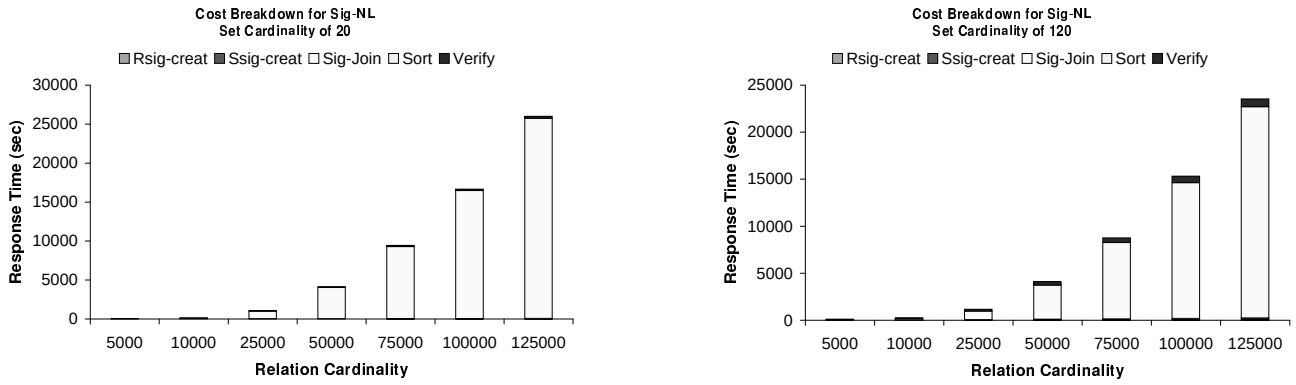

Figure 9: Cost Breakdown for Sig-NL - Set Cardinality of 20 and 120

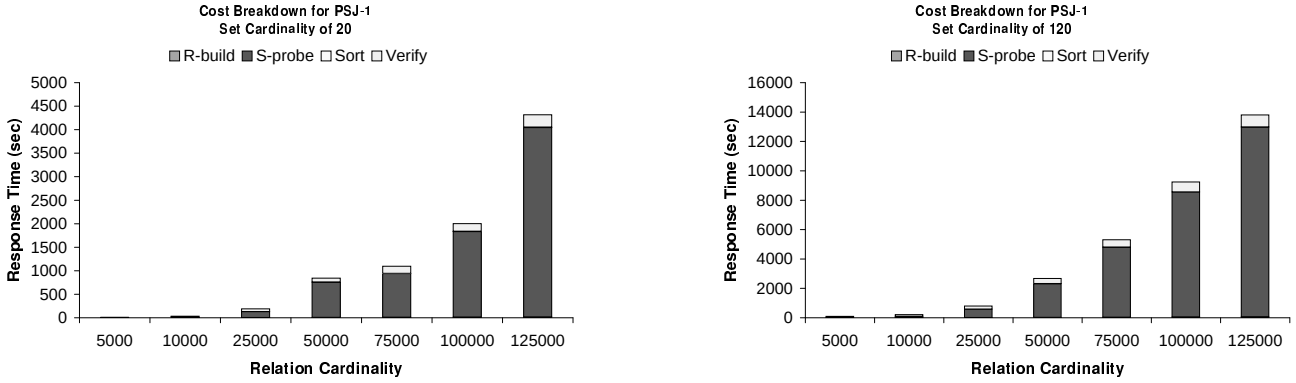

Figure 10: Cost Breakdown for PSJ-1 - Set Cardinality of 20 and 120

bits turned to 1 in R tuple signature will also be turned to 1 in S tuple signature. Since the false drops are kept constant, the effect degenerates into increase in signature size.

Because of the above two opposing effects, an increase in set cardinality decreases the signature size initially reaching a minimum before starts increasing again. Hence in order to keep the false drops minimum, an increase in the signature size is required for smaller set cardinalities. For example, in Sig-NL when the relation cardinality of R (and S) was 25000, the required signature size was 181 bits for a set cardinality of 20 while it was 104 bits for a set cardinality of 120. Note however that as the average set cardinality of S increases, the signature size increases as expected.

The second observation is that PSJ-1 outperforms Sig-NL consistently. This is expected since several unnecessary comparisons are eliminated. Quantitatively, for a set cardinality of 20 and relation cardinality of 25000, Sig-NL requires 625 million comparisons whereas PSJ-1 requires only 80 million comparisons. When the set cardinality is 120, the number of comparisons increases since the expected number of bits set to 1 in the signature increases thereby causing more chains to be examined for a given set of S. Hence the performance gap between the two decreases.

We also conducted experiments where the cardinality of one relation was fixed and the other

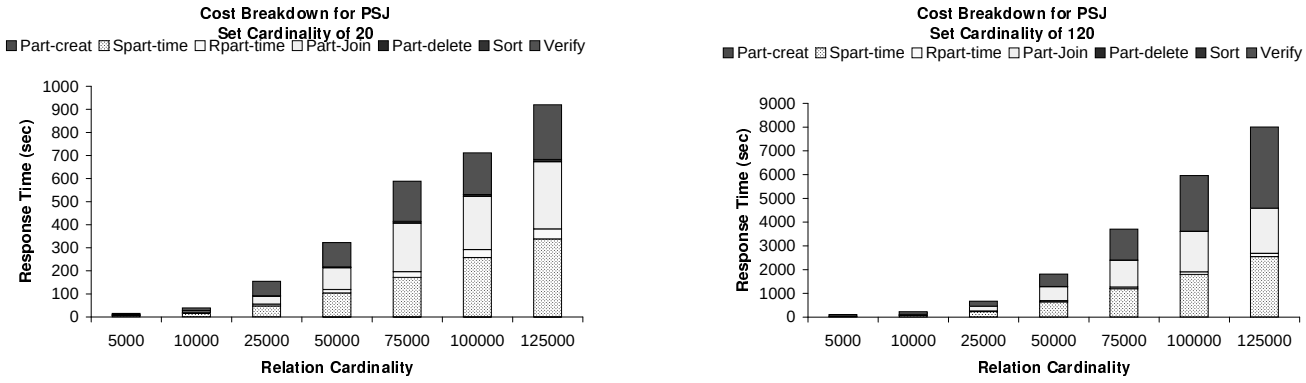

Figure 11: Cost Breakdown for PSJ - Set Cardinality of 20 and 120

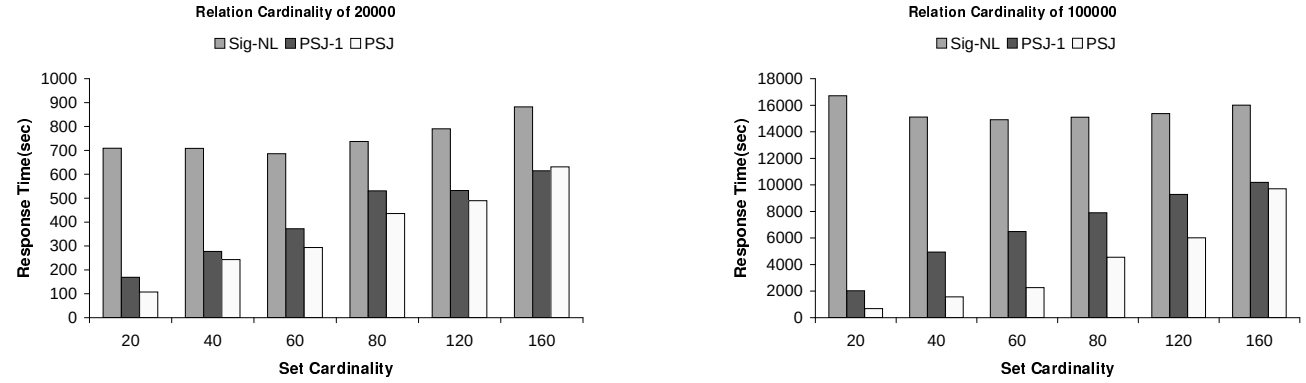

Figure 12: Varying Set Cardinality for Relation Cardinalities of 20000 and 100000

was varied. The trends observed were the same.

5.7 Experiment 2: Varying Set Cardinality

In this experiment, we varied the set cardinality for two different relation cardinalities: 20000 and 100000 to explore the quadrants of small and large relation cardinalities. The signature size for Sig-NL and PSJ-1 and the number of partitions for PSJ were chosen using equations (14) and (20). The domain size was set at 10000. The results are plotted in Figure 12 and the cost breakdown of PSJ-1 and PSJ are shown in Figure 13 and Figure 14.

For a given relation cardinality, as the set cardinality increases, the gap between PSJ and the rest diminishes. In fact for a relation cardinality of 20000 when the set cardinality is 160, PSJ-1 outperforms PSJ. This is because the partitioning cost increases with set cardinality as shown in Figure 14. This happens because more partitions are required and replication is higher. At the larger relation cardinality of 100000, the set cardinality threshold beyond which PSJ-1 outperforms PSJ increases as expected.

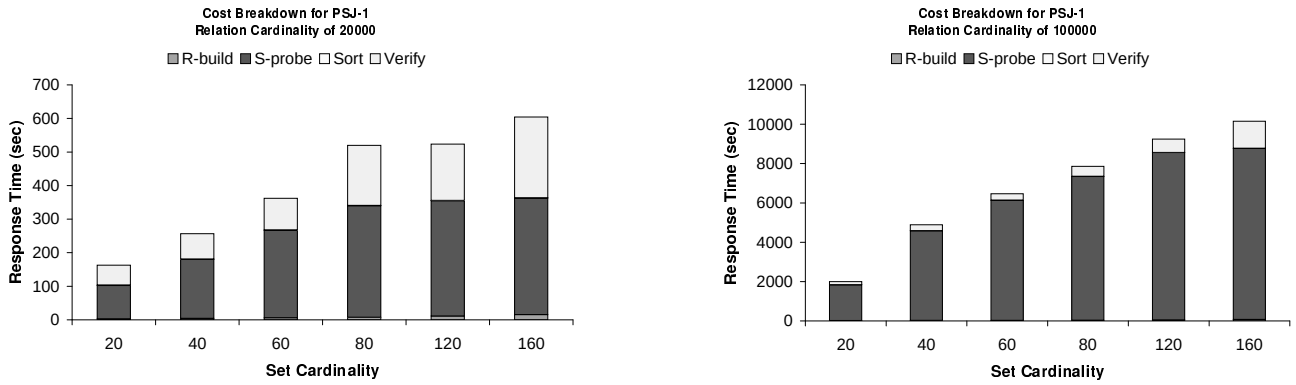

Figure 13: Cost Breakdown for PSJ-1 - Relation Cardinalities of 20000 and 100000

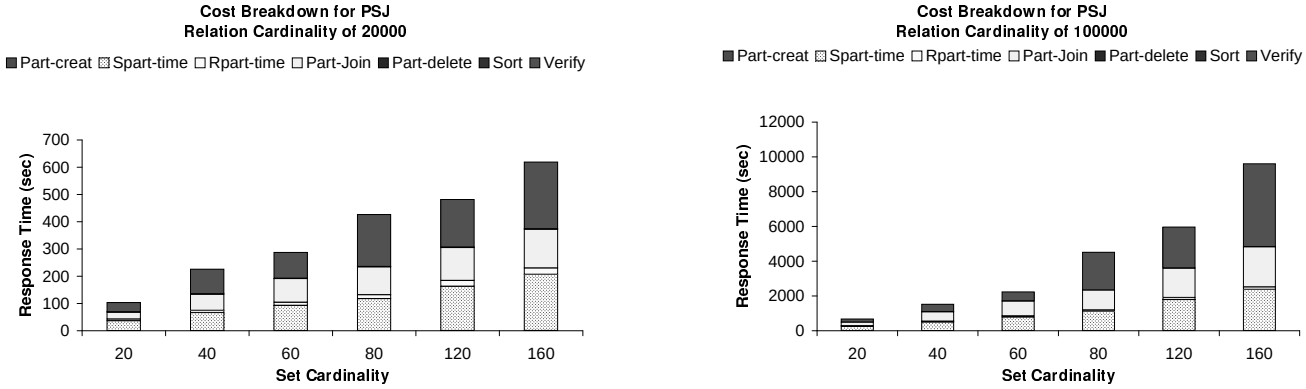

Figure 14: Cost Breakdown for PSJ - Relation Cardinality of 100000

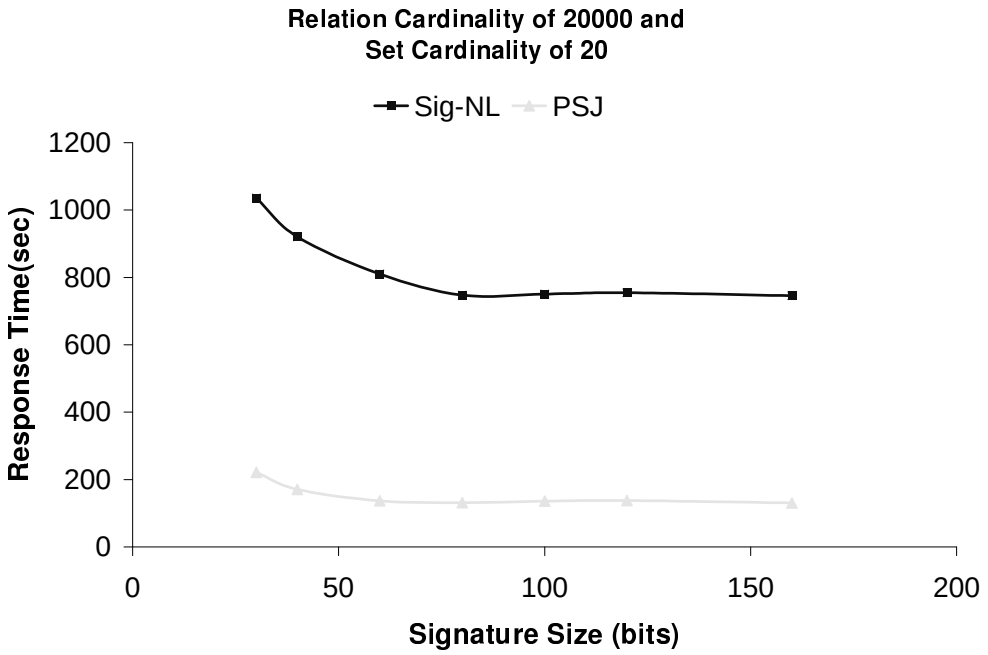

Figure 15: Effect of Signature Size

5.8 Experiment 3: Effect of Signature Size

In this experiment, we study the effect of signature size on the performance of Sig-NL and PSJ. Both algorithms use signatures for producing an intermediate candidate set of result. As noted in Section 3.3, the number of false drops in the candidate set is influenced by the size of the signature. Hence the choice of signature size is important both in Sig-NL and PSJ. For this experiment, we used a relation cardinality of 20,000 for both R and S, an average set cardinality of 10 for R, and average set cardinality of 20 for S. The size of domain was fixed at 10,000. For PSJ, we used the optimal number of 42 partitions, as predicted by equation (14). The result of this experiment is plotted in Figure 15.

The first observation is that for smaller signature sizes, Sig-NL is very expensive. This is because many elements in the domain hash to the same bit, thereby increasing the false drops. Such an increase in the false drops increases the time of the verification phase. As the signature size increases, the number of false drops reduces and hence the performance of Sig-NL improves. However, after a signature size of 80, increasing the signature length does not cause any significant improvement in the performance of Sig-NL. The second observation is that PSJ is relatively immune to the signature size. This is because partitioning reduces the number of false drops.

For this data set, the signature size for Sig-NL predicted by equation (5) was 173 bits. For PSJ with 42 partitions, the signature size predicted by equation (20) was 116 bits. Given the flatness of the PSJ curve, it is not important to get the signature size exactly right.

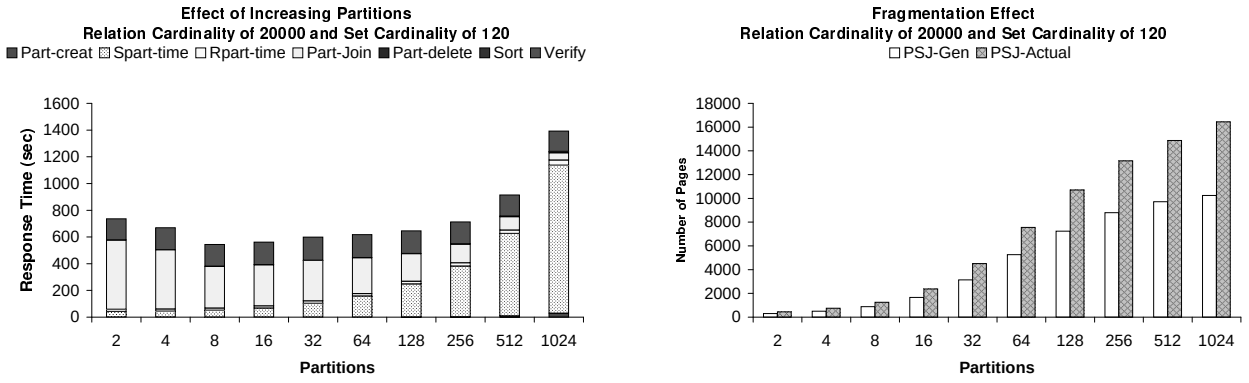

Figure 16: Effect of Increasing Partitions and Fragmentation for Set Cardinality of 120

5.9 Experiment 4: Effect of Increasing Partitions in PSJ

In this experiment, we study the effect of the number of partitions on the performance of PSJ. The relation cardinality of both relations was set at 20,000 and set cardinality was set at 120. The set elements are drawn from a domain size of 10000. An appropriate combination of partitions and signature size was used as determined by equations (14) and (20). The results of this experiment is shown in Figure 16. It shows the breakdown of total cost: partition time, partition creation and deletion times, join time, sort time and verification time. From the figures, we observe that the first graph has three phases: the first phase, in which the total cost decreases gradually as the number of partitions is increased; the second phase in which the total cost is approximately constant; and the third phase in which the total cost starts increasing as the number of partitions becomes very large. In the first phase, when the number of partitions is 1, the join is essentially a signature based partition algorithm with the added overhead of partitioning. As the number of partitions increases, partitioning begins to pay off. However, this improvement in performance is not unbounded. This is when we move into the third phase. In this phase, the partitioning cost starts rising sharply, and this has an unhealthy effect on the overall join performance.

In order to further investigate the sharp increase in partitioning overhead, we plot both the total number of pages generated by the algorithm and the actual number of pages the system uses in Figure 16. It shows that as the number of partitions increases, there is a corresponding increase in the size of the data generated. This is because the replication of S tuples increases. However, the replication of each tuple is bounded by the set cardinality. Hence the increase in the amount of data generated flattens after 64 partitions. The actual number of pages required is substantially higher than the generated data pages. This difference is caused by: per tuple overhead and fragmentation. In addition to increasing fragmentation, the number of pins and unpins, the cost of creating and deleting the partitions also increase with the number of partitions. Thus, the partitioning overhead increases sharply when the number of partitions is large. This experiment shows that the number of partitions has a critical impact on the performance of PSJ. The equation (14) can be used to estimate a reasonable number of partitions. For set cardinality of 120, the number of partitions

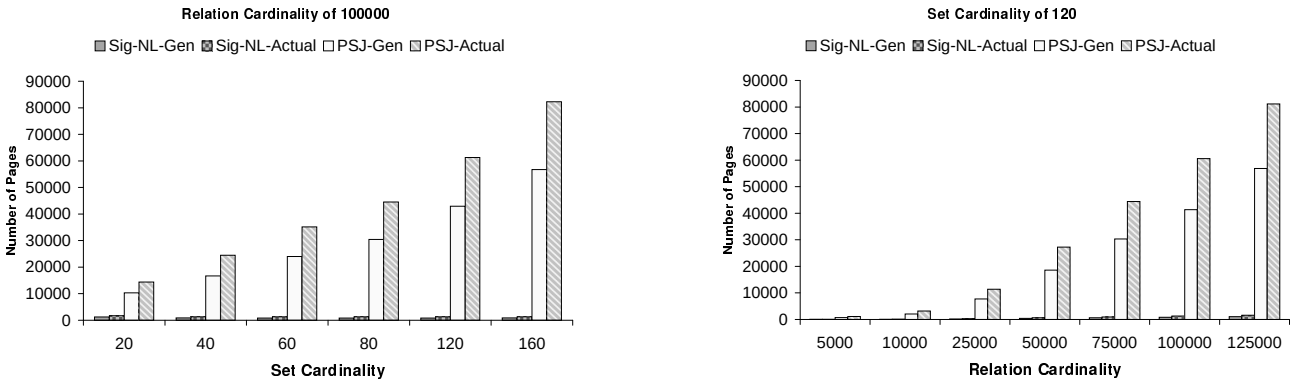

Figure 17: Intermediate Disk Space Requirements for Sig-NL and PSJ

chosen by the equation was 70.

5.10 Experiment 5: Disk Space Requirements

Here we investigate the size of the intermediate space required for Sig-NL and PSJ. We do not consider PSJ-1 since it is an in-memory algorithm. We ran experiments by varying the set cardinality for a relation size of 100000 and also by varying the relation cardinality for a set cardinality of 120. The results are plotted in Figure 17. The graph plots the number of pages generated by each algorithm and the actual number of pages used in disk. The main observation is that Sig-NL requires much less storage than PSJ as expected. The number of pages required by Sig-NL varies slightly because of the variation in signature size. Since the number of pages required by Sig-NL is so low, there is a high probability that these pages will remain in the buffer pool during the operation of the algorithm. Hence Sig-NL can be assumed to be immune from I/O cost except during the signature construction and verification phases. On the other hand, PSJ requires a large amount of intermediate storage that steadily increases as the cardinality increases. This is because a) the number of times the 3-tuple (as described in section 4.3) is replicated increases as set cardinality increases and b) the number of tuples per partition increases as the relation cardinality increases.

For large data sets, the memory requirement for PSJ-1 is very high since the entire set of R signatures has to be accommodated. On the other hand, Sig-NL and PSJ adapt themselves to available amount of memory. Hence they are well suited to a multiuser environment.

6 Conclusions and Future Work

This paper investigates algorithms for computing a set containment join. These algorithms cover two possible implementations of set valued attributes: the unnested external representation and the nested internal representation. The unnested external representation is used by commercial O/R DBMSs for implementing set-valued attributes. In this case, set containment join is implemented using a standard SQL2 query. For the nested internal representation, this paper considers two

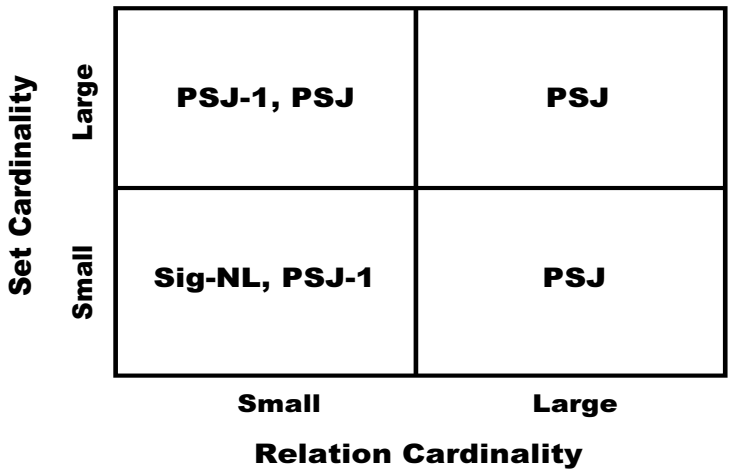

Figure 18: Performance Space of Set Containment Algorithms

algorithms. The first is a variation of nested loops (Sig-NL) that uses signatures to speed up the evaluation of the join predicate. The second algorithm is PSJ, a new partition based algorithm that is proposed in this paper. This algorithm is based on a two level partitioning scheme by using set elements to partition relation R and replicate relation S. Within each partition, it uses an in-memory algorithm based on partitioning of signatures.

This paper also presents a detailed performance study of the three algorithms. The performance space of these algorithms is summarized in Figure 18. For small data sets and small set cardinalities, PSJ works well. The SQL approach and Sig-NL performs reasonably well for extremely small data sets and small set cardinalities; however, as the relation or the set cardinality size increases the performance degrades very rapidly. PSJ with one partition is usable at higher set cardinalities provided there is enough memory. Elsewhere, PSJ is the algorithm of choice.

Since the native SQL approach performed so poorly, we are investigating how the benefits of PSJ can be achieved even in systems that use the unnested external set representation. One obvious approach would be to execute the join by: (a) converting the inputs from the unnested external format to a temporary nested internal approach, (b) doing the join, (c) reconverting the output. In this way the nested internal approach is just an internal data structure of the join algorithm. Clearly this will be much faster than the native SQL over unnested external approach (which took days in some of our tests.) In future work we plan to investigate this and other alternatives.

References

[Bra84] K. Bratbergsengen. Hashing methods and relational algebra operations. In *Proceedings of International Conference on Very Large Databases (VLDB)*, pages 323–333, 1984.

[CDN+97] M. Carey, D. Dewitt, J. F. Naughton, M. Asgarian, P. Brown, J. E. Gerke, and D. N. Shah. The bucky object-relational benchmark. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, 1997.

[DKO+84] D. Dewitt, R. Katz, F. Ohlken, L.Shapiro, M.Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 1–8, 1984.

[DLM93] D. Dewitt, D. Lieuwen, and M. Mehta. Pointer-based join techniques for object-oriented databases. In *PDIS*, 1993.

[DNS91] D. Dewitt, J. Naughton, and D. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *PDIS*, Miami Beach, 1991.

[FC84] C. Faloutsos and S. Christodoulakis. Signature files: An access method for documents and its anaytical performance evaluation. *ACM Transactions on Office Information Systems*, 2(4):267–288, October 1984.

[Fel57] W. Feller. *An Introduction to Probability Theory and Applications*, volume 1. John Wiley and Sons, 1957.

[HKP97] J. Hellerstein, E. Koutsoupias, and C. H. Papadimitriou. On the analysis of indexing schemes. In *Proceedings of Principles of Database Systems (PODS)*, Athens, Greece, 1997.

[HM96] S. Helmer and G. Moerkotte. Evaluation of main memory join algorithms for joins with set comparison join predicates. Technical report, University of Mannheim, 1996.

[HM97] S. Helmer and G. Moerkotte. Evaluation of main memory join algorithms for joins with set comparison join predicates. In *Proceedings of International Conference on Very Large Databases (VLDB)*, Athens, Greece, 1997.

[IKO93] Y. Ishikawa, H. Kitagawa, and N. Ohbo. Evaluation of signature files as set access facilities in oodbs. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 247–256, Washington, D.C., 1993.

[LR96] M. Lo and C. Ravishankar. Spatial hash-joins. In *Proceedings of ACM SIGMOD Conference on Management of Data*, Montreal,Quebec, May 1996.

[MK76] M.W.Blasgen and K.P.Eswaran. On the evaluation of queries in a relational database system. Technical report, IBM, 1976.

[PD96] J. Patel and D. DeWitt. Partition based spatial merge join. In *Proceedings of ACM SIGMOD Conference on Management of Data*, Montreal,Quebec, May 1996.

[PYK+97] J. Patel, J. Yu, N. Kabra, K. Tufte, B. Nag, J. Burger, N. Hall, K. Ramasamy, R. Lueder, C. Ellman, J. Kupsch, S. Guo, J. Larson, D. DeWitt, and J. Naughton. Building a scalable geo spatial dbms: Technology, implementation and evaluation. In *Proceedings of ACM SIGMOD Conference on Management of Data*, Tucson, Arizona, May 1997.

[RKS98] M. Roth, H. Korth, and A. Silberschatz. Extending relational algebra and calculus for nested relational databases. *ACM Transactions on Database Systems*, 13(4):389–417, December 1998.

[RNM99] K. Ramasamy, J. Naughton, and D. Maier. High performance implementation techniques for set valued attributes. http://www.cs.wisc.edu/ ramasamy/sets.ps, 1999.

[RPN] K. Ramasamy, J. Patel, and J. Naughton. Efficient pairwise operations for nested set valued attributes. Working paper.

[SC90] E. J. Shekita and M. J. Carey. A performance evaluation of pointer based joins. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 300–311, 1990.

[SHT+99] J. Shanmugasundaram, G. He, K. Tufte, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *Proceedings of International Conference on Very Large Databases (VLDB)*, Scotland, 1999.

[SPL96] P. Seshadri, H. Pirahesh, and T.Y. C. Leung. Complex query decorrelation. In *Proceedings of IEEE Conference on Data Engineering(ICDE)*, pages 450–458, 1996.

[Sto96] M. Stonebraker. *Object-relational DBMS: The Next Great Wave*. Morgan Kaufmann, 1996.

[Val87] P. Valduriez. Join indices. *ACM Theory of Database Systems (TODS)*, 12(2), 1987.

[Zan83] Carlo Zaniolo. The database language gem. In *Proceedings of 1983 ACM SIGMOD Conference on Management of Data (SIGMOD)*, San Jose, California, May 1983.

[ZMR98] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems*, 23(4):453–490, December 1998.