Published in Proc. 8th Intl. Conf. on Extending Database Technology (EDBT), pp. 427-444, Prague, 2002

Divide-and-Conquer Algorithm for Computing Set Containment Joins

Sergey Melnik* and Hector Garcia-Molina

Stanford University CA 94305, USA

Abstract. A set containment join is a join between set-valued attributes of two relations, whose join condition is specified using the subset ($\subseteq$) operator. Set containment joins are used in a variety of database applications. In this paper, we propose a novel partitioning algorithm called Divide-and-Conquer Set Join (DCJ) for computing set containment joins efficiently. We show that the divide-and-conquer approach outperforms previously suggested algorithms over a wide range of data sets. We present a detailed analysis of DCJ and previously known algorithms and describe their behavior in an implemented testbed.

1 Introduction

Many database applications utilize set containment queries, especially when the underlying database systems support set-valued attributes. For example, consider a database application that recommends to students a list of courses that they are eligible to take. Imagine that the set of courses taken by a student is stored in the set-valued attribute `{courseID}` of table `Attended(studentID, {courseID})`. Another table, `Prereq(courseID, {reqCourseID})`, keeps a set of prerequisite courses. Then, a list of courses that a student may take can be computed using a set containment query `SELECT Attended.studentID, Prereq.courseID WHERE Prereq.{reqCourseID}` $\subseteq$ `Attended.{courseID}`. In this query, the tables are joined on their set-valued attributes using the subset operator $\subseteq$ as the join condition. This kind of join is called set containment join.

Set containment joins are used in a variety of other scenarios. If, for instance, our first relation contained sets of parts used in construction projects, and the second one contained sets of parts offered by each equipment vendor, we could determine which construction projects can be supplied by a single vendor using a set containment join. Or, consider a human resource broker that matches the skills of job seekers with the skills required by the employers. In this scenario, a set containment join on the skills attributes can be used to match the qualifying employees and their potential employers. Notice that containment queries can be utilized even in database systems that support only atomic attribute values, as illustrated in [MGM01] (there we give an example of a set containment query expressed using SQL). Additional types of applications for containment joins

* On leave from the University of Leipzig, Germany

arise when text or XML documents are viewed as sets of words or XML elements, or when flat relations are folded into a nested representation.

It has been shown in [HM97] and [RPNK00] that naive or standard-SQL approaches to computing set containment queries are very expensive. Each of the papers suggested a more efficient way of computing set containment joins. Unfortunately, the algorithm proposed in [HM97] is a main-memory algorithm and cannot cope with large amounts of data. Furthermore, the partitioning-based algorithm suggested in [RPNK00] has limitations when the cardinalities of the sets are large. Often, however, the sets involved in the join computation are indeed quite large. For instance, biochemical databases contain sets with many thousands elements each. In fact, the fruit fly (drosophila) has around 14000 genes, 70-80% of which are active at any time. A snapshot of active genes can thus be represented as a set of around 10000 elements. The algorithm suggested in [RPNK00] is ineffective for such data sets.

The contributions of this paper are two-fold. First, we suggest a novel partitioning-based algorithm that outperforms the existing approaches for a large variety of data sets. We called this algorithm the Divide-and-Conquer Set Join (DCJ), because it can be viewed conceptually as a series of partitioning steps. Second, we develop a detailed analytical model for comparing different partitioning algorithms. We also analyze the behavior of the algorithms in an implemented testbed and discuss how a best-performing algorithm can be chosen for given input relations. To compare DCJ with the approach of [HM97], we develop a partitioning algorithm called the Lattice Set Join (LSJ), which is in essence a disk-based version of the main-memory algorithm introduced in [HM97]. In this paper, we give only a brief summary of the results obtained for LSJ and show that the divide-and-conquer approach always outperforms LSJ.

This paper is structured as follows. In Section 2 we explain how signatures and partitioning are used for computing set containment joins, and illustrate the algorithm that we developed using a simple example. Section 3 deals with the theoretical analysis of our novel algorithm and the existing partitioning algorithms. After that, in Section 4, we provide a qualitative comparison of the algorithms. In Section 5 we examine the performance of the algorithms in an implemented system. Finally, we discuss related work in Section 6 and conclude the paper in Section 7.

2 Algorithms

We start this section with a brief overview of set containment joins, signatures and partitioning. Then we describe the Partitioning Set Join (PSJ) algorithm [RPNK00], which is to our knowledge the best known method for computing set containment joins for relations that do not fit into main memory. Finally, we explain the Divide-and-Conquer Set Join algorithm that we developed.

2.1 Set containment joins, signatures and partitioning

A *set containment* join is a join between set-valued attributes of two relations, whose join condition is specified using the subset ($\subseteq$) operator. Consider two sample relations R and S shown in Table 1. Each of the relations contains one column with four sets of integers. For easy reference, the sets of R and S are labeled using letters a, b, c, d and A, B, C, D, respectively. Computing the containment join $R \bowtie_{\subseteq} S$ amounts to finding all tuples $(s,t) \in R \times S$ such that $s \subseteq t$. In our example, $R \bowtie_{\subseteq} S = \{(a,A),(b,B),(c,C)\}$.

Relation R	Relation S
$a = \{1,5\}$	$A = \{1,5,7\}$
$b = \{10,13\}$	$B = \{8,10,13\}$
$c = \{1,3\}$	$C = \{1,3,13\}$
$d = \{8,19\}$	$D = \{2,3,4\}$

Table 1. Two sample relations with set-valued attributes

$x \in R$	$sig(x)$	$y \in S$	$sig(y)$
a	0010	A	1010
b	0110	B	0111
c	1010	C	1010
d	1001	D	1101

Table 2. 4-bit signatures of sets in R and S

Obviously, we can always compute $R \bowtie_{\subseteq} S$ in a straightforward way by testing each tuple in the cross-product $R \times S$ for the subset condition. In our example, such approach would require $|R| \cdot |S| = 4^2 = 16$ set comparisons. For large relations R and S, doing n^2 comparisons becomes very time consuming. The set comparisons are expensive, since each one requires traversing and comparing a substantial portion of the elements of both sets. Moreover, when the relations do not fit into memory, enumerating the cross-product incurs a substantial I/O cost.

For computing set containment joins efficiently, two fundamental techniques have been suggested: signatures [HM97] and partitioning [RPNK00]. The idea behind signatures is to substitute expensive set comparisons by efficient comparisons of signatures. A *signature* of a set is a hash value over the content of the set that has certain order-preserving properties. To illustrate, consider the example in Table 2. In the table, the signature of each set from the sample relations R and S is represented as a vector of 4 bits. Each set element j turns on a bit at the position ($j \mod 4$) in the bit vector. For instance, for set $d = \{8,19\}$ we set bit 0 ($8 \mod 4$) and bit 3 ($19 \mod 4$), and obtain $sig(\{8,19\}) = 1001$.

Let $\subseteq^b$ be the bitwise inclusion predicate. Notice that $sig(x) \subseteq^b sig(y)$ holds for any pair of sets x,y with $x \subseteq y$. Thus, we can avoid many set comparisons by just testing the signatures for bitwise inclusion. For instance, since $sig(d) \not\subseteq^b sig(A)$, we know that d cannot be a subset of A. Bitwise inclusion can be verified efficiently by testing the equality $sig(x) \& \neg sig(y) = 0$, where & and $\neg$ are the bitwise AND and NOT operators. In our example, after 16 signature comparisons we only need to test 7 pairs of sets for containment: $(a,A),(a,B),(a,C),(b,B),(c,A),(c,C)$, and (d,D). Of these remaining pairs, $(a,B),(a,C),(c,A)$ and (d,D) are rejected as false positives.

Using signatures helps to reduce the number of set comparisons significantly, yet still requires n^2 comparisons of signatures. *Partitioning* has been suggested to further improve performance by decomposing the join task $R \bowtie S$ into k smaller subtasks $R_1 \bowtie S_1, \ldots, R_k \bowtie S_k$ such that $R \bowtie S = \bigcup_{i=1}^{k} R_i \bowtie S_i$. The so-called *partitioning function* π assigns each tuple of R to one or multiple partitions $R_1, \ldots, R_k$, and each tuple of S to one or multiple partitions $S_1, \ldots, S_k$. Consider our sample relations R and S from Table 1. Let $\pi(a) = \pi(b) = \pi(A) = \pi(B) = \{1\}$, $\pi(c) = \pi(d) = \pi(C) = \{2\}$, and $\pi(D) = \{1, 2\}$. That is, R is partitioned into $R_1 = \{a, b\}$, $R_2 = \{c, d\}$, and S is partitioned into $S_1 = \{A, B, D\}$, $S_2 = \{C, D\}$. Note that we have constructed π so that tuples in R_1 can only join S_1 tuples, and R_2-tuples can only join S_2-tuples. Thus, finding $R \bowtie_{\subseteq} S$ amounts to computing $(R_1 \bowtie_{\subseteq} S_1) \cup (R_2 \bowtie_{\subseteq} S_2)$. Notice that computing $R_1 \bowtie_{\subseteq} S_1 = \{a, b\} \bowtie_{\subseteq} \{A, B, D\}$ and $R_2 \bowtie_{\subseteq} S_2 = \{c, d\} \bowtie_{\subseteq} \{C, D\}$ requires only $2 \cdot 3 + 2 \cdot 2 = 10$ signature comparisons. Hence, by using partitioning we reduced the total number of signature comparisons from 16 to 10. We refer to the fraction $\frac{10}{16}$ as a *comparison factor*. The comparison factor ranges between 0 and 1.

Besides reducing the number of required signature comparisons, partitioning helps to deal with large relations R and S that do not fit into main memory by storing the partitions $R_1, \ldots, R_k$ and $S_1, \ldots, S_k$ on disk. To minimize the I/O costs of writing out the partitions to disk and reading them back into memory, the partitions typically contain only the set signatures and the corresponding tuple identifiers. In our example, $|\{a, b\}| + |\{c, d\}| = 4$ signatures from $R_{1,2}$ and $|\{A, B, D\}| + |\{C, D\}| = 5$ signatures from $S_{1,2}$ are stored on disk temporarily. We refer to the ratio between the total number of signatures that are written out to disk and the total number of tuples in R and S as the *replication factor*. In our example, the replication factor is $\frac{4+5}{4+4} = \frac{9}{8}$. Assuming that no partition is permanently kept in main memory, the optimal replication factor that can be achieved in a partition-based join is 1.

A major challenge of effective partitioning is to construct a partitioning function π that minimizes the comparison and replication factors. Obviously, π needs to be correct, i.e., it has to ensure that all joining tuples are found.

2.2 Partitioning Set Join (PSJ)

The Partitioning Set Join (PSJ) is an algorithm proposed by Ramasamy et al [RPNK00]. To illustrate the algorithm, we continue with the example introduced above. Imagine that we want to partition R and S from Table 1 into $k = 8$ partitions. The partition number of each set of R is determined using a single, randomly selected element of the set. Consider the set $a = \{1, 5\} \in R$. Let 5 be a randomly chosen element of a. We assign a to one of the partitions $0, 1, \ldots, 7$ by taking the element value modulo $k = 8$. Thus, a is assigned to partition R_5. Element 10 chosen from $b = \{10, 13\}$ yields partition number $2 = (10 \mod 8)$. Finally, sets c and d both fall into partition R_3 based on randomly chosen elements $3 \in c$ and $19 \in d$. Now we repeat the same procedure for S, but consider *all* elements of each set for determining the partition numbers. Taking all elements into account ensures that all joining tuples will be found. Thus,

$A = \{1, 5, 7\}$ is assigned to partitions S_1, S_5, and S_7, $B = \{8, 10, 13\}$ goes into partitions S_0, S_2, and S_5, etc. The complete partition assignment for R and S is summarized in Figure 1.

Once both relations are partitioned, i.e. the set signatures and tuple identifiers have been written out to disk, each pair of partitions is read from disk and joined independently. For example, when R_3 and S_3 are joined, the signatures of sets c and d are read from R_3, and are compared with the signatures of sets C and D stored in S_3. Hence, computing $R_3 \bowtie_{\subseteq} S_3$ results in $2 \cdot 2 = 4$ signature comparisons. The total number of signature comparisons required in our example amounts to $0+0+2+4+0+3+0+0 = 9$, whereas the total of 16 signatures need to be written out to disk. Thus, in this example, we obtain the comparison factor $\frac{9}{16} \approx 0.56$, and replication factor $\frac{16}{4+4} = 2$.

2.3 Divide-and-Conquer Set Join (DCJ)

The Divide-and-Conquer Set Join (DCJ) is a novel partitioning algorithm that we present in this paper. Again, we illustrate the algorithm using our running example of Table 1 and $k = 8$ partitions. We explain DCJ using a series of partitioning steps depicted in Figure 2. In every step, one *boolean hash function* is used to transform an existing partition assignment into a new assignment with twice as many partitions. This transformation, or repartitioning, is done by applying either operator α or operator β to a given pair of partitions $R_i \bowtie S_i$, as indicated by the labels 'α' and 'β' placed on the forks in Figure 2. Although we illustrate DCJ conceptually as a branching tree, the final partition assignment is computed without using any intermediate partitions (see algorithmic specification in [MGM01]).

Partition	$R_i \bowtie S_i$
0	⋈ B
1	⋈ A C
2	b ⋈ B D
3	c d ⋈ C D
4	⋈ D
5	a ⋈ A B C
6	⋈
7	⋈ A

Fig. 1. Partitioning with PSJ: 9 comparisons, 16 replicated

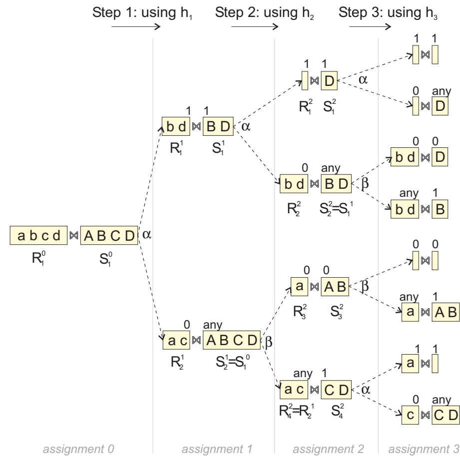

Fig. 2. Divide and conquer: 8 comparisons, 14 replicated

Table 3 defines three boolean hash functions that we will use in our example to partition the relations. Each of the hash functions takes a set of integers as input, and returns 0 or 1 as output. For example, function h_3 returns 1 (i.e. fires) for a given set s if and only if s contains an integer divisible by 5 or by 7. Notice that every of h_1, h_2 and h_3 is *monotone* in the sense that if any of the functions fires for a given set x, then it is guaranteed to fire for each superset of x. The intuition is to make each of the functions fire independently of the other with an approximately equal probability for any randomly selected set of R or S. Each of the functions h_1, h_2, and h_3 are characterized by a set of prime divisors, as shown in Table 3. The boolean values returned by the hash functions for each set of R and S are listed in Table 4. Other types of hash functions can be used by our algorithm. See Section 3 for an additional discussion of hash functions.

Hash fct	Prime divisors	Definition
h_1	$\{2\}$	$h_1(s) = 1 \iff \exists e \in s : (e \mod 2 = 0)$
h_2	$\{3\}$	$h_2(s) = 1 \iff \exists e \in s : (e \mod 3 = 0)$
h_3	$\{5, 7\}$	$h_3(s) = 1 \iff \exists e \in s : (e \mod 5 = 0) \vee (e \mod 7 = 0)$

Table 3. Definition of three boolean hash functions

$x \in R$	h_1	h_2	h_3	$y \in S$	h_1	h_2	h_3
a	0	0	1	A	0	0	1
b	1	0	0	B	1	0	1
c	0	1	0	C	0	1	0
d	1	0	0	D	1	1	0

Table 4. Applying hash functions to elements of R and S

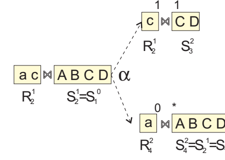

Fig. 3. Why β is needed: applying α when $|R_2^1| < |S_2^1|$ leads to a larger replication factor

Relations $R = \{a, b, c, d\}$ and $S = \{A, B, C, D\}$ form the initial partition assignment $R \bowtie S = R_1^0 \bowtie S_1^0$, where the superscript 0 indicates the step number. In Step 1, we derive a new partition assignment $(R_1^1 \bowtie S_1^1) \cup (R_2^1 \bowtie S_2^1)$ from $R \bowtie S$ using operator α as follows. First, the initial partition R is split into R_1^1 and R_2^1 based on the value of h_1. That is, the sets b and d with $h_1(b) = h_1(d) = 1$ are sent to R_1^1, while the sets a and c with $h_1(a) = h_1(c) = 0$ are assigned to R_2^1. We abbreviate this procedure concisely as $R_1^1 := R/h_1$, $R_2^1 := R/\neg h_1$. The values 1 and 0 taken by h_1 are depicted above the partitions R_1^1 and R_2^1 in Figure 2.

Now imagine that we pick an arbitrary set from S, say set A. Since $h_1(A) = 0$ then, due to monotonicity of h_1, all joining subsets of A are guaranteed to be contained in $R_2^1 := R/\neg h_1$. Therefore, it is sufficient to assign set A to partition S_2^1 only, without losing any joining tuples. The same argument applies for set C. For sets B and D, h_1 takes value 1, meaning that joining subsets of

B and those of D may be found both in R_1^1 and in R_2^1. Hence, B and D need to be replicated, i.e., assigned to both partitions S_1^1 and S_2^1. By doing so, S is repartitioned into $S_1^1 := S/h_1$ and $S_2^1 := S$ based on the values taken by h_1 (1 for S_1^1 and 'any' for S_2^1, as shown in the figure). Thus, Step 1 consists of a single application of operator α using hash function h_1, yielding the partition assignment 1, i.e. $(\{b,d\} \bowtie \{B,D\}) \cup (\{a,c\} \bowtie \{A,B,C,D\})$. Notice that instead of $4 \cdot 4 = 16$ signature comparisons required for $R \bowtie S$, only $2 \cdot 2 + 2 \cdot 4 = 12$ signature comparisons would be needed for joining the partitions of assignment 1.

In Step 2, we use function h_2 to derive the partition assignment 2 from assignment 1. Assignment 2 is obtained by first applying operator α to $R_1^1 \bowtie S_1^1$, and then applying a different operator called β, to $R_2^1 \bowtie S_2^1$. Operator β works in the same way as α, except that S_2^1 is split first, and a portion of R_2^1 is replicated. Because β splits partition S_2^1, which contains supersets, we replicate the sets of R_2^1 with $h_2 = 0$ instead of those with $h_2 = 1$. Thus, the 'top' partition of R_2^1 becomes $R_2^1/\neg h_2$, while the 'bottom' partition contains all of R_2. Clearly, operator β performs a correct repartitioning; since h_2 is monotone, all joining supersets of the sets in R_2^1/h_2 are guaranteed to be contained in S_2^1/h_2. Thus, it is sufficient to replicate only those sets from R_2^1 with $h_2 = 0$, i.e. just the set a.

Operator	Ideally, when	Resulting partition assignment
$\alpha(R \bowtie S, h)$	$\lvert R\rvert \geq \lvert S\rvert$	$(R/h \bowtie S/h) \cup (R/\neg h \bowtie S)$
$\beta(R \bowtie S, h)$	$\lvert R\rvert < \lvert S\rvert$	$(R/\neg h \bowtie S/\neg h) \cup (R \bowtie S/h)$

Table 5. Repartitioning of $R \bowtie S$ using operators α and β, and a monotone boolean hash function h

The definition of β, along with that of α, is shown in Table 5. Either operator α and β performs correct repartitioning and can be applied at each fork. The reason for using β in addition to α is to minimize replication. Notice that by using operator β for repartitioning $R_2^1 \bowtie S_2^1$ into $(\{a\} \bowtie \{A,B\}) \cup (\{a,c\} \cup \{C,D\})$, we reduced the number of comparisons from 8 to 6, and increased the number of signatures that need to be stored from 6 to 7. Figure 3 illustrates what would have happened if we used operator α. Although we would have obtained the same reduction in number of comparisons, the number of signatures to be replicated would have grown to 8. Since $|R_2^1| < |S_2^1|$, β causes less replication by splitting the larger partition S_2^1 and replicating the smaller partition R_2^1.

Ideally, we would apply α for repartitioning of any given $R_j \bowtie S_j$ whenever $|R_j| \geq |S_j|$, and use β when $|R_j| < |S_j|$. However, since generating and writing out the intermediate partitions to disk is prohibitive, their exact sizes are not known. Consequently, we use a simple heuristic. To minimize the replication factor, α and β are applied in an alternating fashion as suggested by fork labels in Figure 2. That is, the 'top' pair of partitions produced by applying α in Step i is repartitioned using α in Step $i+1$. In contrast, the 'bottom' pair is repartitioned using β. Thus, we use the pattern $\alpha \to \alpha, \beta$. Similarly, the 'top'

and 'bottom' pair of partitions produced by β are repartitioned using β and α, respectively (pattern $\beta \rightarrow \beta, \alpha$). The intuition behind this heuristic is to always use β on partitions that were replicated in the previous step. For example, since S_2^1 was obtained by replication in Step 1, we apply β to split it in Step 2.

In Step 3, we arrive at the final partition assignment by applying function h_3 (since we are using 8 partitions, and each repartitioning step doubles the number of partitions, the number of steps is $\log_2 8 = 3$). In the final assignment, the total of $0+0+2+2+0+2+0+2 = 8$ signature comparisons are required, whereas 14 signatures need to be written out to disk. Thus, we obtain a comparison factor of $\frac{8}{16} = 0.5$, and a replication factor of $\frac{14}{4+4} = 1.75$.

3 Analysis of the algorithms

In this section we present a summary of the theoretical analysis of the partitioning algorithms PSJ, DCJ, and LSJ. For a detailed discussion, which includes the derivation of all formulas, please refer to [MGM01]. As an efficiency measure of the algorithms we utilize the comparison and replication factors. Recall that the comparison factor is the ratio between the actual number of signature comparisons, and $|R| \cdot |S|$. In other words, the comparison factor is the probability that the signatures of two randomly selected sets $r \in R$ and $s \in S$ will be compared during the join computation. The replication factor is the ratio of the number of signatures of R and S stored on disk temporarily, and $|R| + |S|$. The comparison factor approximates the CPU load, whereas the replication factor reflects the I/O overhead of partitioning.

Set containment join $R \bowtie_{\subseteq} S$ can be characterized by a variety of parameters including the distribution of set cardinalities in relations R and S, the distribution of set element values, the selectivity of the join, or the correlation of element values in sets of both relations. In our analysis, we are making the following simplifying assumptions:

1. The R, S set elements are drawn from an integer domain $\mathcal{D}$ using a uniform probability distribution[1]. The size $|\mathcal{D}|$ of the domain is much larger than the number of partitions k and the set cardinalities of R and S.
2. Each set $r \in R$ contains a fixed number of θ_R elements, while each set $s \in S$ contains θ_S elements, $0 < \theta_R \leq \theta_S$.
3. Joining each pair of partitions R_i and S_i requires $|R_i| \cdot |S_i|$ signature comparisons (for instance, partitions are joined using a nested loop algorithm).

We will relax these assumptions in our experiments in Sections 4 and 5. All other factors relevant to computing the join are considered identical for every of the partitioning algorithms. These factors include the number of bits in the signatures, the size of the available main memory, the buffer management policy of the database system, etc. For estimating the comparison and replication factors, we additionally use a derived parameter $\lambda = \frac{\theta_S}{\theta_R}$ that denotes the ratio

[1] Notice that non-integer domains can be mapped onto integers using hashing.

of the set cardinalities, and the parameter $\rho = \frac{|S|}{|R|}$ that denotes the ratio of the relation sizes. The variables that we utilize for analyzing the algorithms are summarized in Table 6. For instance, for our sample relations in Table 1 we obtain $|R| = |S| = 4$, $\rho = \frac{4}{4} = 1$, $\theta_R = 2$, $\theta_S = 3$, and $\lambda = \frac{3}{2} = 1.5$, i.e., the sets in relation S are 50% larger than the sets of R.

θ_R, θ_S	Set cardinalities in R and S
λ	Ratio of set cardinalities, $\lambda = \frac{\theta_S}{\theta_R}$
k	Number of partitions
l	Number of boolean hash functions used in LSJ and DCJ ($l = \log_2 k$)
$\|R\|, \|S\|$	Relation cardinalities
ρ	Ratio of relation cardinalities, $\rho = \frac{\|S\|}{\|R\|}$

Table 6. Variables used for analyzing the algorithms

The comparison and replication factors derived using the above assumptions are listed in Table 7. The table also shows the results obtained for the Lattice Set Join (LSJ) algorithm (an extension of the algorithm presented in [HM97]), which we analyze in the extended report [MGM01]. LSJ utilizes boolean hash functions like those used in DCJ, and yields the same comparison factor. In LSJ, each set $r \in R$ is assigned to a partition whose index is obtained as a boolean number $h_1(r)h_2(r)\cdots h_l(r)$. Each set $s \in S$ is assigned to partition $h_1(s)h_2(s)\cdots h_l(s)$ and, additionally, to each partition whose index is bitwise included in $h_1(s)h_2(s)\cdots h_l(s)$. Thus, the partitions generated by LSJ logically form a power lattice.

Note that in our model the selectivity of the join $R \bowtie S$ can be varied using the parameters θ_R, θ_S, and $|\mathcal{D}|$. As we show in [MGM01], the expected selectivity is $\frac{\theta_S!(|\mathcal{D}|-\theta_R)!}{(\theta_S-\theta_R)!|\mathcal{D}|!}$. For instance, for $\theta_R = 2$, $\theta_S = 3$, and $|\mathcal{D}| = 10$, we obtain the selectivity of $\frac{3!(10-2)!}{(3-2)!10!} \approx 0.066$. That is, the expected number of joining tuples for relations R and S having 4 tuples each (like those in Table 1) is $0.066 \cdot 4^2 \approx 1$. If $\mathcal{D}$ is large, the selectivity is almost zero. For example, for $|\mathcal{D}| = 1000$, $\theta_R = 10$ and $\theta_S = 20$, the selectivity is below 10^{-18}, i.e., a join between R and S with a billion tuples each is expected to return just one tuple.

Optimal hash functions Recall that DCJ partitions the relations R and S using l monotone boolean hash functions $h_1, \ldots, h_l$ into $k = 2^l$ partitions. In [MGM01] we derive the *optimal* firing probability for these hash functions that minimizes the comparison factor $comp_{\text{DCJ}}$ (and $comp_{\text{LSJ}}$). There are several ways of constructing such optimal functions, i.e., those that fire independently of each other with a fixed optimal probability. One simple approach is the following. Consider

Algorithm	Comparison and replication factors
PSJ	$comp_{\mathrm{PSJ}} = 1 - \left(1 - \frac{1}{k}\right)^{\theta_S}$ $repl_{\mathrm{PSJ}} = \frac{1}{1+\rho} + \frac{\rho}{1+\rho} k (1 - (1 - \frac{1}{k})^{\theta_S})$
DCJ	$comp_{\mathrm{DCJ}} = \left(1 - \frac{1}{1+\lambda}\left(\frac{\lambda}{1+\lambda}\right)^{\lambda}\right)^{\log_2 k}$ $repl_{\mathrm{DCJ}} = (1\ 1) \cdot \left[\frac{1}{1+\rho}\begin{pmatrix} \frac{1}{1+\lambda} & 1 \\ \frac{\lambda}{1+\lambda} & \frac{\lambda}{1+\lambda} \end{pmatrix}^{\log_2 k} + \frac{\rho}{1+\rho}\begin{pmatrix} 1 - \left(\frac{\lambda}{1+\lambda}\right)^{\lambda} & 1 - \left(\frac{\lambda}{1+\lambda}\right)^{\lambda} \\ 1 & \left(\frac{\lambda}{1+\lambda}\right)^{\lambda} \end{pmatrix}^{\log_2 k}\right] \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix}$
LSJ	$comp_{\mathrm{LSJ}} = comp_{\mathrm{DCJ}}$ $repl_{\mathrm{LSJ}} = \frac{1}{1+\rho} + \frac{\rho}{1+\rho} \sum_{t=0}^{\log_2 k} 2^i C_t^{\log_2 k} \left(1 - \left(\frac{\lambda}{1+\lambda}\right)^{\lambda}\right)^t \left(\frac{\lambda}{1+\lambda}\right)^{\lambda(\log_2 k - t)}$

Table 7. Summary of replication and comparison factors for PSJ, LSJ, and DCJ

that for each given set s of fixed cardinality $|s|$ we compute a bit string[2] of length b. For each element $x \in s$, we set a bit in the bit string at position ($x \mod b$). If the set elements are drawn uniformly from a large domain, the probability of each bit to be one is $1-(1-\frac{1}{b})^{|s|}$. Let function h_i fire whenever bit i is set in the bit string. Thus, we obtain b functions $h_1, \ldots, h_b$ that fire with equal probability $P(h_i(s)) = 1 - (1 - \frac{1}{b})^{|s|}$. For $b = 1$, we get just one function that fires with the probability of 1. For each larger b, we obtain b functions that fire with smaller probabilities. For example, for $b = 200$ and $|s| = 100$ we obtain 200 functions that fire with a probability of $1 - (1 - \frac{1}{200})^{100} \approx 0.4$.

By varying b, we can approximate any given probability between zero and one. If the domain $\mathcal{D}$ from which the set elements are drawn is much larger than b (e.g. b is less than 5% of $|\mathcal{D}|$), all b functions obtained for a given b fire (roughly) independently from each other. Once the b functions have been obtained, we can select l of them for use in DCJ. Notice that the bit-string approach for computing the boolean hash functions described above gives us a sufficient number of functions to use for partitioning. In [MGM01] we show that the value b that we have to use to obtain the optimal firing probability of the hash functions is $b = \frac{1}{1-(\frac{\lambda}{1+\lambda})^{\frac{1}{\theta_R}}}$. For example, for $\theta_R = 50$, $\theta_S = 100$ we get $b \approx 124$. Thus, we could use up to $l = 124$ hash functions, i.e., up to $k = 2^{124}$ partitions if needed. In [MGM01], we present an algorithmic specification of the approach described above. Also, we investigate an alternative technique based on disjoint sets of prime numbers as illustrated in Table 3.

[2] We use the term bit string instead of signature to avoid ambiguity. Although the bit strings are computed in the same way as signatures, they are not related to the signatures stored in partitions in any way.

4 Qualitative comparison of the algorithms

In this section we examine what the formulas of Table 7 tell us. We also discuss the accuracy of the predictions of our formulas.

Comparison factor First, we illustrate the reduction of the comparison factor with the growing number of partitions. All comparison factors in Table 7 are determined by the parameters θ_R, θ_S, and k. In Figure 4, we depict $comp_{\text{DCJ}}$ and $comp_{\text{PSJ}}$ for three containment join problems that correspond to the set cardinalities $\theta_R = \theta_S = 10$, $\theta_R = \theta_S = 100$, and $\theta_R = \theta_S = 1000$. Since $comp_{\text{LSJ}}$ is equivalent to $comp_{\text{DCJ}}$, we will not consider $comp_{\text{LSJ}}$ separately any further. Since DCJ dependends on the ratio λ of set cardinalities only, and $\lambda = 1$ in all three cases, the three curves for DCJ fall into one, depicted as a thick solid line. As can be seen in the figure, both $comp_{\text{DCJ}}$ and $comp_{\text{PSJ}}$ decrease steadily with growing k. However, the benefit of PSJ diminishes for large set cardinalities. For example, for $k = 128$ and $\theta_R = \theta_S = 1000$, PSJ requires around 7.5 times more comparisons (with $comp_{\text{PSJ}} \approx 1$) than DCJ (with $comp_{\text{DCJ}} \approx 0.13$). On the other hand, for small sets like $\theta_R = \theta_S = 10$, PSJ outperforms DCJ in the number of comparisons starting with $k \approx 40$. As a matter of fact, as k grows, PSJ eventually catches up with DCJ even for large set cardinalities. For example, for $\theta_R = \theta_S = 1000$ the breakeven point at which $comp_{\text{PSJ}} = comp_{\text{DCJ}}$ lies at $k \approx 135000$. However, as we explain below, replication overhead increases with k, limiting the maximal number of partitions that we can use effectively for computing the join.

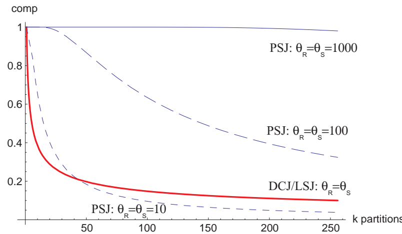

Fig. 4. Comparison factor vs. k

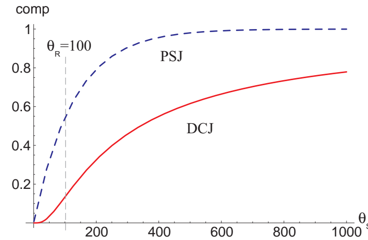

Fig. 5. Comparison factor vs. θ_S ($k = 128$)

Figure 5 demonstrates how the comparison factor increases with the growing cardinality of sets in relation S. We fix the set cardinalities in R at $\theta_R = 100$ and vary the set cardinalities[3] in S from $\theta_S = 10$ to $\theta_S = 1000$ for a constant number of partitions $k = 128$. Note that varying θ_S corresponds to varying λ from 0.1 to 10. As illustrated in Figure 5, $comp_{\text{DCJ}}$ remains below $comp_{\text{PSJ}}$ as the cardinality ratio grows (although not shown in the figure, $comp_{\text{DCJ}} < comp_{\text{PSJ}}$

[3] When $\theta_S < \theta_R = 100$, then the result of the join is known to be empty.

holds for all $\theta_S > 1000$). Moreover, in all scenarios, even those in which initially $comp_{\text{DCJ}} > comp_{\text{PSJ}}$, DCJ will eventually catch up and outperform PSJ as θ_S increases[4]. For example, starting with $\theta_R = \theta_S = 10$, and $k = 64$, we obtain $0.18 \approx comp_{\text{DCJ}} > comp_{\text{PSJ}} \approx 0.15$. Still, as θ_S grows, DCJ catches up with PSJ at $\theta_S \approx 110$, resulting in a comparison factor of 0.82.

Replication factor We examine the replication factor for the same settings as we utilized in the discussion of the comparison factor. Note that the replication factor depends on the ratio ρ of the relation sizes. Due to space limitations, we focus only on the case where $|R| = |S|$, i.e., $\rho = 1$. Figure 6 shows the growth of the replication factors $repl_{\text{LSJ}}$, $repl_{\text{DCJ}}$, and $repl_{\text{PSJ}}$ with the increasing number of partitions for the cases $\theta_S = \theta_R = 10$, $\theta_S = \theta_R = 100$, and $\theta_S = \theta_R = 1000$. Factors $repl_{\text{LSJ}}$ and $repl_{\text{DCJ}}$ depend only on the ratio of the set cardinalities; thus we obtain just one curve for LSJ and another one for DCJ. Notice that $repl_{\text{DCJ}}$ grows much slower with k than $repl_{\text{LSJ}}$. Moreover, $repl_{\text{DCJ}}$ outperforms $repl_{\text{PSJ}}$ even for $\theta_R = \theta_S = 10$. For large sets, like $\theta_R = \theta_S = 1000$, and $k = 128$, PSJ needs to write out $64.5 \cdot (|R| + |S|)$ signatures as partition data. This is 16.7 times more data to be stored temporarily than that generated by DCJ. Notice, however, that $repl_{\text{PSJ}}$ is bound by $\frac{1}{1+\rho} + \frac{\rho}{1+\rho} \cdot \theta_S$ (to see this, note that $\lim_{k\to\infty} k(1 - (1 - \frac{1}{k})^{\theta_S}) = \theta_S$). In contrast, $repl_{\text{LSJ}}$ and $repl_{\text{DCJ}}$ are unbound with growing k. This observation suggests that for any given θ_R and θ_S, there is a breakeven k, starting from which $repl_{\text{PSJ}}$ becomes smaller than $repl_{\text{DCJ}}$. For large sets, such k may be so enormous that the fact that PSJ is bound and DCJ/LSJ are not is practically irrelevant. For example, for $\theta_R = \theta_S = 1000$, $repl_{\text{DCJ}}$ becomes as large as the maximal value of $repl_{\text{PSJ}}$ ($0.5 + 500 = 500.5$), when $k \approx 2^{36}$.

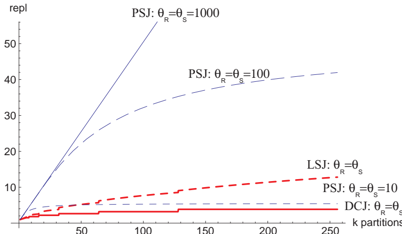

Fig. 6. Replication factor vs. k

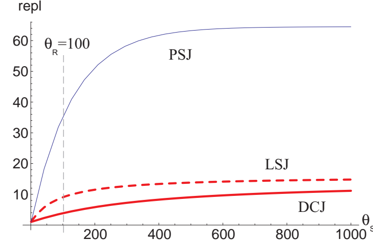

Fig. 7. Replication factor vs. θ_S ($k = 128$)

The impact of the set cardinality ratio on the replication factor is demonstrated in Figure 7. Again, we fix $k = 128$, $\theta_R = 100$, and vary θ_S from 10 to 1000. Correspondingly, λ ranges from 0.1 to 10. Notice that $repl_{\text{DCJ}}$ approaches

[4] This fact can be derived from formulas in Table 7.

$repl_{LSJ}$ with increasing λ, but never catches up[4]. Hence, DCJ always outperforms LSJ, and therefore we will focus our subsequent discussion on the superior algorithm DCJ.

The qualitative analysis in this section suggests that for each of the partitioning algorithms the comparison factor (and thus CPU load) decreases with growing k, whereas the replication factor (and thus I/O overhead) increases. Consequently, there is an *optimal* number of partitions k that minimizes the overall running time for each of the algorithms. Furthermore, our analysis indicates that PSJ is the algorithm of choice for small set cardinalities, while DCJ starts outperforming PSJ when the set cardinalities increase. In Section 5, we present the experimental results that substantiate these observations.

Accuracy of analytical model To study the accuracy of our formulas in realistic scenarios, we used five different distributions of element values, and five distributions of set cardinalities [MGM01]. Using simulations, we examined both the individual impact of varying just the element distribution or just the set cardinality distributions, as well as the combined effect. In summary, we found that for a variety of scenarios, the formulas of Table 7 deliver relatively accurate predictions that lie within 15% of the actual values. Across all experiments we observed that DCJ tends to be more negatively affected by varying the distributions than PSJ.

5 Experiments

We implemented the set containment join operator in Java using the Berkeley DB as the underlying storage manager. In our implementation, each tuple of the input relations R and S consists of a tuple identifier, a set of integers stored as a variable-size ordered list, and a fixed-size payload. The payload represents other attributes of the relations. In the experiments described below we used a payload of 100 bytes. The relations are stored as B-trees with the tuple identifiers serving as keys. To provide a fair evaluation of different partitioning algorithms, we implemented the set containment join operator is such a way that just the actual partitioning algorithm can be exchanged, other conditions remaining equal. In [MGM01] we document the Java implementations of each of the algorithms LSJ, DCJ, and PSJ as deployed in our testbed.

The partitions of both relations are stored in two B-trees[5], one B-tree per relation. We are not storing the partitions in plain files to exploit the buffering mechanisms of the storage manager. Each partition contains a list of pairs (set signature, tuple ID). In an initial implementation, we kept the data of each partition in a single B-tree record. However, we observed that the time required for appending data to the partitions was increasing significantly with growing partition sizes[6]. It proved much more efficient to split each partition into por-

[5] Since the number of partitions is relatively small, the overhead of maintaining a B-tree is negligible.

[6] Berkeley DB supports appending data to variable-size records incrementally, and does not require reading the records into memory first.

tions of equal sizes, while still keeping the partition in a single B-tree, and to use the combination of the portion number and partition index as the key of the B-tree. In the joining phase, the portions of partitions are read in batches to avoid random I/O. Splitting the partitions has an additional advantage in that it allows us to use large partitions that do not fit into the memory available for the join computation. After comparing all signatures in two partition batches, the identifiers of potentially joining tuples of R and S are sorted, and the corresponding tuples are fetched from disk.

Case study The experiments described below were performed on a 600 MHz Pentium III laptop running Linux. A total of 30 MB of memory was available to the Java Virtual Machine. Additionally, 10 MB was used by the Berkeley DB. To minimize the file buffering done by the operating system, we restricted the total available OS memory to 74 MB at boot time.

Figure 8 shows the impact of the partition number k on the execution time of the DCJ join for $|R| = |S| = 10000$, $\theta_R = 50$, and $\theta_S = 100$ (using a uniformly distributed element domain of size 10000 and uniformly distributed set cardinalities $45 \ldots 55$ in R and $90 \ldots 110$ in S). Each data point was obtained as an average of five runs using a 'cold' cache. We observe that there is an optimal value of $k = 32$ that yields the best execution time (24 sec). For $k > 32$, the growing partitioning overhead overweighs the reduction of the number of comparisons. Reading the partitions from the disk constitutes a part of the joining phase, therefore we see increasing joining time. Figure 9 illustrates the application of PSJ for the same setting. Notice that increasing the number of partitions does not help PSJ to reduce the execution time. In fact, using PSJ reduces the number of comparisons noticeably only starting from $k \approx 32$ ($comp_{\text{PSJ}} = 0.95$). At this point, however, PSJ becomes dominated by the partitioning I/O, and additional reduction of comparisons does not improve the execution time. Thus, in this scenario, PSJ proves ineffective with the best execution time of 48 sec.

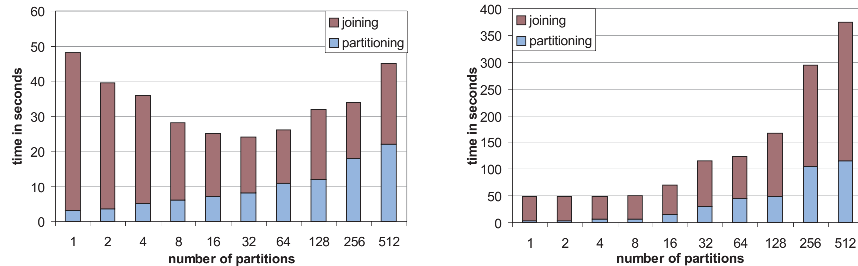

Fig. 8. Optimal partition number for DCJ ($|R| = |S| = 10000$, $\theta_R = 50$, $\theta_S = 100$)

Fig. 9. PSJ is I/O-bound and ineffective ($|R| = |S| = 10000$, $\theta_R = 50$, $\theta_S = 100$)

The above results emphasize that finding an optimal number of partitions is crucial for deploying DCJ and PSJ effectively. In a real system, we cannot afford running the algorithms with different partition numbers to determine the optimal setting. Fortunately, the analytical model that we developed in Section 3 helps us predict the best operational values for the algorithms. Using these values we can estimate the best execution time of each algorithm and choose the best performing one. For predicting the optimal number of partitions and the corresponding best execution time of either algorithm, we use a two-step approach. In the first step, for two given relations R and S, we estimate the comparison and replication factors using the formulas of Table 7. The estimate obtained in this first step depends only on the content of relations R and S, and the partitioning algorithm; it does not depend on hardware or the testbed we are using. In the second step, we predict the execution time knowing the estimated comparison and replication factors. This prediction is clearly system-dependent. However, as we will demonstrate, it can be applied for both partitioning algorithms used on the same system. Finally, for each partitioning algorithm we calculate the optimal number of partitions k that minimizes $time_{\text{DCJ}}$ or $time_{\text{PSJ}}$, and determine which algorithm to use for the given relations R and S.

Predicting execution times We approximate the running time of either algorithm using a function $time(x, y, k)$, where $x = comp \cdot |R| \cdot |S|$ is the total number of comparisons, $y = repl \cdot (|R| + |S|)$ is the total number of signatures to be stored temporarily, and k is the number of partitions. Notice that the join selectivity and the signature size are not included in this function. To choose the parameters for *time*, we build upon the detailed experimental results obtained for PSJ by Ramasamy et al. As reported in [RPNK00], with the growing number of partitions k, fragmentation becomes a significant factor, which we need to take into account. In contrast, the authors demonstrate that the exact choice of the signature size is less critical, as long as the signatures are large enough so that none or very few false positives are produced. Hence, in the experiments below, as well as in Figures 8 and 9, we choose a fixed signature size of 160 bits. To limit the complexity of the study, we are making an additional simplifying assumption that the join selectivity is small, i.e., at most a few tuples are returned as a result. In fact, both algorithms spend a comparable additional amount of time on reading out the result from the relations R and S. This additional time does not need to be considered in the comparison of the algorithms.

In [MGM01] we found that the function $time(x, y, k) = c_1 \cdot x + c_2 \cdot y \cdot k^{c_3}$ results in the smallest average prediction error compared to many other functions. The first part of the equation, $c_1 \cdot x$, represents the CPU time required for signature comparisons. The second part, $c_2 \cdot y \cdot k^{c_3}$, represents the I/O time for writing and reading the partitions, while k^{c_3} reflects the negative fragmentation effect that kicks in with growing k. For a given hardware configuration, the parameters c_1, c_2 and c_3 are obtained by applying the least-squares curve fitting method on data points collected for a variety of synthetic input relations. For each pair of synthetic relations, we run PSJ and DCJ with distinct values of k, and record for each run the overall execution time, the number x of comparisons done, and

the number y of signatures stored on disk temporarily. We refer to this step as 'calibration' of hardware. For our testbed implementation and the hardware settings described above, we obtained the equation $time(x, y, k) = 5.12686 \cdot 10^{-7} \cdot x + 8.28197 \cdot 10^{-7} \cdot y \cdot k^{0.691485}$ using 114 data points. The equation returns time in seconds and, applied to the data points that we used, yields an average prediction error of 15.4%.

Choosing the best algorithm Given the time equation, the decision between DCJ and PSJ for two input relations R and S can be made using the following steps:

1. Determine the actual sizes of the relations.
2. Determine the average set cardinalities θ_R and θ_S using sampling or available statistics.
3. Estimate the comparison and replication factors for DCJ and PSJ using the formulas of Table 7 for a number of different values of k, for example for $k = 2^l$, $1 \leq l \leq 13$.
4. Apply the time equation to determine the best execution times of both algorithms for the above values of k using the estimated comparison and replication factors[7].
5. Find the best execution time and pick the algorithm that produced it along with the optimal partition number k.

In addition to using our time equation at run time to select k and the algorithm to run, we can use the equation to understand in what cases DCJ or PSJ perform best. Figure 10 summarizes the experimental analysis of both algorithms. Both graphs in the figure connect breakeven points like the point $\theta_R = 50$, $\theta_S = 100$, $|R| = |S| = 128000$, at which both algorithms yield equal execution times. Each graph divides the first quadrant into two areas, the one where PSJ wins over DCJ (below the graph), and the one in which DCJ outperforms PSJ (above the graph). The solid line corresponds to the cardinality ratio $\lambda = 1$, whereas the dotted line illustrates $\lambda = 2$. Figure 10 allows us to estimate what algorithm to deploy for the given input relations. For example, given $\theta_R = \theta_S = 50$ and $|R| = |S| = 100000$, the figure indicates that DCJ is clearly the algorithm of choice. On the other hand, if $\theta_R = \theta_S = 10$, we should go for PSJ. Please keep in mind that the graphs shown in the figure may have different shapes for other systems.

One final remark is due here; recall that DCJ (and LSJ) can make effective use of k partitions only if k is a power of two. Hence, DCJ is less flexible in choosing the partition number k. However, our experiments suggest that in practice this inflexibility is not critical. For example, in Figure 8 we can see that the execution times of DCJ are roughly similar for $k = 16$, $k = 32$, and $k = 64$. In other words, the inability to choose say $k = 48$ does not cripple the performance of DCJ. Furthermore, as we explain in [MGM01], the limitation in choosing k can be addressed using the modulo approach suggested in [HM97].

[7] Since the formulas in Table 7 are fairly complex, determining the optimal k analytically is hard. Therefore, we use the probing approach.

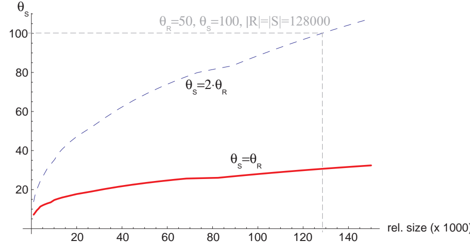

Fig. 10. When use DCJ instead of PSJ? Each graph separates the space of input relations into area where DCJ wins (area above graph) and where PSJ wins (area below graph)

6 Related Work

The set containment join and other join operators for sets enjoyed significant attention in the area of data modeling. However, very little work deals with efficient implementations of these operators. Helmer and Moerkotte [HM97] were the first to directly address the implementation of set containment joins. They investigated several main memory algorithms including different flavors of nested-loop joins, and suggested the Signature-Hash Join (SHJ) as a best alternative. Later, Ramasamy et al [RPNK00] developed the Partitioning Set Join (PSJ), which does not require all data to fit into main memory. They showed that PSJ performs significantly better than the SQL-based approaches for computing the containment joins using unnested representation. Prior to [HM97] and [RPNK00], the related work focused on signature files, which had been suggested for efficient text retrieval two decades ago. A detailed study of signature files is provided by Faloutsos and Christodoulakis in [FC84]. Ishikawa et al [IKO93] applied the signature file technique for finding subsets or supersets that match a fixed given query set in object-oriented databases. The inherent theoretical complexity of computing set containment joins was addressed in [CCKN01, HKP97]. Partitioning has been utilized for computing joins over other types of non-atomic data, e.g., for spatial joins [PD96]. Index-based approaches for accessing multidimensional data were studied e.g. in [BK00].

In [MGM01] we introduce and study in detail the Lattice Set Join (LSJ), a partitioning algorithm that extends the main-memory algorithm SHJ [HM97] and does not require all data to fit into main memory. We also present an alternative approach to computing the boolean hash function used in LSJ and DCJ, based on disjoint sets of primes.

Many other options for implementing partitioning-based algorithms for atomic attributes have been discussed in the database literature. For example, keeping a fixed number of partitions permanently in main memory improves the execution time when much memory is available. Similarly, separating the joining phase and the verification phase by first writing out potentially joining tuple identifiers of all partitions to disk may improve performance. Due to space

limitations, we do not discuss these implementation options. For generating synthetic databases used in our experiments, we deployed the methods described in [GEBW94].

7 Conclusion

In this paper we suggested a novel algorithm called the Divide-and-Conquer Set Join for computing the set containment joins. We compared the performance of DCJ with that of PSJ [RPNK00] and LSJ [MGM01]. We developed a detailed analytical model that allowed us to study the join algorithms qualitatively, and to tune them for different input relations. Furthermore, we explored the behavior of the algorithms experimentally using an implemented testbed. We found that DCJ always outperforms LSJ in terms of the replication factor. In contrast, PSJ and DCJ provide complementary approaches for computing set containment joins. Specifically, when the set cardinalities are large, DCJ introduces a significant performance improvement as compared to PSJ. On the other hand, PSJ wins over DCJ when small sets are used. The work presented in [HM97], [RPNK00], and in this paper raises the question whether even better algorithms for set containment joins exist. Currently, we are trying to develop a hybrid algorithm that combines the strengths of PSJ and DCJ. Developing efficient algorithms for other set join operators, for instance the intersection join, is another challenging and mostly unexplored research direction.

References

[BK00] C. Böhm and H.-P. Kriegel. Dynamically Optimizing High-Dimensional Index Structures. In *Proc. EDBT'00*, 2000.

[CCKN01] J.-Y. Cai, V. T. Chakaravarthy, R. Kaushik, and J.F. Naughton. On the complexity of join predicates. In *Proc. PODS'01*, 2001.

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

[GEBW94] J. Gray, S. Englert, K. Baclawski, and P.J. Weinberger. Quickly generating billion-record synthetic databases. In *Proc. SIGMOD'94*, 1994.

[HKP97] J. Hellerstein, E. Koutsoupias, and C. H. Papadimitriou. On the analysis of indexing schemes. In *Proc. PODS'97*, 1997.

[HM97] S. Helmer and G. Moerkotte. Evaluation of main memory join algorithms for joins with set comparison join predicates. In *Proc. VLDB'97*, 1997.

[IKO93] Y. Ishikawa, H. Kitagawa, and N. Ohbo. Evaluation of signature files as set access facilities in OODBS. In *Proc. SIGMOD'93*, 1993.

[MGM01] S. Melnik and H. Garcia-Molina. Divide-and-Conquer Algorithm for Computing Set Containment Joins. Extended Technical Report, `http://dbpubs.stanford.edu/pub/2001-32`, September 2001.

[PD96] J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. In *Proc. SIGMOD'96*, 1996.

[RPNK00] K. Ramasamy, J. M. Patel, J. F. Naughton, and R. Kaushik. Set Containment Joins: the Good, the Bad and the Ugly. In *Proc. VLDB'00*, 2000.