Adaptive Algorithms for Set Containment Joins

SERGEY MELNIK
and
HECTOR GARCIA-MOLINA
Stanford University, USA

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 deployed in many database applications, even those that do not support set-valued attributes. In this paper, we propose two novel partitioning algorithms, called the Adaptive Pick-and-Sweep Join (APSJ) and the Adaptive Divide-and-Conquer Join (ADCJ), which allow computing set containment joins efficiently. We show that APSJ outperforms previously suggested algorithms for many data sets, often by an order of magnitude. We present a detailed analysis of the algorithms and study their performance on real and synthetic data using an implemented testbed.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*query processing*

General Terms: Algorithms, Experimentation, Performance

1. INTRODUCTION

Set containment queries are utilized in many database applications, especially when the underlying database systems support set-valued attributes. For example, consider a database application used by a human-resource broker company to match the skills of job seekers with the skills required by employers. Imagine that the set of skills needed for filling an open position is stored in the set-valued attribute {`reqSkills`} of table `Jobs(jobID, {reqSkills})`. Another table, `Jobseekers(personID, {availSkills})`, keeps a set of skills of potential recruits. Then, a match between the qualifying job seekers and the jobs can be computed using a set containment query `SELECT Jobseekers.personID, Jobs.jobID WHERE`

Authors' addresses: Department of Computer Science, Stanford University, Gates Hall 4A, Stanford, CA 94305-9040 USA; email: {melnik,hector}@db.stanford.edu. S. Melnik's permanent address: Department of Computer Science, University of Leipzig, Augustusplatz 10-11, 04109 Leipzig, Germany.
This is a preliminary release of an article accepted by ACM Transactions on Database Systems. The definitive version is currently in production at ACM and, when released, will supersede this version.

`Jobs.{reqSkills}` $\subseteq$ `Jobseekers.{availSkills}`. 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 database application that recommends to students a list of courses that they are eligible to take. Such recommendation can be computed using a set containment join on prerequisite courses and courses already taken by students.

Notice that containment queries can be utilized even in database systems that support only atomic attribute values. Consider a relational database of a stock broker company. Imagine that the investment portfolios of the customers are kept in a relational table `Portfolios(portfolioID, stockID, numPositions)`, while the information about the composition of the mutual funds is kept in a table `Funds(fundID, stockID, percentage)`. Assume that the stock broker wants to find the portfolios that mirror mutual funds, i.e. those that contain just a portion of the stocks from a certain mutual fund, and no other stocks. The query

```
SELECT P.portfolioID, F.fundID FROM Portfolios P, Funds F
WHERE P.stockID = F.stockID
      GROUP BY P.portfolioID, F.fundID
      HAVING COUNT(*) = (SELECT COUNT(*) FROM Portfolios P2
                         WHERE P2.portfolioID = P.portfolioID)
```

joins `Portfolios` and `Funds` on the `stockID` attribute and returns those portfolios whose stock positions are entirely contained in a fund, together with the `fundID`. To see how this query works notice that the number of the joining tuples for a given `P.portfolioID` and `F.fundID` must be equal to the total number of stocks in portfolio `P.portfolioID`. The above query corresponds to a set containment query, although expressed in a less obvious way. Additional types of applications for containment joins 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.

The two best known algorithms for computing set containment joins efficiently are the Partitioning Set Join (PSJ) proposed in [Ramasamy et al. 2000] and the Divide-and-Conquer Join (DCJ) that we suggested in [Melnik and Garcia-Molina 2002]. PSJ and DCJ introduce crucial performance gains compared with straightforward approaches. A major limitation of PSJ is that it quickly becomes ineffective as set cardinalities grow. In contrast, DCJ depends only on the ratio of set cardinalities in both relations, and, therefore, wins over PSJ when the sets are large. Often, 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. PSJ is ineffective for such data sets.

The contribution of this paper are two novel algorithms called the Adaptive Pick-and-Sweep Join (APSJ) and the Adaptive Divide-and-Conquer Join (ADCJ), which

Relation R
$a = \{2, 9\}$
$b = \{8, 18\}$
$c = \{1, 3\}$

Relation S
$A = \{2, 4, 9\}$
$B = \{3, 8, 18\}$
$C = \{1, 3, 4\}$
$D = \{3, 4, 7\}$

Table I. Two sample relations with set-valued attributes

extend and improve on the best known algorithms PSJ and DCJ. We show that ADCJ always outperforms DCJ, especially when the relations to be joined have different sizes. APSJ overcomes the main limitation of PSJ, namely, its inability to deal with large sets effectively (like DCJ, APSJ depends only on the ratio of set cardinalities in both relations). Moreover, it turns out that in most scenarios APSJ is the top performer overall.

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 algorithms that we developed using a simple example. Section 3 deals with the theoretical analysis of the algorithms. After that, in Section 4, we provide a qualitative comparison of the algorithms. Section 5 describes the testbed that we implemented. In Section 6 we examine the performance of the algorithms using our testbed. Then, in Section 7, we address the issue of choosing the best performing algorithm. The performance trends are analyzed in Section 8. Finally, we discuss related work in Section 9 and conclude the paper in Section 10.

2. ALGORITHMS

In this section, we explain the algorithms that we developed using a simple example. We start with a brief overview of set containment joins, signatures and partitioning. As a first algorithm, we describe the Partitioning Set Join (PSJ) algorithm [Ramasamy et al. 2000]. A reader familiar with set containment joins and PSJ may skip ahead to Section 2.3 where we start describing our new algorithms.

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 I. Each of the relations contains one column with sets of integers, three sets in R and four in S. For easy reference, the sets of R and S are labeled using letters a, b, c and A, B, C, D, respectively. Computing the containment join $R \bowtie_{\subseteq} S$ amounts to finding all tuples $(r, s) \in R \times S$ such that $r \subseteq s$. In our example, $R \bowtie_{\subseteq} S = \{(a, A), (b, B), (c, C)\}$.

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 $|R| \cdot |S| = 3 \cdot 4 = 12$ set comparisons. For large relations R and S, doing $|R| \cdot |S|$ 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 [Helmer and Moerkotte 1997] and partitioning [Ra-

$x \in R$	$sig(x)$
a	0110
b	1010
c	0101

$y \in S$	$sig(y)$
A	1110
B	1011
C	1101
D	1001

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

masamy et al. 2000]. 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. To be useful for set containment joins, the signatures need to preserve the partial order on sets induced by the subset predicate. To illustrate, consider the example in Table II. 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 $b = \{8, 18\}$ we set bit 0 ($8 \mod 4$) and bit 2 ($18 \mod 4$), and obtain $sig(\{8, 18\}) = 1010$.

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 $sig(b) \not\subseteq^b sig(C)$, we know that b cannot be a subset of C. 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 12 signature comparisons we only need to test 4 pairs of sets for containment: $(a, A), (b, A), (b, B)$, and (c, C). Of these remaining pairs, (b, A) is rejected as a false positive.

Using signatures helps to reduce the number of set comparisons significantly, yet still requires $|R| \cdot |S|$ 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 I. Let $\pi(a) = \pi(b) = \pi(A) = \pi(B) = \{1\}$, $\pi(c) = \pi(C) = \{2\}$, and $\pi(D) = \{1, 2\}$. That is, R is partitioned into $R_1 = \{a, b\}$, $R_2 = \{c\}$, 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 $R_2 \bowtie_{\subseteq} S_2 = \{c\} \bowtie_{\subseteq} \{C, D\}$ requires only $2 \cdot 3 + 1 \cdot 2 = 8$ signature comparisons. Hence, by using partitioning we reduced the total number of signature comparisons from 12 to 8. We refer to the fraction $\frac{8}{12}$ 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, $|\{a, b\}| + |\{c\}| = 3$ signatures from $R_{1,2}$ and $|\{A, B, D\}| + |\{C, D\}| = 5$

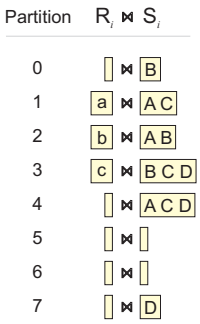

Fig. 1. Partitioning with PSJ: 7 comparisons, 15 replicated

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 $\frac{3+5}{3+4} = \frac{8}{7}$. 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. [2000]. To illustrate the algorithm, we continue with the example introduced above. Imagine that we want to partition R and S from Table I 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 $a = \{2, 9\} \in R$. Let 9 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 with index $(9 \mod 8) = 1$, i.e., to partition R_1. Element 18 chosen from $b = \{8, 18\}$ yields partition number $2 = (18 \mod 8)$. Finally, set c falls into partition R_3 based on randomly chosen element $3 \in c$. 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, $A = \{2, 4, 9\}$ is assigned to partitions S_2, S_4, and S_1, $B = \{3, 8, 18\}$ goes into partitions S_3, S_0, and S_2, etc. The complete partition assignment for R and S is summarized in Figure 1. Notice that PSJ requires that no R set be empty (a set with no elements cannot be assigned to any of the partitions without losing joining tuples).

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 signature of set c is read from R_3, and is compared with the signatures of sets B, C, and D stored in S_3. Hence, computing $R_3 \bowtie_{\subseteq} S_3$ results in $1 \cdot 3 = 3$ signature comparisons. The total number of signature comparisons required in our example amounts to $0+2+2+3+0+0+0+0 = 7$, whereas a total of 15 signatures need to be written out to disk. Thus, in this example, we obtain the comparison factor $\frac{7}{12} \approx 0.58$, and

$x \in R$	$h_1 h_2 h_3 h_4 h_5 h_6 h_7$
a	0 <u>1</u> 0 1 0 0 0
b	0 0 0 0 0 0 0
c	<u>1</u> 0 1 0 0 0 0

$y \in S$	$h_1 h_2 h_3 h_4 h_5 h_6 h_7$
A	0 1 0 1 0 0 0
B	0 0 1 0 0 0 0
C	1 0 1 1 0 0 0
D	0 0 1 1 0 0 1

Table III. Boolean hash functions used in APSJ example

replication factor $\frac{15}{3+4} \approx 2.14$.

2.3 Adaptive Pick-and-Sweep Join (APSJ)

The Adaptive Pick-and-Sweep Join (APSJ) generalizes and extends the PSJ algorithm. We illustrate APSJ using our running example of Table I and $k = 8$ partitions. Assume that there exist $k - 1 = 7$ *boolean hash functions* $h_1, \ldots, h_7$ that take a set of integers as input and return 0 or 1 as output. For example, consider the functions defined as $h_i(x) = 1 \iff \exists e \in x : (e \mod 9) = i$ for $i = 1, \ldots, 7$. The function with index i fires for set r if and only if r contains an element, which, taken modulo 9, yields i. Each of these functions is *monotone* in the sense that whenever h_i fires (i.e., returns 1) for a given set x, it is guaranteed to fire for each superset of x. For example, consider set $c = \{1, 3\}$. Since $(1 \mod 9) = 1$ and $(3 \mod 9) = 3$, we have $h_1(c) = h_3(c) = 1$. For set $C = \{1, 3, 4\}$, h_4 fires in addition to h_1 and h_3, since $(4 \mod 9) = 4$. Table III lists the values taken by all seven functions for the sets a, b, c and A, B, C, D. In general, APSJ can utilize any kind of monotone hash function, not just the modulo-based ones illustrated above.

Using these $k - 1 = 7$ functions, we partition our sample relations into $k = 8$ partitions as follows. For each set $r \in R$, we consider the indexes of the hash functions that fired, i.e., $\{j \mid h_j(r) = 1\}$. We randomly *pick* an index i from this set, and assign r to partition R_i. If the set is empty, we assign r to the 'default' partition R_0. For example, for set c we can choose between index 1 and 3, so say we select 1 and place c in R_1. (The selected indexes are underlined in Table III.) Set b is placed in R_0. Every set $s \in S$ is inserted into all partitions S_j with $h_j(s) = 1$, i.e., we *sweep* the indexes of all firing functions. Additionally, each s is assigned to the 'default' partition S_0. Thus, for example, set A is assigned to partitions S_2, S_4, and, additionally, to partition S_0. The complete partition assignment produced by APSJ for our sample relations is depicted in Figure 2.

Notice that because we use the default partitions R_0 and S_0, $k-1$ hash functions produce k partitions. The default partitions allow us to partition the relations correctly even if R contains empty sets, or, in general, sets for which none of h_i fires (recall that PSJ cannot deal with empty sets). In our example, the joining tuples b and B are found when the partitions R_0 and S_0 are read from disk. Overall, $4 + 1 + 1 + 0 + 0 + 0 + 0 + 0 = 6$ signature comparisons are needed, while the total of 16 signatures need to be stored on disk. Hence, we obtain the comparison factor $\frac{6}{12} = 0.5$ and replication factor $\frac{16}{3+4} \approx 2.14$.

In our tiny running example, APSJ wins over PSJ since it is lucky: had c been randomly assigned to bucket 4, APSJ would use more signature comparisons than PSJ. However, in real data, when the set cardinalities are large, PSJ tends to assign almost all sets of S to each of the S_i partitions, yielding many signature comparisons

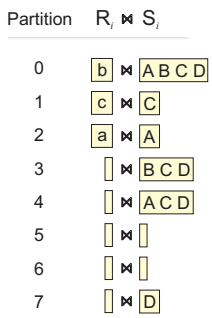

Fig. 2. Partitioning with APSJ: 6 comparisons, 16 replicated

and high replication. APSJ offers an extra 'tuning knob' that is not available in PSJ, namely the boolean hash functions. Because of this flexibility, APSJ can often be tuned to achieve better performance than PSJ. Notice that if all hash functions fire with very *high* probabilities, then each S_i will include most of S, so joins will be expensive. In contrast, if the functions fire with very *low* probabilities, then R_0 will contain most of R, and we will have to join $R_0 \bowtie S_0 = R \bowtie S$. Clearly, to minimize the work, we need to select a firing probability somewhere in the middle. In Section 3.1 we show how to construct the APSJ hash functions *adaptively* depending on the characteristics of the input relations.

Both algorithms PSJ and APSJ can be tuned by varying the number of partitions. The more partitions we use, the fewer comparisons are necessary. However, a larger number of partitions also causes more replication. (This tradeoff is common for all partitioning algorithms that we consider and will be illustrated in detail in Sections 4 and 6).

2.4 Adaptive Divide-and-Conquer Set Join (ADCJ)

The Adaptive Divide-and-Conquer Set Join (ADCJ) is based on the DCJ algorithm that we present in [Melnik and Garcia-Molina 2002]. Again, we illustrate the ADCJ algorithm using our running example of Table I and $k = 8$ partitions. We explain the algorithm using a series of partitioning steps depicted in Figure 3. In every step, one monotone 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 each pair of partitions $R_i \bowtie S_i$, as indicated by the labels 'α' and 'β' placed on the forks in Figure 3. Although we illustrate ADCJ conceptually as a branching tree, the final partition assignment is computed without using any intermediate partitions (see Appendix G). First, we explain the main idea of DCJ and then present the contribution of ADCJ, the adaptive design of the α, β-pattern.

The monotone boolean hash functions that we use in Figure 3 are defined as $h_i(x) = 1 \iff \exists e \in x : (e \mod 4) = i$, where $i = 1, 2, 3$. Notice that this definition is similar to the one used for APSJ, except that each element value is taken modulo 4 instead of 9. Table IV shows the values of functions h_1, h_2, h_3 for the sets from our sample relations. Since the number of partitions doubles in each step, only $\log_2 8 = 3$ steps and, therefore, only 3 boolean hash functions are

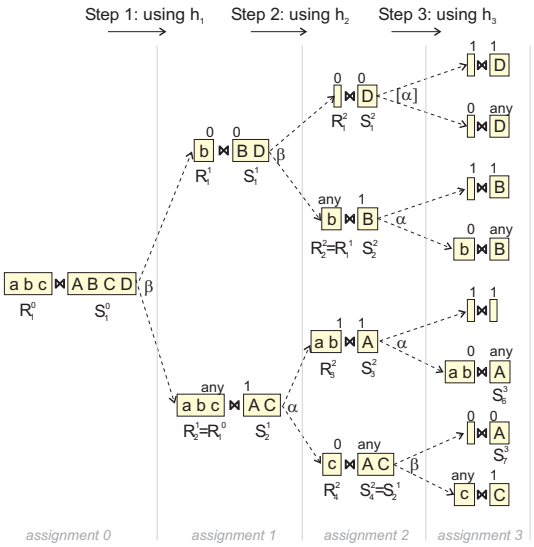

Fig. 3. Partitioning with ADCJ: 4 comparisons, 11 replicated

$x \in R$	$h_1h_2h_3$
a	1 1 0
b	0 1 0
c	1 0 1

$y \in S$	$h_1h_2h_3$
A	1 1 0
B	0 1 1
C	1 0 1
D	0 0 1

Table IV. Boolean hash functions used in ADCJ example

required to obtain $k = 8$ partitions. Just like APSJ, ADCJ works with any kind of hash functions, as long as they are monotone.

Relations $R = \{a, b, c\}$ 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 β and hash function h_1. Sets B and D with $h_1(B) = h_1(D) = 0$ are assigned to partition S_1^1, while the remaining sets A and C with $h_1(A) = h_1(C) = 1$ are inserted into S_1^2. We abbreviate this procedure concisely as $S_1^1 := S/\neg h_1$, $S_1^2 := S/h_1$. Since h_1 is monotone, each subset x of B or D must satisfy $h_1(x) = 0$. Therefore, partition $S_1^1 = \{B, D\}$ needs to be joined only with those sets in $R = \{a, b, c\}$ that satisfy $h_1(x) = 0$, i.e. just with set b. In contrast, each set of R may possibly be a subset of A or C. Thus, we obtain $R_1^1 := R/\neg h_1$ and $R_2^1 := R$ (the values 0 and 'any' taken by h_1 are depicted above R_1^1 and R_2^1). Notice that instead of $4 \cdot 3 = 12$ signature comparisons required for $R \bowtie S$, only $1 \cdot 2 + 3 \cdot 2 = 8$ signature comparisons would be needed for joining the partitions of assignment 1.

Given a pair of partitions $R_i \bowtie S_i$, operator β splits partition S_i and replicates partition R_i. In contrast, operator α splits R_i and replicates S_i. Figure 3 shows how operator α is used to repartition $R_2^1 \bowtie S_2^1 = \{a, b, c\} \bowtie \{A, C\}$. First, R_2^1 is split into $R_3^2 = R_2^1/h_2 = \{a, b\}$ and $R_4^2 = R_2^1/\neg h_2 = \{c\}$. Since each superset x

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

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

of a or b must satisfy $h_2(x) = 1$, R_3^2 needs to be joined only with those sets of $S_2^1 = \{A, C\}$ that satisfy $h_2(x) = 1$, i.e., just with the set A. Hence, S_3^2 is obtained as $S_3^2 = S_2^1/h_2$, whereas S_4^2 must contain all of $S_2^1 = \{A, C\}$. The definitions of operators α and β are presented in Table V.

Adaptive design of α,β-pattern. The operators α and β both perform correct repartitioning and thus can be applied interchangeably at each fork in the branching tree of Figure 3. Different patterns of applying α and β yield distinct partition sizes in the final assignment, so we can improve performance by selecting the operators judiciously. Optimal performance is achieved when the comparison and replication factors are minimal. As shown in Appendix C, the comparison factor is determined entirely by the firing probabilities of the hash functions, and is independent of the α,β-labeling of the tree. However, the choice of α,β-pattern is crucial for minimizing replication. The smallest replication factor is obtained if at each fork we always split the larger partition and replicate the smaller one. Otherwise, if a suboptimal choice is made, the replication factor of the subtree originating at that fork increases and makes the overall replication factor grow. Hence, if $|R_i| \geq |S_i|$, we should apply operator α, otherwise we should use β. For example, in Step 1, we have $|R| = 3 < 4 = |S|$. Therefore, operator β is best. If we computed the intermediate partitions, we would know their sizes and could apply the above rule. However, we do not generate the intermediate partitions, since storing them temporarily on disk is prohibitively expensive.

Suppose for now that we know the optimal α,β-pattern, i.e., the one that minimizes replication. Then, we can compute the partition assignment of each set of R or S by 'tracing' its way through the tree, with no need for intermediate, materialized partitions. In our example, set A belongs initially to $S_1^0 = S$. Given that the β is applied at the first fork, we compute $h(A)$ to decide whether A is sent to S_1^1 ('up') or S_2^1 ('down'). Since $h_1(A) = 1$, A is sent 'down'. At the next fork we send A both 'up' (S_2^2) and 'down' (S_4^2), based on $h_2(A) = 1$ and the use of operator α. Now the path of A splits, and we have to track both paths. After the final step, A is assigned to S_6^3 and S_7^3. In Appendix G we present a formal specification of the ADCJ algorithm that computes the partition assignment for each set based on the above technique.

Thus, our final challenge is to determine a 'good' α,β-pattern for the partitioning technique of the previous paragraph. We design the pattern adaptively based on the characteristics of the input relations. The key idea is to *estimate* the sizes of the intermediate partitions using the firing probabilities of the hash functions. Suppose that in our example we know that functions h_1, h_2, h_3 fire with probability of 0.5 for sets in R, and with probability 0.6 for sets in S. Consider partitions $R_2^1 \bowtie S_2^1$ obtained in Step 1 using function h_1. The expected size of partition $S_2^1 = S/h_1$ can be estimated as $|S_2^1| = 0.6 \cdot |S| = 0.6 \cdot 4 = 2.4$. Given that $|R_2^1| = |R| = 3 > 2.4 = |S_2^1|$, we select operator α for repartitioning $R_2^1 \bowtie S_2^1$. Assuming that $R_2^1 \bowtie S_2^1$ are

repartitioned using α, we can estimate the sizes of partitions R_3^2 and S_3^2. Since $R_3^2 = R_2^1/h_2 = R/h_2$, we get $|R_3^2| = 0.5 \cdot |R| = 0.5 \cdot 3 = 1.5$, while the expected size of S_3^2 is $0.6 \cdot |S_2^1| = 0.6^2 \cdot |S| = 1.44$. Because $|R_3^2| = 1.5 > 1.44 = |S_2^1|$, we choose operator α again to repartition $R_3^3 \bowtie S_3^2$. Of course, the actual partition sizes may deviate from the expected values, so we can choose a suboptimal operator. For example, the estimated size of partition R_1^2 is $|R_1^2| = |(R/\neg h_1)/\neg h_2| = (1-0.5)^2 \cdot |R| = 0.75$, whereas $|S_1^2| = (1-0.6)^2 \cdot |S| = 0.64$. Thus, we choose to apply operator α. However, as shown in Figure 3, in our example the actual sizes of R_1^2 and S_1^2 turn out to be 0 and 1, i.e., β would have been a better choice. In fact, choosing β would require one less signature to be stored to disk.

To summarize, our algorithm computes the partition assignment in three stages.

(1) First, we construct the hash functions that minimize the comparison factor (just like in DCJ).

(2) Second, we determine the α,β-tree that reduces replication using the firing probabilities of the hash functions.

(3) Finally, we compute the partition assignment by tracing each set of R and S through the α,β-tree.

In the final assignment produced in our example (Assignment 3), the total of $0+0+0+1+0+2+0+1 = 4$ signature comparisons are required, whereas 11 signatures need to be written out to disk (one more than absolutely necessary if we had used β for $R_1^2 \bowtie S_1^2$). Thus, we obtain comparison factor $\frac{4}{12} \approx 0.33$ and replication factor $\frac{11}{3+4} \approx 1.57$, close to the best possible replication factor of $\frac{10}{3+4} \approx 1.42$.

3. ANALYSIS OF THE ALGORITHMS

We start the discussion of the partitioning algorithms APSJ and ADCJ by presenting our analytical model. As an efficiency measure 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.

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

$\|R\|, \|S\|$	Relation cardinalities
ρ	Ratio of relation cardinalities, $\rho = \frac{\|S\|}{\|R\|}$
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 hash functions used in APSJ, $l = k - 1$

Table VI. Variables used for analyzing the algorithms

(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 6. 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 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 VI. For instance, for our sample relations in Table I we obtain $|R| = 3$, $|S| = 4$, $\rho = \frac{4}{3} \approx 1.33$, $\theta_R = 2$, $\theta_S = 3$, and $\lambda = \frac{3}{2} = 1.5$, i.e., the sets in relation S are 50% larger than the sets of R.

Note that in our model the selectivity[2] of the join $R \bowtie S$ can be varied using the parameters θ_R, θ_S, and $|\mathcal{D}|$. As we show in Appendix D, 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 $|\mathcal{D}| = 8$, we obtain the selectivity of $\frac{3!(8-2)!}{(3-2)!8!} \approx 0.11$. That is, the expected number of joining tuples for relations R and S having 3 and 4 tuples each (like those in Table I) is $0.11 \cdot 3 \cdot 4 \approx 1.3$. 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.

Boolean hash functions. Both in APSJ and ADCJ we use monotone boolean hash functions to partition the relations. In our analysis, we consider a subclass of monotone boolean hash functions with the following two properties. First, each of the functions h_i fires independently of the others. Second, the firing probability of each h_i for a set s is $P(h_i(s)) = 1 - p^{|s|}$, where $p \in [0, 1]$. In our testbed, we construct the functions with the above properties using a so-called bit-string technique, as illustrated in Section 2.3. That is, for each given set s of fixed cardinality $|s|$ we compute a bit string[3] of length b. For each element $x \in s$, we set a bit in the

[2]For two relations R and S, the selectivity of the join $R \bowtie S$ is the fraction of elements in the cross-product $R \times S$ that participate in the join.
[3]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.

bit string at position ($hash(x) \mod b$). (In general, we apply some simple hash function $hash$ to x before taking modulo to 'smooth out' the element domain.) 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 $P(h_i(s)) = 1-(1-\frac{1}{b})^{|s|} = 1-p^{|s|}$. 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.

In both APSJ and ADCJ we select a subset of the available b hash functions to do the partitioning. If the number l of the selected functions is much smaller than b, and b is much smaller than the size of the domain, i.e., $l \ll b \ll |\mathcal{D}|$, then the selected l functions fire (roughly) independently from each other. As we show in Appendix A, even if l is close to b, our analysis presented below remains accurate. In the appendix we also demonstrate that the bit-string technique produces enough functions to use in APSJ and ADCJ.

3.1 Analysis of APSJ

APSJ uses $k-1=l$ monotone boolean hash functions to partition relations R and S into k partitions. Each h_i fires with probability $P(h_i(r)) = 1-p^{\theta_R}$ for sets of R and with probability $P(h_i(s)) = 1-p^{\theta_S}$ for sets of S. Recall that each set r of relation R is assigned to exactly one of partitions $R_0, R_1, \ldots, R_l$ based on the index of a randomly chosen function h_i with $h_i(r)=1$. If none of $h_1,\ldots,h_l$ fires, r is assigned to R_0. Since h_i are independent, the probability of all of h_i to remain silent for a random $r \in R$ is $\prod_{i=1}^{l}(1-P(h_i(r))) = p^{\theta_R l}$. Hence, the expected number of signatures in partition R_0 is $|R| \cdot p^{\theta_R l}$. The rest of the signatures are distributed uniformly over partitions $R_1,\ldots,R_l$. In other words, each R_i contains on average $\frac{|R|-|R_0|}{l} = \frac{1}{l}(1-p^{\theta_R l})|R|$ signatures. Each set $s \in S$ is assigned to all S_i such that $h_i(s)=1$, and, additionally, to the 'default' partition S_0. That is, S_0 contains all of S, i.e., $|S_0| = |S|$. The probability of h_i to fire for a random set $s \in S$ is $P(h_i(s)) = 1-p^{\theta_S}$. Thus, each of $S_1,\ldots,S_l$ has on average $(1-p^{\theta_S})|S|$ signatures.

Now the comparison factor for APSJ can be computed as $comp_{\text{APSJ}} = \frac{\sum_{i=0}^{l}|R_i|\cdot|S_i|}{|R|\cdot|S|} = \frac{|R|\cdot p^{\theta_R l}\cdot|S| + l\cdot\frac{1}{l}(1-p^{\theta_R l})|R|\cdot(1-p^{\theta_S})|S|}{|R|\cdot|S|} = p^{\theta_R l} + (1-p^{\theta_R l})\cdot(1-p^{\theta_S}) = 1-p^{\theta_S}+p^{\theta_R l+\theta_S}$. The comparison factor is minimized[4] when $p = p_{opt} = \left(\frac{\theta_S}{\theta_S+\theta_R l}\right)^{\frac{1}{\theta_R l}}$. Substituting $\lambda = \frac{\theta_S}{\theta_R}$, we obtain $p_{opt} = \left(\frac{\lambda}{\lambda+l}\right)^{\frac{1}{\theta_R l}}$. Inserting p_{opt} in the formula for $comp_{\text{APSJ}}$ yields $comp_{\text{APSJ}} = 1-\frac{l}{l+\lambda}\cdot\left(\frac{\lambda}{\lambda+l}\right)^{\frac{\lambda}{l}}$. Since $l=k-1$, we get $comp_{\text{APSJ}} = 1-\frac{k-1}{k-1+\lambda}\cdot\left(\frac{\lambda}{\lambda+k-1}\right)^{\frac{\lambda}{k-1}}$.

[4]We minimize the comparison factor since the number of signature comparisons grows quadratically with the relation sizes and is the key performance penalty. Additional gains could be achieved by minimizing both factors simultaneously exploiting the known CPU and I/O performance characteristics.

Algorithm	Comparison and replication factors
PSJ	$comp_{\text{PSJ}} = 1 - \left(1 - \frac{1}{k}\right)^{\theta_S}$ $repl_{\text{PSJ}} = \frac{1}{1+\rho} + \frac{\rho}{1+\rho} k(1 - (1 - \frac{1}{k})^{\theta_S})$
APSJ	$comp_{\text{APSJ}} = 1 - \frac{k-1}{k-1+\lambda} \cdot \left(\frac{\lambda}{k-1+\lambda}\right)^{\frac{\lambda}{k-1}}$ $repl_{\text{APSJ}} = \frac{1}{1+\rho} + \frac{\rho}{1+\rho} \cdot \left(k - (k-1)\left(\frac{\lambda}{k-1+\lambda}\right)^{\frac{\lambda}{k-1}}\right)$
ADCJ	$comp_{\text{ADCJ}} = \left(1 - \frac{1}{1+\lambda}\left(\frac{\lambda}{1+\lambda}\right)^{\lambda}\right)^{\log_2 k}$ $repl_{\text{ADCJ}} = \texttt{replADCJ}(\lambda, k, \rho)$ (see Algorithm 1)

Table VII. Summary of replication and comparison factors for PSJ, APSJ, and ADCJ

The replication factor is determined as $repl_{\text{APSJ}} = \frac{\sum_{i=0}^{l} |R_i|+|S_i|}{|R|+|S|} = \frac{|R|+\sum_{i=0}^{l}|S_i|}{|R|+|S|} = \frac{|R|}{|R|+|S|} + \frac{|S|}{|R|+|S|} \cdot (1 + l \cdot (1 - p^{\theta_S}))$. Since $\rho = \frac{|S|}{|R|}$, we obtain $repl_{\text{APSJ}} = \frac{1}{1+\rho} + \frac{\rho}{1+\rho}(1 + l \cdot (1 - p^{\theta_S}))$. Substituting p by p_{opt}, and l by $k-1$ finally yields $repl_{\text{APSJ}} = \frac{1}{1+\rho} + \frac{\rho}{1+\rho} \cdot \left(k - (k-1)\left(\frac{\lambda}{k-1+\lambda}\right)^{\frac{\lambda}{k-1}}\right)$.

3.2 Analysis of ADCJ and PSJ

In Appendix B we derive the comparison and replication factors for PSJ. For ease of reference, $comp_{\text{PSJ}}$ and $repl_{\text{PSJ}}$ are listed in Table VII. The contribution of ADCJ is an optimized pattern according to which operators α and β are applied. In Appendix C we derive the comparison factor of the divide-and-conquer approach and demonstrate that it is independent of the operator pattern. As a consequence, the comparison factor of ADCJ (shown in Table VII) is equivalent to that of DCJ. The replication factor for ADCJ is hard to analyze and cannot be described using a closed formula. This is unfortunate, since estimating the comparison and replication factors is essential for choosing the best algorithm for the given input relations, as we discuss in Section 6. To overcome this limitation, we provide an algorithm that can be used for computing $repl_{\text{ADCJ}}$ (see Algorithm 1). The algorithm computes numerically the expected sizes of all partitions (just as we explained in Section 2.4), and adds up their sizes to obtain the replication factor.

Notice that the formulas for the comparison and replication factors of PSJ depend directly on the set cardinality θ_S. In contrast, the formulas for APSJ and ADCJ depend only on the ratio $\lambda = \frac{\theta_S}{\theta_R}$. This initial observation suggests that APSJ and ADCJ should be able to deal with large sets more effectively than PSJ.

4. QUALITATIVE COMPARISON OF THE ALGORITHMS

In the remainder of this paper, we will explore three aspects that are important for understanding and comparing the performance of the algorithms that we presented:

—First, we examine what the formulas that we derived in the previous sections tell us. In Section 4.1, we provide a qualitative estimate of how each of the algorithms performs with the increasing number of partitions, different relation sizes, or varying set cardinalities.

—Second, we investigate the accuracy of the predictions of our formulas. In Section 4.2, we demonstrate how the actual comparison and replication factors devi-

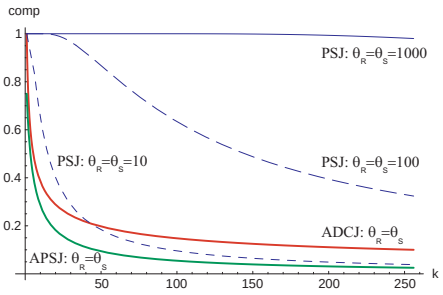

Fig. 4. Comparison factor vs. k

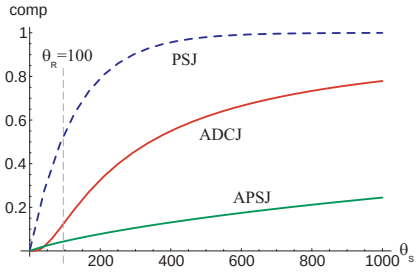

Fig. 5. Comparison factor vs. θ_S ($k = 128$)

ate from the predicted values under different distributions of element values and set cardinalities.

—Finally, we explore the behavior of the algorithms in an implemented system. In Section 6, we show how the algorithms perform in practice, and demonstrate how our analytical model helps to find operational values for the algorithms.

4.1 Understanding the formulas

Comparison factor. First, we illustrate the reduction of the comparison factor with the growing number of partitions. All comparison factors in Table VII are determined by the parameters θ_R, θ_S, and k. In Figure 4, we depict $comp_{\text{APSJ}}$, $comp_{\text{ADCJ}}$ 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 $comp_{\text{DCJ}}$ is equivalent to $comp_{\text{ADCJ}}$, we will not consider $comp_{\text{DCJ}}$ separately. Because ADCJ and APSJ depend on the ratio λ of set cardinalities only, and $\lambda = 1$ in all three cases, the three curves for each of these algorithms fall into one, depicted as a thick solid line. As can be seen in the figure, all comparison factors 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 is ineffective (with $comp_{\text{PSJ}} \approx 1$), while ADCJ requires 13% of comparisons as opposed to the full cross-product, and APSJ only 4.5%. On the other hand, for small sets like $\theta_R = \theta_S = 10$, PSJ outperforms ADCJ in the number of comparisons starting with $k \approx 40$. As a matter of fact, as k grows, PSJ eventually catches up even with APSJ at $k \approx 2^{13}$ (not shown in the figure). However, as we explain below, replication overhead increases with k, limiting the maximal number of partitions that can be used effectively for computing the join.

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[5] 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, $comp_{\text{ADCJ}}$ remains below $comp_{\text{PSJ}}$ as the cardinality ratio grows (although not shown in the figure, $comp_{\text{ADCJ}} < comp_{\text{PSJ}}$ holds for all

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

$\theta_S > 1000$).

Moreover, in all scenarios, even those in which initially $comp_{\text{ADCJ}} > comp_{\text{PSJ}}$, ADCJ (as well as APSJ) will eventually catch up and outperform PSJ as θ_S increases[6]. For example, starting with $\theta_R = \theta_S = 10$, and $k = 64$, we obtain $0.18 \approx comp_{\text{ADCJ}} > comp_{\text{PSJ}} \approx 0.15$. Still, as θ_S grows, ADCJ catches up with PSJ at $\theta_S \approx 110$, resulting in a comparison factor of 0.82 (at the same time, $comp_{\text{APSJ}} \approx 0.39$). Overall, for any $k > 2$, APSJ requires less comparisons than ADCJ (assuming large element domains).

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. We start with the case where $|R| = |S|$, i.e., $\rho = 1$. Figure 6 shows the growth of the replication factors $repl_{\text{APSJ}}$, $repl_{\text{ADCJ}}$, 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 $repl_{\text{APSJ}}$ and $repl_{\text{ADCJ}}$ depend only on the ratio of the set cardinalities; thus we obtain just one curve for APSJ and another one for ADCJ. Furthermore, both curves are almost identical for the settings of Figure 6 (the curve for DCJ is very close to that of ADCJ and is not shown in the figure). Notice that $repl_{\text{ADCJ}}$ and $repl_{\text{APSJ}}$ outperform $repl_{\text{PSJ}}$ even for $\theta_R = \theta_S = 10$. For larger sets, like $\theta_R = \theta_S = 100$, and $k = 128$, PSJ needs to write out $35 \cdot (|R| + |S|)$ signatures as partition data. This is 10 times more data to be stored temporarily than that generated by APSJ and ADCJ. 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, $repl_{\text{ADCJ}}$ and $repl_{\text{APSJ}}$ 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 $repl_{\text{ADCJ}}$ or $repl_{\text{APSJ}}$. For large sets, such k may be so enormous that the fact that PSJ is bound and ADCJ/APSJ are not is practically irrelevant. For example, for $\theta_R = \theta_S = 1000$, $repl_{\text{ADCJ}}$ becomes as large as the maximal value of $repl_{\text{PSJ}}$ $(0.5 + 500 = 500.5)$, when $k \approx 2^{36}$. Since factor $repl_{\text{ADCJ}}$ grows faster with increasing k than $repl_{\text{APSJ}}$, APSJ eventually catches up and outperforms ADCJ starting from any setting. However, the breakeven value of k may be high, especially for large λ. For example, for $\lambda = 10$, APSJ does not catch up with ADCJ until $k \approx 2^{16}$.

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 $repl_{\text{ADCJ}}$, and even $repl_{\text{DCJ}}$ eventually win over $repl_{\text{APSJ}}$. Moreover, not only $repl_{\text{ADCJ}}$ outperforms $repl_{\text{DCJ}}$, but, surprisingly, $repl_{\text{ADCJ}}$ peaks at some value of λ and starts decreasing from that point on. In Figure 7, the peak replication for ADCJ (5.8) is produced at $\lambda \approx 4.8$, or $\theta_S \approx 480$. Notice that replication factor of 5.8 is still 30% better than the corresponding values for APSJ (8.47) or DCJ (8.9).

Figure 8 illustrates the benefit of using ADCJ when the superset relation S is larger than the subset relation R. Notice that the replication factors of PSJ, APSJ, and DCJ increase with growing ρ, whereas $repl_{\text{ADCJ}}$ decreases starting from $\rho = 1$, and, in fact, approaches 1 as ρ continues to grow. Although $repl_{\text{PSJ}}$ saturates

[6]This fact can be derived from formulas in Table VII.

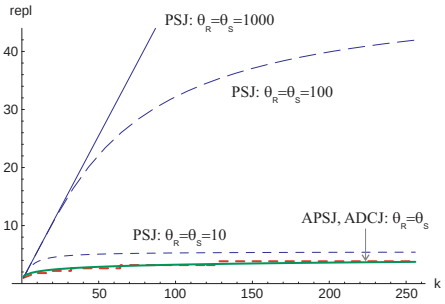

Fig. 6. Replication factor vs. k

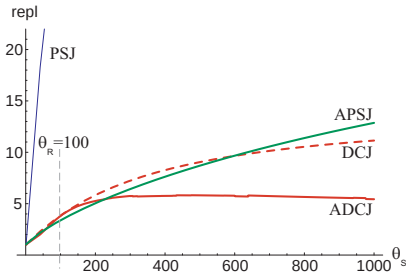

Fig. 7. Replication factor vs. θ_S (for $\rho = 1$)

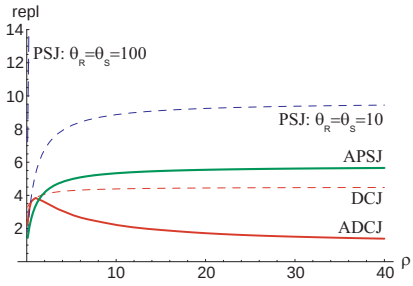

Fig. 8. Replication factor vs. ρ (for $\lambda = 1$)

quickly, for larger sets the replication overhead of PSJ is still extremely high. For larger values of λ and k, the curve for DCJ peaks at some ρ and starts decreasing from that point on. However, for any setting, ADCJ always outperforms 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 very small set cardinalities (below 10 elements), while APSJ and ADCJ start outperforming PSJ when the set cardinalities increase. In most scenarios, APSJ yields the smallest comparison factor, whereas ADCJ may outperform APSJ due to small replication factor for larger λ or ρ. In Section 6, we present the experimental results that substantiate these observations.

4.2 Accuracy of analytical model

To study the accuracy of our formulas in realistic scenarios, we used synthetic and real data. For generating the synthetic data, we used five different distributions of element values, and five distributions of set cardinalities as listed in Table VIII. Starting with the distributions that are close to the assumptions of our analytical model, we gradually made them more and more distinct. Using simulations, we studied both the individual impact of varying just the element distribution or just

Case	Element distribution	Set cardinality distributions in R : in S
A	uniform (5000, $\frac{10000}{\sqrt{12}}$)	uniform (50, $\frac{10}{\sqrt{12}}$) : (100, $\frac{20}{\sqrt{12}}$)
B	normal (5000, 1000)	normal (50, 5) : (100, 10)
C	normal (5000, 500)	normal (50, 10) : (100, 20)
D	normal (5000, 100)	normal (50, 20) : (100, 40)
E	uniform (5000, $\frac{200}{\sqrt{12}}$)	uniform (50, $\frac{100}{\sqrt{12}}$) : (100, $\frac{200}{\sqrt{12}}$)

Table VIII. Element and set cardinality distributions characterized by mean μ and standard deviation σ, which are denoted as pairs (μ, σ).

the set cardinality distributions, as well as the combined effect. We discuss this study in more detail in Appendix E. Experiments on real data are presented in Section 6.

In summary, we found that for a variety of set cardinality distributions the formulas of Table VII (including Algorithm 1 for ADCJ) deliver relatively accurate predictions that lie within 15% of the actual values, as long as the element domains are at least 10 times larger than the average set cardinalities and a large number of domain elements is used in the sets. Our predictions deviate more from actual values when the domain size $|\mathcal{D}|$ approaches the average set cardinalities θ_R and θ_S. In our study, the selectivity of the joins ranged from $3.4 \cdot 10^{-107}$ to $3.6 \cdot 10^{-2}$. When the join selectivity is high, the execution time of either algorithm is dominated by the retrieval of the joining tuples.

Across all experiments we observed that APSJ and ADCJ tend to be more negatively affected by varying the distributions than PSJ. This effect is mainly attributed to problems with the generation of the boolean hash functions. Recall that for APSJ and ADCJ to work optimally, we need to generate hash functions that fire with a certain fixed probability. The smaller the element domain, the worse the bit-string approach approximates the required probabilities.

5. TESTBED

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. To provide a fair evaluation of different partitioning algorithms, we implemented the set containment join operator in such a way that just the actual partitioning algorithm can be exchanged, other conditions remaining equal. In Appendix G we document the Java implementations of each of the algorithms APSJ and ADCJ as deployed in our testbed.

For conducting our experiments we used a more flexible and accurate evaluation testbed than the one deployed in [Melnik and Garcia-Molina 2002]. To improve the accuracy of measurements we specifically addressed the issues of I/O caching done by operating systems and monitoring Java memory usage. OS caching effectively makes the data on the secondary storage be loaded into the main memory, distorting the I/O characteristics of the experiments, and is hard to control. In [Melnik and Garcia-Molina 2002], we minimized OS caching by limiting the total amount of OS memory. In the current testbed, we modified and recompiled the kernel of the

Linux OS to completely disable read caching and prefetching[7]. To eliminate the OS caching of write operations, we mounted the disk in a synchronous mode.

The second issue that we addressed is the memory usage of the Java Virtual Machine, which is hard to measure and to control, not least due to its automatic garbage collection. We implemented our own memory monitor that is explicitly called throughout the code on allocating and deallocating memory and counts the actual number of bytes used for signatures and tuples.

Every join computation runs within a fixed memory window. Its size can be chosen independently from the database cache size. In the partitioning phase, the memory window is divided equally among each partition. Once a partition's memory share fills up, it is written out to disk. In the joining phase, partitions R_i and S_i are joined using the available memory window. If R_i and S_i do not fit into the memory window, they are divided into blocks of half the window size and are joined blockwise in a nested loop. The IDs of the candidate tuples are written out to disk or kept in a special memory buffer, as explained below.

We experimented with two verification procedures called verify-disk and verify-mem. In verify-disk, we generalized the technique used in [Ramasamy et al. 2000] in order to cope with limited memory. In the joining phase, just as in [Ramasamy et al. 2000], the IDs of the candidate tuples are written out to disk. In the verification phase, if all candidate tuples of R do not fit into the available memory window, we read the tuple IDs of R and S blockwise, sort them in memory to achieve sequential reads, and fetch the tuples to be verified in each block. To estimate dynamically the maximal number of signatures and tuples that can be read into the memory window, we store with each signature the size of the corresponding set. Thus, after reading n signatures we can determine the memory needed to hold their respective tuples. To minimize the storage overhead, the set size is approximated by its upper bound using a two-exponent stored in a single byte. For example, value 10 stored with the signature means that the respective set contains at most $2^{10} = 1024$ elements.

The second technique, verify-mem, was designed to reduce the verification time in cases when the verification data is especially large and writing it out to disk may be prohibitive. The key idea is to avoid storing the candidate pairs on disk completely. For this purpose, we utilize a fixed-size verification buffer that is used during the joining phase to hold the candidate pairs and their tuples. In our testbed, the verification buffer plays the role of the result buffer that is typically used in commercial databases to hold the intermediate results of query operators. Every candidate pair produced in the joining phase is appended to the verification buffer. Once it fills up, the tuple IDs of R and S are sorted and the respective tuples are fetched from disk. Thanks to the size estimates stored in the signatures, the tuples are guaranteed to fit into the available verification buffer. After the verification buffer has been 'drained', it is cleared and the joining phase resumes.

An added benefit of interlacing joining and verification in verify-mem is that the first results become available much earlier than using a strictly ordered three-phase technique. By increasing the verification buffer, the partitioning algorithms can trade memory for speed, whereas the nested-loop joins are CPU-bound and cannot

[7] A more standard solution of mounting a disk as a raw device [Ramasamy et al. 2000] was not an option since Berkeley DB (v4.0.14) does not support raw device access.

exploit additional memory effectively.

We implemented two nested loop algorithms, which enumerate the complete cross-product $R \times S$, to contrast their performance with that of the partitioning algorithms. In the naive nested loop join, the subset predicate is evaluated for each pair of sets. In the signature nested loop join, each pair of signatures is tested for bitwise inclusion producing candidate pairs that need to be verified. The nested loop algorithms run within a fixed memory window, just as the partitioning algorithms.

Overall, our testbed provides an execution environment close to what a commercial DBMS would see, except that it provides the flexibility to experiment with different join algorithms and collect relevant execution data.

6. EXPERIMENTS

In this section, we illustrate the performance of the algorithms on real and synthetic data sets. The real data sets help us study the effectiveness of the algorithms for distributions of set elements and cardinalities that can be found in practical applications. Using synthetic data we explore the behavior of the algorithms for a variety of settings and illustrate the performance trends.

First, we compare the performance of APSJ and PSJ and that of ADCJ and DCJ using the data gathered in the Stanford WebBase project. We highlight a challenging issue that is due to using signatures and is common to all partitioning algorithms. After that, we discuss the experiments on the weblog and SWISSPROT data, and illustrate the performance trends of the new algorithms using synthetic data. For conducting the experiments we used a 1.6 GHz machine with 512 MB of memory and an IDE disk that provided a read bandwidth of 13 MB/s and a write bandwidth of 0.93 MB/s.

WebBase data: outlinks-200. In this experiment we used the data gathered in the Stanford WebBase project [Hirai et al. 2000]. We used a snapshot of the WebBase repository containing about 100 million web pages to find web directories that are not subsumed by other, larger directories. We considered a web page to be a directory if it contains at least 200 outgoing links. More than 100K pages in the repository satisfied this condition. We created a relation R(`id`, `outlinks`) that contains one tuple for each directory page. The set-valued attribute `outlinks` holds the identifiers of all pages referenced by the page `id`. Each page identifier is represented as a 32-bit integer. To find the subsumed directories, we perform a self-join $R \bowtie_{\subseteq} R$.

The relation R with $|R| = 109949$ and average set cardinality $\theta_R = 224$ contains a total of 93 MB of raw data. This raw data was stored in a 160 MB Berkeley DB database file (72% B-tree overhead). The size of the result $|R \bowtie_{\subseteq} R|$ is 1672579, i.e., the join selectivity amounts to $1.4 \cdot 10^{-4}$. The signature size of 160 bits used in the experiment yielded ca. 3.5 million pairs of tuple IDs to be verified, i.e., an average of 1.1 false positives for a matching pair.

Figures 9 and 10 depict the execution times of PSJ and APSJ for the above setting, which we refer to as outlinks-200. The execution time of each phase is shown in stacked columns. The database cache size and the memory window were set to 32 MB each. The verify-disk technique was used.The average set cardinality

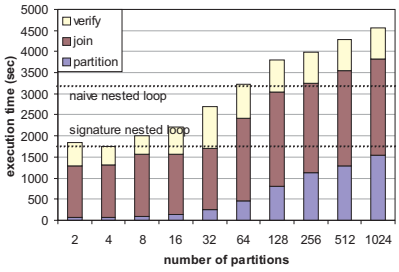

Fig. 9. PSJ on outlinks-200 data

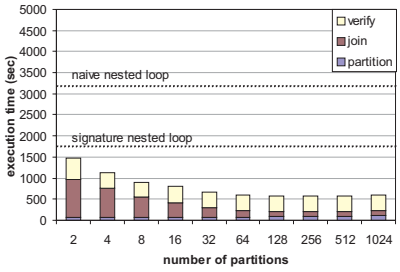

Fig. 10. APSJ on outlinks-200 data

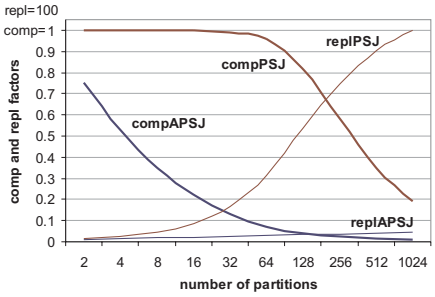

Fig. 11. Comparison and replication factors in outlinks-200

needed for setting the parameters of APSJ were obtained by sampling 1% of the relations. Each data point was obtained as an average of three 'cold' runs.

The figures illustrate the case in which PSJ is virtually ineffective due to large set cardinalities. To understand the performance graphs better, consider the actual comparison and replication factors obtained in this experiment that are shown in Figure 11 (on the vertical axis, *comp* ranges from 0 to 1, whereas *repl* ranges from 0 to 100). As predicted by the formulas of Table VII, $comp_{\text{PSJ}}$ starts dropping substantially only from $k \geq 128$, where $comp_{\text{PSJ}}^{k=128} \approx 0.82$. However, $repl_{\text{PSJ}}^{k=128}$ soars to around 53, wiping out the gains of the reduced CPU time. The value of $comp_{\text{PSJ}}^{k=1024} \approx 0.19$ is comparable to $comp_{\text{APSJ}}^{k=16} \approx 0.22$, i.e., the CPU load of APSJ in the joining phase at $k = 16$ is approximately equal to that of PSJ at $k = 1024$. However, the joining phase of APSJ takes only 230 sec, whereas that of PSJ takes 2280 sec, almost ten times as much. The difference is due to two major factors: the overhead required for reading a much larger amount of partitioning data and an increased fragmentation of the partitioning data due to a large number of partitions.

The performance graphs of ADCJ and DCJ for outlinks-200 are almost identical, since the ratio ρ of relation cardinalities is 1 (we omit the graphs for brevity). Both perform slightly worse than APSJ. For comparison, the execution times of the naive nested loop join and the signature nested loop join are shown as dotted lines in Figures 9 and 10. The naive nested loop join on outlinks-200 requires more than one hour, i.e., seven times longer than APSJ at $k = 128$.

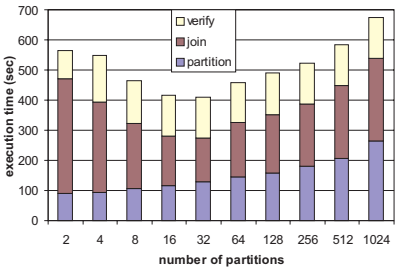

Fig. 12. DCJ on outlinks-150 data

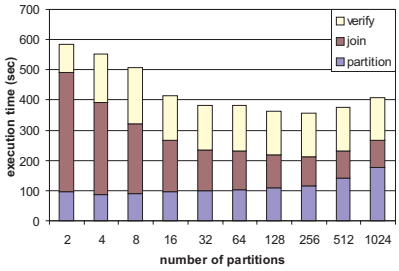

Fig. 13. ADCJ on outlinks-150 data

WebBase data: outlinks-150. To illustrate a case in which ADCJ does outperform DCJ, we conducted another experiment called outlinks-150. In this experiment, a web page is considered a directory if it contains at least 150 outgoing links. A total of $|S| = 341967$ pages satisfy this condition. Now imagine that our goal is to find 'new' directories in a subsequent crawl, e.g., those that have been added or modified since the previous crawl and whose outlinks are not completely covered by an existing directory. We extracted $|R| = 12400$ 'new' directories to be tested for the subset condition against the 'old' ones. The tuples of R were obtained as a random sample taken from S. Notice that $|R| < |S|$ with $\rho \approx 27$. According to the analysis of Section 4.1, in this case ADCJ should outperform DCJ.

Figures 12 and 13 depict the execution times of ADCJ and DCJ for outlinks-150, using the same configuration as in outlinks-200. The result size $|R \bowtie_{\subseteq} S|$ is 288953, with 570K pairs to verify. For each k, ADCJ outperforms DCJ due to a smaller replication factor. For example, at $k = 128$, $repl_{\text{DCJ}}^{k=128} \approx 4.39$ yielding the total of 88 MB of partitioning data to be generated. In contrast, $repl_{\text{ADCJ}}^{k=128} \approx 1.55$, with only 30 MB of partitioning data. Since $repl_{\text{ADCJ}}$ grows slower than $repl_{\text{DCJ}}$, ADCJ can leverage the CPU-time savings at a larger k more effectively. So, the optimal execution time of ADCJ is reached at $k = 256$, whereas DCJ starts degrading after $k = 32$. In outlinks-150, the running time of APSJ is slightly worse than that of ADCJ and DCJ, while PSJ is ineffective (the graphs are omitted for compactness). The running times of the signature nested loop and the naive nested loop lie at 800 sec and 1460 sec, respectively, out of range of the graphs.

The above experiments illustrate that the adaptive partitioning schemes of APSJ and ADCJ reduce CPU load by keeping the replication factor low. Moreover, smaller replication means that the total execution time is less sensitive to the choice of the number k of partitions, a critical parameter of all partitioning algorithms. For example, for $64 \leq k \leq 1024$, the respective running times of APSJ (Figure 10) and ADCJ (Figure 13) vary only insignificantly.

Weblog data. The 'outlinks' experiments hint to another major challenge in using the partitioning algorithms, in addition to choosing a good k. Notice that in both experiments the verification phase, in which the final result is obtained, constitutes a large fraction of the total execution time. In outlinks-200, it dominates the join computation. In outlinks-150, the percental gains of ADCJ over DCJ are reduced substantially due to verification. The above experiments suggest that in a case when

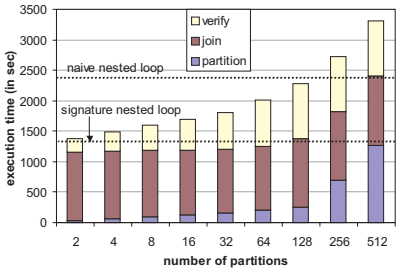

Fig. 14. PSJ on weblog data

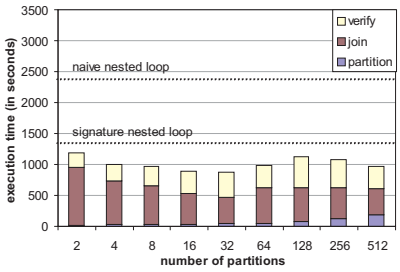

Fig. 15. APSJ on weblog data

the join result is very large, the verify-disk procedure may become prohibitively expensive. In this subsection, we study the impact of large results in more detail and illustrate the benefits of the verify-mem technique.

We conducted an experiment using the weblog data gathered by the web server of the C.S. department at Leipzig University, Germany over the past two years. In the weblog experiment, we identify website users who viewed a subset of pages visited by other users. Such information can, for example, be utilized by a recommendation system that suggests to the visitors potential pages of interest. The relation S contains sets of pages visited from each IP address, represented as 32-bit integers. To factor out non-representative users and obvious search engine accesses, we discarded all sets that contain less than 5 or more than 1000 elements. Thus, we obtained $|S| = 182144$ different sets with an average cardinality of $\theta_S = 28$. For populating the relation R, we picked only those users coming from the .de domain, approximating the German website visitors, a possible target group of the recommendation system. We obtained $|R| = 41781$ with $\theta_R = 36$. Thus, the join $R \bowtie_{\subseteq} S$ has the parameters $\lambda \approx 0.77$ and $\rho \approx 4.43$.

The result of the join contains 21.5 million pairs, i.e., the join selectivity is relatively high at 0.003. For a signature size of 160 bits, 87.8 million pairs to verify need to be stored on disk. This verification data consumes approximately 1.06 GB of disk space. The time required for writing and reading this data, and for fetching the candidate tuples is around 50 min, making all partitioning algorithms that use the verify-disk technique perform worse than a naive nested loop join, which takes 41 min. Furthermore, storing a gigabyte of temporary data may cause a disk overflow even on modern hardware. The relations R and S utilize 10 MB and 32 MB on disk, respectively. To make the available memory smaller than the database size, we set the database cache and memory window to 4 MB each. The verification window, or result buffer, was set to 28 MB, enough to keep around 1.5% of the join result on average (the buffer was filled around 70 times during joining).

Figures 14 and 15 depict the execution times of PSJ and APSJ on the weblog data using the verify-mem technique. In case of APSJ the complete result can be produced in less than 15 min. Although the fraction of the verification time is still substantial, it lies far below 50 min needed for verify-disk. PSJ turns out to be ineffective due to quickly growing replication, just as in the outlinks experiments.

Another way of reducing the number of false positives is to increase the signature

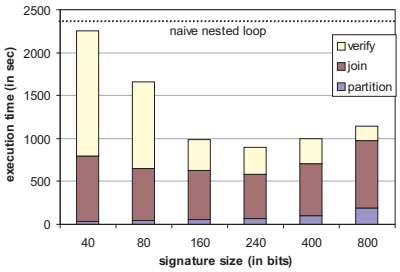

Fig. 16. Execution time vs. signature size (APSJ, $k = 64$)

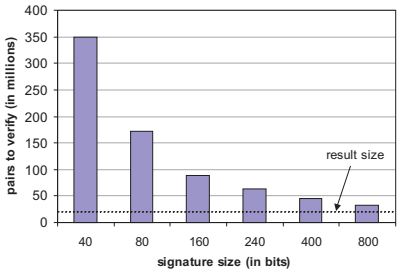

Fig. 17. Number of candidate pairs vs. signature size (APSJ, $k = 64$)

Relation	Size	Avg. set cardinality	Max. set cardinality	DB file size
Author	146935	11	7423	12.6 MB
Gene	83485	2	512	2.5 MB
Organism	4834	190	8002	4.35 MB

Table IX. Relations extracted from the SWISSPROT database

size. In the weblog experiment, we could confirm the results of [Ramasamy et al. 2000] that the choice of the signature size does not impact the performance of the partitioning algorithms significantly. Figure 16 shows the execution time of APSJ with $k = 64$ for signature sizes ranging between 40 and 800 bits. Notice that the horizontal axis is quasi logarithmic. The overall performance remains roughly the same for signature sizes between 160 and 400 bits. Although the number of false positives drops with the growing signature size (see Figure 17), the size of partitioning data and the CPU time for signature comparisons increase.

SWISSPROT data. For completeness, we present a scenario that is not favorable to our new algorithms. We downloaded the SWISSPROT database[8], which contains detailed information about 110000 proteins. From this data we extracted three relations that are summarized in Table IX. Each relation has the signature R(`id`, `proteins`). In the relation Author, each set represents the proteins studied by an individual researcher, i.e., those that appear in his or her publications. In the relation Genes, each set contains the proteins that a given gene codes for. The relation Organism contains sets of proteins grouped by classes of organisms such as humans, primates or bacteria.

Using these three relations, six set containment joins in total can be performed. These are listed in Table X. For example, experiment (II) allows us to identify all researchers who published about each protein of a given organism. These researchers are likely to have specific interest and expertise in certain species. We used the same memory settings as in the weblog experiment (4 MB cache, 4 MB memory window, 28 MB verification buffer). The signature size of 400 bits was used in (IV) and (VI) because of large set cardinalities of the superset relation Organism. In all other experiments, the signature size was set to 160 bits.

[8]Available online at http://www.expasy.org/sprot/, release 4.0

Experiment	Joined relations	Result size	Selectivity	λ	ρ
I	Organism $\bowtie_{\subseteq}$ Gene	1477	$3.6 \cdot 10^{-6}$	0.01	17.3
II	Organism $\bowtie_{\subseteq}$ Author	8923	$1.3 \cdot 10^{-5}$	0.058	30.6
III	Author $\bowtie_{\subseteq}$ Gene	120904	$9.8 \cdot 10^{-6}$	0.18	0.56
IV	Author $\bowtie_{\subseteq}$ Organism	1273889	$1.7 \cdot 10^{-3}$	17.3	0.33
V	Gene $\bowtie_{\subseteq}$ Author	2343579	$1.9 \cdot 10^{-4}$	5.5	1.77
VI	Gene $\bowtie_{\subseteq}$ Organism	624291	$1.5 \cdot 10^{-3}$	95	0.058

Table X. Experiments performed on SWISSPROT data

Algorithm	I	II	III	IV	V	VI
APSJ	60	**140**	860	480	**1200**	270
ADCJ	70	155	770	1100	4200	1040
PSJ	**30**	175	**100**	510	1300	250
nested loop	120	230	3200	**330**	1800	**170**

Table XI. Best times of SWISSPROT experiments (in sec)

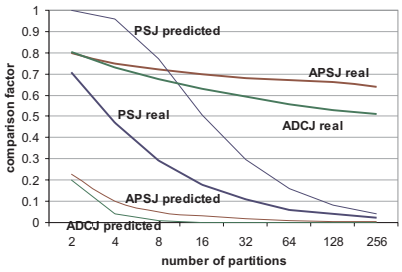

Fig. 18. Comparison factor in (II)

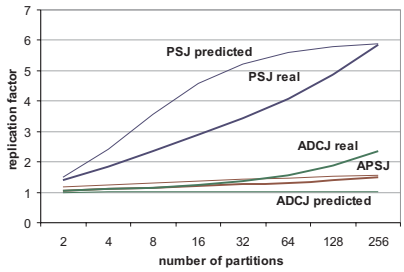

Fig. 19. Replication factor in (II)

The best execution times obtained for each algorithm and each experiment are summarized in Table XI. For each experiment, we ran each algorithm for $k = 2^l$, $l = 1, \ldots, 9$ and picked its best execution time. The top times in each experiment are highlighted in bold. In experiments (I) and (III), the set cardinalities of the superset relation Gene are extremely low (around 2), making PSJ the algorithm of choice. In (II) and (V), APSJ wins due to less replication, despite the fact that its comparison factor deviates substantially from the values predicted by the formula of Table VII. Although in general the non-uniform distributions affect the performance of the algorithms negatively (as in Figure 18), in some cases, such as case (V) illustrated in Figures 20 and 21, the actual values turn out to be better than the predicted ones.

In cases (IV) and (VI), the partitioning algorithms loose to the naive nested loop join. One reason for that is a large number of false positives. In (IV), 50 million false positives are produced on average for 1.2 million tuples in the result, whereas in (VI) the ratio is 30 million to 620 thousand. Just as in the weblog experiment, increasing the signature size does not reduce the running time noticeably due to growing replication data and CPU load of signature comparisons.

The high number of false positives can be explained by the fact that the SWISSPROT relations contain a significant fraction of very large and very small sets.

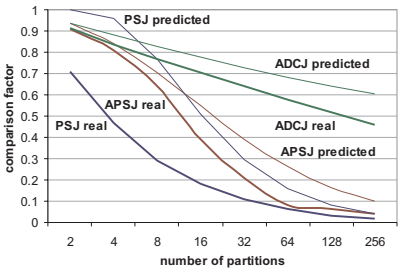

Fig. 20. Comparison factor in (V)

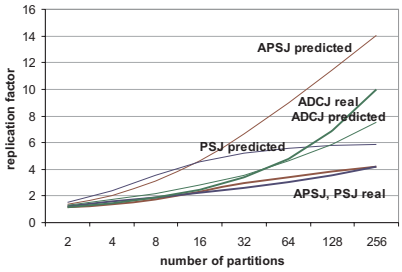

Fig. 21. Replication factor in (V)

The large sets produce almost full signatures, which are likely to yield many false positives. Nearly empty signatures are also prone to producing false positives. For example, in (IV), given close to 80000 sets with only one element each, we get on average 200 signatures with the same single bit set for a signature of 400 bits. We think that processing the 'large' and 'small' sets separately, e.g., using a nested loop join for those sets only, may help improve the overall performance. We do not examine such a hybrid algorithm in this paper.

In this section, we have studied several scenarios. In some of them, our new algorithms are the best, but in others they are not. An important question is how we can choose the optimal operational values for each algorithm and decide in practice which algorithm to use in which scenario. This question is the topic of the next section.

7. CHOOSING OPERATIONAL VALUES OF ALGORITHMS

Finding an optimal number of partitions is essential for deploying the partitioning algorithms effectively. In a real system, we cannot afford running the algorithms for different values of k to determine the optimal k. The technique that we developed in [Melnik and Garcia-Molina 2002] helps us predict the best operational values for the algorithms and choose the best performing algorithm.

We approximate the running time of each 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. [2000]. As they reported, 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. For predicting the execution times, 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. As illustrated in Section 6, the partitioning algorithms spend a comparable additional amount of time on verifying and 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 [Melnik and Garcia-Molina 2002] we derive the equation for *time* using the least-squares curve fitting method. We found that the function $time(x, y, k) = c_1 \cdot x + c_2 \cdot y \cdot k^{c_3}$ resulted in the smallest average error of all candidate functions that we explored. 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.

We think that in real systems it would be possible to automate the computation of the time equation and the choice of the best algorithm. The time equation could be obtained at configuration time of the system by generating a fixed number of synthetic input relations R and S and obtaining the data points for different values of k using algorithms APSJ, ADCJ, and PSJ. Then, the curve-fitting method could be automatically applied to choose the equation that predicts the execution time most accurately. After the equation is obtained, the hardware has been 'calibrated'. Then the equation can be used to predict the running time of actual joins that must be run. Given the time equation, the decision what algorithm to select for two input relations R and S can be made using the following steps:

(1) Determine the actual sizes of the relations and the average set cardinalities θ_R and θ_S using available statistics or sampling.

(2) Estimate the comparison and replication factors using the formulas of Table VII for a number of different values of k, for example for $k = 2^l$, $1 \leq l \leq 13$.

(3) Apply the time equation to determine the best execution times of each of the algorithms for the above values of k using the estimated comparison and replication factors[9].

(4) Find the best execution time and pick the algorithm that produced it along with the optimal partition number k.

To illustrate the use of the above procedure in our testbed, consider Figures 22 and 23. The figures depict the predicted and the actual execution times obtained for two synthetic relations $|R| = |S| = 50000$ ($\theta_R = 50$, $\theta_S = 100$) using the algorithms ADCJ and PSJ. The time equation $time(x, y, k) = 5.0824 \cdot 10^{-7} \cdot x + 7.3093 \cdot 10^{-7} \cdot y \cdot k^{0.9162}$ was obtained by generating six synthetic relations and 'calibrating' the hardware on a 600 MHz Linux machine with 256 MB of memory. For generating the synthetic relations, we used different combinations of the element and set cardinality distributions of cases A and B of Table VIII.

In Step 1, the relation sizes were obtained by querying the relation metadata maintained by Berkeley DB. The average set cardinalities were computed by sampling 5% of the relations. In Steps 2 and 3, the predicted execution time was determined by first computing the comparison and replication factors using the formulas of Table VII, and then inserting the corresponding values of $x = comp \cdot |R| \cdot |S|$ and $y = repl \cdot (|R| + |S|)$ in the above *time* formula. Notice that in case of PSJ the predicted optimal number of partitions ($k = 512$) matches the experimental value. In contrast, the experimental optimal k for ADCJ is 1024, whereas the formula

[9]Since the formulas in Table VII are fairly complex, determining the optimal k analytically is hard. Moreover, no closed formula for $repl_{\text{ADCJ}}$ is available. Therefore, we use the probing approach.

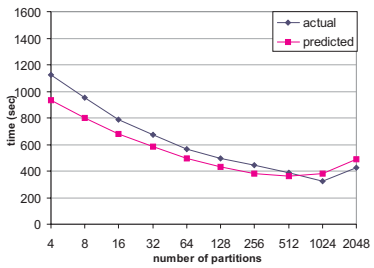

Fig. 22. Execution times for ADCJ ($|R| = |S| = 50000$, $\theta_R = 50$, $\theta_S = 100$)

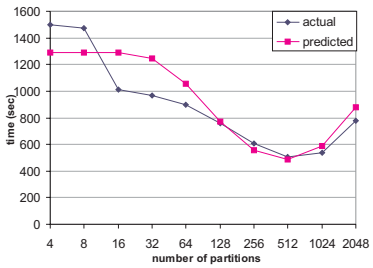

Fig. 23. Execution times for PSJ ($|R| = |S| = 50000$, $\theta_R = 50$, $\theta_S = 100$)

predicts $k = 512$. Even though the predicted k for ADCJ was less accurate, the formula still provides a good estimate of the best execution time.

The predicted time depends on precise estimates of the comparison and replication factors and may be less accurate for highly correlated data. Furthermore, as we illustrated in Section 6, non-uniform data may yield a large number of false positives and a prohibitively expensive verification phase. To account for these challenges in a real system, the execution of the partitioning algorithm picked in Step 4 could be augmented as follows:

(5) After the partitioning is done, check the actual comparison and replication factors. If the values are unexpectedly poor (e.g., *comp* is close to 1), fall back to a nested loop join.

(6) If the actual factors match the predictions, run the joining phase for a fraction of the total time, and then check how many false positives are obtained. If there are prohibitively many false positives, switch to a nested loop join.

8. PERFORMANCE TRENDS

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 APSJ, ADCJ, or PSJ perform best. For any given hardware configuration, the space of input relations can be divided into areas where one of the algorithms outperforms the others. Figure 24 shows eight regions divided into areas where one of the algorithms excels. Each region is composed of 100 discrete data points. Each point corresponds to a particular data set characterized by four parameters θ_R, $|R|$, λ and ρ. The values of parameters $|R|$ (in thousands) and θ_R are depicted along the x-axis and y-axis, respectively, while λ and ρ are kept constant for each region. To obtain Figure 24 experimentally, 31200 joins need to be executed. If each experiment takes on average 10 min, more than seven month of computation would be required. Therefore, to illustrate the performance trends we used the time equation to determine the algorithm that is likely to perform best at each point.

The top four regions were obtained using the time equation of Section 7 calibrated on a 600 MHz machine. For example, the top left region shows the areas of excellence of the algorithms for $\lambda = 1$ and $\rho = 1$. As we can see, APSJ outperforms

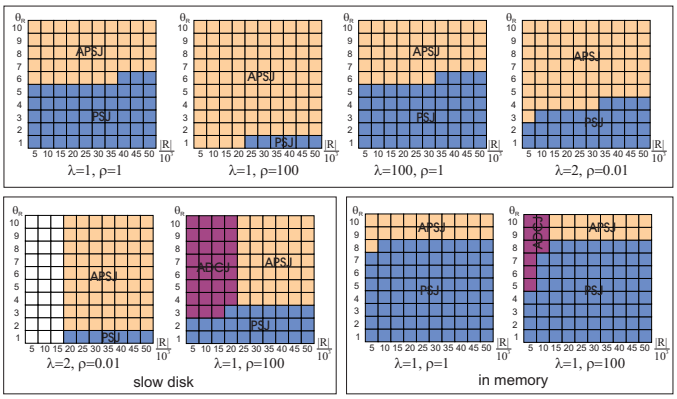

Fig. 24. Performance regions for APSJ, ADCJ, and PSJ for different hardware settings

the algorithms ADCJ and PSJ starting from $\theta_R = \theta_S = 6$ for relation sizes up to 35000, and from $\theta_R = \theta_S = 7$ for relation sizes between 40000 and 50000. Although not shown in the figure, APSJ continues outperforming the other algorithms for larger θ_R. For example, for $|R| = |S| = 20000$, $\theta_R = \theta_S = 100$, APSJ is 11 times faster than PSJ and twice as fast as ADCJ. Notice that ADCJ does not appear in the top four regions at all. In fact, in this hardware configuration that we used, ADCJ is always dominated by either APSJ or PSJ.

The bottom four regions show the performance regions for another two hardware settings. In the 'in memory' setting, we configured our testbed to do in-memory partitioning and calibrated the time equation as $time(x, y, k) = 5.0824 \cdot 10^{-7} \cdot x + 3.6546 \cdot 10^{-5} \cdot y$. Notice that $c_3 = 0$ (and thus $k^{c_3} = 1$), since there is no fragmentation effect. In the 'slow disk' setting, we modified the time equation to simulate quadratic fragmentation impact (reported in [Ramasamy et al. 2000]) and a slow disk as $time(x, y, k) = 5.0824 \cdot 10^{-7} \cdot x + 7.3093 \cdot 10^{-6} \cdot y \cdot k^2$. In both settings, when relation S is 100 times larger than R, ADCJ becomes the algorithm of choice for smaller sizes of R. However, the gain of ADCJ over APSJ is just around 10% (not shown in the figure). Surprisingly, in the in-memory setting ADCJ outperforms APSJ by a larger margin. For instance, for the point $|R| = 1000$, $|S| = 100000$, $\theta_R = \theta_S = 10$ (truncated in bottom right region), ADCJ is three times better than APSJ and four times better than PSJ. Notice that in the 'slow disk' setting with $\lambda = 2$, $\rho = 0.01$, none of the algorithms is effective for $|R| < 20000$ (blank area in the region). In this case, relation S is very small ($|S| = 0.01 \cdot |R| < 200$), and the partitioning overhead makes each of the algorithms less effective than a nested loop join. We observed this trend in the SWISSPROT experiments (IV) and (VI) in Section 6, where the superset relation Genes was relatively small and the disk was quite slow for a 1.6 GHz CPU.

By varying the time equation, we simulated several other hardware settings beyond the three shown in Figure 24. In all cases that we examined, either APSJ or ADCJ outperforms PSJ when $\theta_R > 9$, and in many other cases even if θ_R is smaller than 9. In most configurations APSJ turns out to be the top performer for larger

sets. However, for smaller relations and large ρ, ADCJ wins over APSJ. Please keep in mind that the results presented above are based on the assumption that the element domains are large, the data is distributed relatively uniformly, and the result size is small. As we demonstrated in Section 4.2, ADCJ may outperform APSJ for a broader range of data sets if the element domain is small. Furthermore, as we showed in Section 6, PSJ may win over ADCJ and APSJ if the data is highly correlated; the nested loop may outperform all partitioning algorithms if the result size and the number of false positives are very large.

9. RELATED WORK

The set containment join and other join operators for sets enjoyed significant attention in the area of data modeling. However, relatively little work deals with efficient implementations of these operators. Helmer and Moerkotte [1997] 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. [2000] 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 [Helmer and Moerkotte 1997] and [Ramasamy et al. 2000], 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 [1984]. Ishikawa et al. [1993] applied the signature file technique for finding subsets or supersets that match a fixed given query set in object-oriented databases.

In [Melnik and Garcia-Molina 2002] we presented the Divide-and-Conquer Set Join (DCJ) and the Lattice Set Join (LSJ) algorithms. LSJ is a partitioning algorithm which extends the main-memory algorithm SHJ [Helmer and Moerkotte 1997]. We demonstrated that DCJ always outperforms LSJ in terms of the replication factor. In [Melnik and Garcia-Molina 2002] we developed a comprehensive model for analyzing different partitioning algorithms that takes into account different set cardinalities and relation sizes, and measures the efficiency of the algorithms using the comparison and replication factors. In this paper, we used this analytical model for studying our novel algorithms APSJ and ADCJ.

The adaptive algorithms presented in this paper introduce significant improvement over PSJ and DCJ. In particular, ADCJ always outperforms DCJ due to smaller replication factor, just like DCJ outperforms LSJ. In [Melnik and Garcia-Molina 2002] we suggested for DCJ a fixed pattern for applying operators α and β, which works reasonably well when the input relations R and S have approximately equal sizes and the set cardinalities are approximately the same (i.e., $\rho \approx 1, \lambda \approx 1$). In this paper, we compute the α,β-pattern adaptively based on the characteristics of the input relations to minimize replication.

For brevity we do not discuss several aspects relevant for computing set containment joins. Examples are trading CPU time for I/O time by selecting the algorithm and partition number appropriately, choosing the signature size optimally, or using multi-stage partitioning (some of these aspects are examined in [Ramasamy et al.

2000]). For generating synthetic databases used in our experiments, we deployed the methods described in [Gray et al. 1994]. The inherent theoretical complexity of computing set containment joins was addressed in [Cai et al. 2001; Hellerstein et al. 1997]. Partitioning has been utilized for computing joins over other types of non-atomic data, e.g., for spatial joins [Patel and DeWitt 1996]. A possible alternative to partitioning joins are index joins. Index-based approaches for accessing multi-dimensional data were studied e.g. in [Böhm and Kriegel 2000].

10. CONCLUSION

We presented two novel partitioning algorithms, the Adaptive Pick-and-Sweep Join (APSJ) and the Adaptive Divide-and-Conquer Join (ADCJ), which allow computing many set containment joins several times more efficiently than the previously known approaches. We provided a detailed analysis of the algorithms and studied their performance using an implemented testbed. We found that APSJ, ADCJ, and the existing algorithm PSJ need to be used complementary for maximal performance. PSJ is the algorithm of choice when the set cardinalities are very small, e.g., below ten elements. For larger cardinalities, APSJ tends to outperform all other algorithms. In some settings, especially in those where the superset relation is much larger than the subset relation, or the element domain is small, ADCJ wins over APSJ and PSJ.

By conducting experiments on real data, we identified a challenge common to all partitioning algorithms, which has been underestimated in the previous work: when the result sizes are large, or many false positives are produced in the joining phase, the partitioning algorithms may become less effective than a naive nested loop join.

The work presented in this paper suggests that set containment joins can be computed quite efficiently when the set element domains are large. It would be interesting to see whether the hash functions used in APSJ and ADCJ can be constructed optimally for small or non-uniform domains, or whether the algorithms presented in this paper reduce the theoretical complexity of containment joins below $O(|R|\cdot|S|)$. Additional performance improvement could be achieved by applying a combination of different partitioning algorithms in several stages, e.g., first ADCJ on disk, then APSJ in memory, or by using a hybrid algorithm suggested in Section 6. Developing efficient algorithms for other set join operators, for instance the intersection join, is another challenging research direction.

REFERENCES

Böhm, C. and Kriegel, H.-P. 2000. Dynamically optimizing high-dimensional index structures. In *Proceedings of Advances in Database Technology - EDBT 2000, 7th International Conference on Extending Database Technology, Konstanz, Germany, March 27-31, 2000*, C. Zaniolo, P. C. Lockemann, M. H. Scholl, and T. Grust, Eds. Lecture Notes in Computer Science, vol. 1777. Springer.

Cai, J.-Y., Chakaravarthy, V. T., Kaushik, R., and Naughton, J. 2001. On the complexity of join predicates. In *PODS'01, Proceedings of the 20th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 21-23, 2001, Santa Barbara, California.* ACM Press.

Faloutsos, C. and Christodoulakis, S. 1984. Signature files: An access method for docu-

ments and its analytical performance evaluation. *ACM Transactions on Information Systems (TOIS) 2,* 4, 267–288.

Gray, J., Sundaresan, P., Englert, S., Baclawski, K., and Weinberger, P. J. 1994. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994*, R. T. Snodgrass and M. Winslett, Eds. ACM Press, 243–252.

Hellerstein, J. M., Koutsoupias, E., and Papadimitriou, C. H. 1997. On the analysis of indexing schemes. In *PODS'97, Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12-14, 1997, Tucson, Arizona*. ACM Press, 249–256.

Helmer, S. and Moerkotte, G. 1997. Evaluation of main memory join algorithms for joins with set comparison join predicates. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, Eds. Morgan Kaufmann, 386–395.

Hirai, J., Raghavan, S., Garcia-Molina, H., and Paepcke, A. 2000. Webbase: A repository of web pages. In *WWW'00, Proceedings of the 9th International World Wide Web Conference*. Computer Networks, vol. 33. Elsevier.

Ishikawa, Y., Kitagawa, H., and Ohbo, N. 1993. Evaluation of signature files as set access facilities in oodbs. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, P. Buneman and S. Jajodia, Eds. ACM Press, 247–256.

Melnik, S. and Garcia-Molina, H. 2002. Divide-and-conquer algorithm for computing set containment joins. In *Proceedings of Advances in Database Technology - EDBT 2002, 8th International Conference on Extending Database Technology, Prague, Czech Republic, March 25-27*, C. S. Jensen, K. G. Jeffery, J. Pokorný, S. Saltenis, E. Bertino, K. Böhm, and M. Jarke, Eds. Lecture Notes in Computer Science, vol. 2287. Springer.

Patel, J. M. and DeWitt, D. J. 1996. Partition based spatial-merge join. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, H. V. Jagadish and I. S. Mumick, Eds. ACM Press, 259–270.

Ramasamy, K., Patel, J. M., Naughton, J. F., and Kaushik, R. 2000. Set containment joins: The good, the bad and the ugly. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang, Eds. Morgan Kaufmann, 351–362.

A. ATOMIC MONOTONE FUNCTIONS AND BIT-STRING TECHNIQUE

Both in APSJ and ADCJ we use monotone boolean hash functions to partition the relations. In this appendix we describe the subclass of the functions used in the analysis of APSJ in more detail and show that the bit-string technique that we use for generating the hash functions in our testbed provides enough functions to use for partitioning in APSJ and ADCJ.

The algorithms APSJ and ADCJ work correctly with arbitrary monotone hash functions (recall that we call h monotone if it satisfies the property that if h fires for set s, it is guaranteed to fire for each superset of s). However, to facilitate the analysis presented in Section 3, we consider a smaller class of functions for which the decision whether h fires for s or not can be made by examining each element of s one by one. We call such functions *atomic*.[10] Each atomic monotone function h can be described as $h(\{x_1, \ldots, x_n\}) = g(x_1) \vee \cdots \vee g(x_n)$, where g is some

[10]If h is viewed as coloring of the lattice formed by the subset relation, each 1-colored node can be traced down to an atom (i.e. set with just one element) over 1-colored nodes.

(not necessarily monotone) boolean function. The firing probability of an atomic monotone function can be determined as follows. Each function g decomposes the domain $\mathcal{D}$ from which the set elements are drawn into two disjoint portions, $\{x \mid g(x) = 1\}$ and $\{x \mid g(x) = 0\}$. The probability that g does not fire for a random set element $x \in \mathcal{D}$ is $p = \frac{\{x|g(x)=0\}}{|\mathcal{D}|}$. If $|\mathcal{D}|$ is much larger than the set cardinality $|s|$, then the probability of drawing an element x with $g(x) = 0$ in each of $|s|$ trials is constant and equals p. That is, the firing probability of h for set s is $P(h(s)) = 1 - p^{|s|}$.

In Section 3 we presented the bit-string technique that we use to construct b atomic monotone hash functions that fire independently of each other with probability $P(h_i(s)) = 1 - p^{|s|} = 1 - (1 - \frac{1}{b})^{|s|}$. Of these b functions, l are selected for partitioning in APSJ. In Section 3.1, we relied on the assumption that the selected functions fire independently of each other to compute the probability that all of them remain silent. The independence assumption is satisfied when $l \ll b \ll |D|$. In worst case, when all of b functions need to be used in APSJ (i.e. $l = b$), the probability that all b functions remain silent for set s is zero (the design of the functions guarantees that at least one bit in the bit string will be set). Assuming independence of the functions, the probability that no function fires is $(1 - \frac{1}{b})^{b \cdot |s|} < e^{-|s|}$. For $|s| = 20$, we have $e^{-20} < 10^{-8}$, which is very close to zero.

When the hash functions are constructed using the bit-string approach, $p = 1 - \frac{1}{b}$. In Section 3.1 we derive the optimal value p_{opt} that minimizes the comparison factor for APSJ. That is, the length of the bit-string that we have to use for generating the optimal hash functions can be computed as $b_{opt} = \frac{1}{1-p_{opt}} = \frac{1}{1-\left(\frac{\lambda}{\lambda+k-1}\right)^{\frac{1}{\theta_R(k-1)}}}$. For example, for $\theta_R = 50$, $\theta_S = 100$, and $k = 64$ partitions, we get $b_{opt} \approx 905$. Since $l = 63 < 905$, we have a sufficient number of functions to choose from. In fact, one can show that for any $k \leq 2^{14}$ and $10 \leq \theta_R \leq \theta_S$, the bit-string approach gives a sufficient number of hash functions to be used by APSJ, i.e., $b_{opt} \geq k - 1$. More generally, for any $k \leq 2^{14}$ and $10 \cdot v \leq \theta_R \leq \theta_S$, we get $b_{opt} \geq v \cdot (k - 1)$, or $k - 1 \leq \frac{b_{opt}}{v}$, i.e., for larger sets the functions fire more independently.

As we show in Appendix C, p_{opt} for ADCJ is determined as $(\frac{\lambda}{1+\lambda})^{\frac{1}{\theta_R}}$. Consequently, the value b_{opt} that minimizes the comparison factor for ADCJ is computed as $b_{opt} = \frac{1}{1-(\frac{\lambda}{1+\lambda})^{\frac{1}{\theta_R}}}$. Again, one can show that for $10 \leq \theta_R \leq \theta_S$, the bit-string approach produces at least 14 functions to choose from, i.e., up to 2^{14} partitions can be used in ADCJ.

B. ANALYSIS OF PSJ

In PSJ, each set r of relation R is assigned to exactly one of partitions $R_1, \ldots, R_k$ based on a randomly chosen element of r. Since the elements of r are drawn uniformly from a large domain, the probability that a given element $x \in r$ yields a certain partition number j is $\frac{1}{k}$. Therefore, each partition R_i will contain on average $\frac{|R|}{k}$ set signatures. In contrast, *all* elements of a set $s \in S$ are used for determining the partition assignment for s. Again, the probability that a given element $x \in s$ yields a certain partition number j is $\frac{1}{k}$. The probability that none of θ_S elements of s yield partition number j is, therefore, $(1 - \frac{1}{k})^{\theta_S}$. Hence, the

probability that at least one of the elements of s triggers the assignment of s to partition S_j is $1-(1-\frac{1}{k})^{\theta_S}$. In other words, the average number of signatures in each partition S_j is $(1-(1-\frac{1}{k})^{\theta_S})\cdot|S|$.

Each pair of partitions $R_j \bowtie S_j$ results in $|R_j|\cdot|S_j| = \frac{|R|}{k}\cdot(1-(1-\frac{1}{k})^{\theta_S})\cdot|S|$ signature comparisons. Multiplying this number by k and dividing by $|R|\cdot|S|$ produces the comparison factor $comp_{\text{PSJ}} = 1-\left(1-\frac{1}{k}\right)^{\theta_S}$. The replication factor is the sum of all signatures in all partitions divided by $|R|+|S|$. Hence, we obtain $repl_{\text{PSJ}} = \frac{|R|+k(1-(1-\frac{1}{k})^{\theta_S})\cdot|S|}{|R|+|S|} = \frac{|R|}{|R|+|S|} + \frac{|S|}{|R|+|S|}k(1-(1-\frac{1}{k})^{\theta_S}) = \frac{1}{1+\rho} + \frac{\rho}{1+\rho}k(1-(1-\frac{1}{k})^{\theta_S})$. Notice that both the comparison and replication factor grow with the sizes of sets in S. For large θ_S, $comp_{\text{PSJ}} \approx 1$, making PSJ ineffective.

C. COMPARISON FACTOR OF ADCJ

To partition relations R and S, ADCJ utilizes the operators α and β. Operator α repartitions $R \bowtie S$ into $(R/h \bowtie S/h) \cup (R/\neg h \bowtie S)$ using a boolean hash function h. That is, the number of comparisons required after applying α is $|R|\cdot|S|\cdot\big(P(h(r))\cdot P(h(s)) + P(\neg h(r))\cdot 1\big)$. Since $P(h(r)) = 1-p^{|r|}$, and $P(h(s)) = 1-p^{|s|}$, applying α reduces the number of comparisons to $|R|\cdot|S|\cdot\big((1-p^{|r|})(1-p^{|s|})+p^{|r|}\big) = |R|\cdot|S|\cdot(1-p^{|s|}+p^{|r|+|s|})$.

Analogously, β repartitions $R \bowtie S$ into $(R/\neg h \bowtie S/\neg h) \cup (R \bowtie S/h)$. Hence, the number of comparisons required after applying β is $|R|\cdot|S|\cdot\big((1-P(h(r)))\cdot(1-P(h(s))) + 1\cdot P(h(s))\big) = |R|\cdot|S|\cdot(1-p^{|s|}+p^{|r|+|s|})$. That is, by using a hash function h, both α and β reduce the number of comparisons by a factor of $(1-p^{|s|}+p^{|r|+|s|})$. Since each pair of partitions $R_j \bowtie S_j$ in assignments $1,\ldots,l$ is obtained by using either α or β, the resulting comparison factor is $comp_{\text{ADCJ}} = \prod_{i=1}^{l}(1-p_i^{|s|}+p_i^{|r|+|s|})$.

We can minimize the comparison factor by designing the hash functions in a certain 'optimal' way. The expression $\prod_{i=1}^{l}(1-p_i^{|s|}+p_i^{|r|+|s|})$ is minimized when $p_1 = p_2 = \cdots = p_l = \left(\frac{|s|}{|r|+|s|}\right)^{\frac{1}{|r|}}$, i.e., each of h_i fires with the same probability. If all sets of R and S have cardinalities θ_R and θ_S, we obtain $p_i = \left(\frac{\theta_S}{\theta_R+\theta_S}\right)^{\frac{1}{\theta_R}}$. Using the cardinality ratio $\lambda = \frac{\theta_S}{\theta_R}$, p_i can be rewritten as $p_i = (\frac{\lambda}{1+\lambda})^{\frac{1}{\theta_R}}$. Furthermore, since $k = 2^l$, we have $l = \log_2 k$. Consequently, we obtain

$$comp_{\text{ADCJ}} = \prod_{i=1}^{\log_2 k}\left(1-\left(\frac{\lambda}{1+\lambda}\right)^{\frac{\theta_S}{\theta_R}}+\left(\frac{\lambda}{1+\lambda}\right)^{\frac{\theta_R+\theta_S}{\theta_R}}\right) = \left(1-\frac{1}{1+\lambda}\left(\frac{\lambda}{1+\lambda}\right)^{\lambda}\right)^{\log_2 k}$$

D. JOIN SELECTIVITY IN THE ANALYTICAL MODEL

Let the elements of sets in relations R and S be drawn uniformly from the domain $\mathcal{D}$, and the cardinalities of the sets in R and S be θ_R and θ_S, respectively. For two relations R and S, the selectivity of the join $R \bowtie S$ is the fraction of elements in the cross-product $R \times S$ that participate in the join. In other words, the join selectivity is the probability that $r \subseteq s$ holds for any two randomly chosen sets $r \in R$ and $s \in S$. Imagine that we are examining each element of r, one after another, in a

random order. For the first element that we select, the probability that we picked one specific element of s is that of picking a specific element of $\mathcal{D}$, i.e., $\frac{1}{|\mathcal{D}|}$. Since $|s| = \theta_S$, the probability that we picked one of the elements of s is $\frac{\theta_S}{|\mathcal{D}|}$. Now consider the second element of r. The probability that we picked another specific element of s is now $\frac{1}{|\mathcal{D}|-1}$. Hence, the probability that as a second element we picked one of the remaining $\theta_S - 1$ elements of s is $\frac{\theta_S-1}{|\mathcal{D}|-1}$. Analogously, for the third element of r, we get $\frac{\theta_S-2}{|\mathcal{D}|-2}$. Continuing this line of reasoning for all θ_R elements of r, we obtain $P(r \subseteq s)$ as a product of the probabilities that each element of r belongs to s, i.e., $P(r \subseteq s) = selectivity(|\mathcal{D}|, \theta_R, \theta_S) = \frac{\theta_S}{|\mathcal{D}|} \cdot \frac{\theta_S-1}{|\mathcal{D}|-1} \cdots \frac{\theta_S-\theta_R+1}{|\mathcal{D}|-\theta_R+1} = \frac{\theta_S!(|\mathcal{D}|-\theta_R)!}{(\theta_S-\theta_R)!|\mathcal{D}|!}$ (for $\theta_R \leq \theta_S$).

E. ACCURACY OF ANALYTICAL MODEL

In this appendix we study the accuracy of the formulas for comparison and replication factors of APSJ and ADCJ (see Table VII) for different set cardinality and set element distributions. The experiments described below are not implementation specific, and depend just on the content of relations R and S. We used five different distributions of element values, and five distributions of set cardinalities. These distributions are summarized in Table VIII (values generated from these distributions are rounded to get the discrete distributions we need). Starting with the distributions that are close to the assumptions of our analytical model, we gradually make them more and more distinct. For example, in case A the set elements are drawn uniformly from the domain $\mathcal{D} = \{0, \ldots, 10000\}$. In other words, the element distribution has the mean of 5000 and the standard deviation[11] of $\frac{10000}{\sqrt{12}} \approx 2886$. The cardinalities of sets in R are drawn uniformly from $\{45, \ldots, 55\}$, whereas the cardinalities in S are drawn from $\{90, \ldots, 110\}$. Thus, case A is relatively close to our assumptions that $\mathcal{D}$ is large and the sets in R and S have fixed cardinalities. In contrast, case D illustrates a scenario in which the element values obey a normal (Gaussian) distribution with standard deviation $\sigma = 100$. In other words, 95% of element values are contained in the interval $[\mu - 2\sigma, \mu + 2\sigma] = \{4800, \ldots, 5200\}$. In case E, the element value domain is limited to just 200 elements. From A to E, we gradually increase the variance of the cardinality distributions, culminating in uniform distributions $\{0, \ldots, 100\}$ for S and $\{0, \ldots, 200\}$ for R.

Figures 25 and 26 illustrate the impact of the distributions used in cases A–E on the predictions of our formulas. The graphs show the individual impact of varying just the element distribution, or just the set cardinality distributions, or both. For example, the bottom curve in Figure 25 (labeled 'APSJ cardinality') illustrates how the actual comparison factor for APSJ becomes less accurate when we vary the cardinality distributions and keep the element distribution uniform with $\mathcal{D} = \{0, \ldots, 10000\}$. For each data point, we generated the test relations ten times with $|R| = 2000$, $|S| = 10000$ and determined the average actual comparison and replication factors for $k = 128$. The curve ('APSJ element') shows how the comparison factor for APSJ deviates from the predicted value when we vary the element distribution and keep the set cardinalities constant at $\theta_R = 50$, $\theta_S = 100$.

[11] The standard deviation of a uniform distribution over domain $[a, b]$ is computed as $\frac{b-a}{\sqrt{12}} \approx \frac{b-a}{3.46}$.

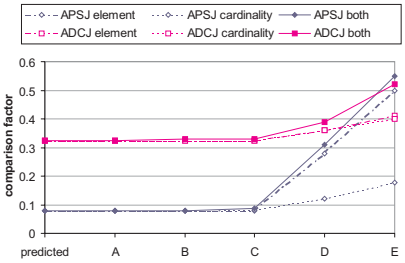

Fig. 25. Impact of distributions on comparison factor

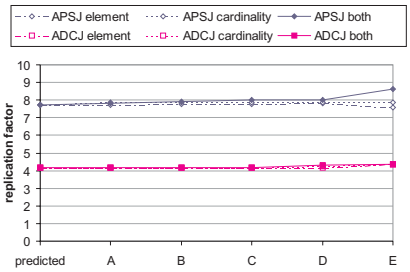

Fig. 26. Impact of distributions on replication factor

The solid curves illustrate the combined impact of varying both element and set cardinality distributions.

Notice that the replication factor for APSJ and ADCJ matches the predicted values accurately under all distributions that we use. The predictions for the comparison factors are precise for the cases A, B, and C. However, the negative impact of the element distribution on comparison factors becomes significant when the domain size $|\mathcal{D}|$ approaches the average set cardinalities θ_R and θ_S. For example, in case E the actual value of $comp_{\text{APSJ}} \approx 0.55$ exceeds by far the predicted value of $comp_{\text{APSJ}} \approx 0.08$. In fact, $comp_{\text{APSJ}}$ becomes even larger than $comp_{\text{ADCJ}} \approx 0.52$, much to the contrary of our prediction. In many scenarios we observed that the performance of APSJ degrades significantly faster with shrinking element domains than that of ADCJ. As we demonstrate in [Melnik and Garcia-Molina 2002], PSJ is even less sensitive to varying distributions. Nevertheless, the gains of APSJ and ADCJ often compensate for the increased number of comparisons that is due to varying distributions. For example, in case E, we obtain $comp_{\text{PSJ}} \approx 0.61$, $repl_{\text{PSJ}} \approx 63$, i.e., in this scenario APSJ and ADCJ outperform PSJ with respect to both efficiency measures.

As a final remark, notice that the selectivity of the joins rapidly increases from case A to case E. For instance, for constant θ_R and θ_S and the element distribution of case A, we obtain the selectivity of $3.4 \cdot 10^{-107}$ using the formula of Appendix D. In contrast, in case E (with constant θ_R and θ_S) we get a selectivity of $2.2 \cdot 10^{-19}$, which is larger by many orders of magnitude. Experimentally, we determined that the selectivities in cases D and E (varying both element and cardinality distributions) are $7.3 \cdot 10^{-5}$ and $3.6 \cdot 10^{-2}$, respectively. When the join selectivity is high, the execution time of either algorithm is dominated by the retrieval of the joining tuples. Thus, the prediction accuracy of the comparison and replication factors may be a less critical issue.

We did several additional experiments with different partition numbers and relation sizes, which we omit here for brevity. Across all experiments we observed that APSJ and ADCJ tend to be more negatively affected by varying the distributions than PSJ. As we explained in Section 4.2, this effect is mainly attributed to problems with the generation of the boolean hash functions. In summary, we conclude that for a variety of set cardinality distributions the formulas of Table VII (including Algorithm 1 for ADCJ) deliver relatively accurate predictions that lie

within 15% of the actual values, as long as the element domains are at least 10 times larger than the average set cardinalities and a large number of domain elements is used in the sets.

F. CHOOSING NUMBER OF PARTITIONS WHICH IS NOT POWER OF TWO

Recall that ADCJ can make effective use of k partitions only if k is a power of two. Hence, ADCJ 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 13 we can see that the execution time of ADCJ 'flattens out' in the interval $128 \leq k \leq 1024$. In other words, the inability to choose say $k = 200$ does not cripple the performance of ADCJ. Furthermore, the limitation in choosing k can be addressed using the modulo approach suggested in [Helmer and Moerkotte 1997]. To illustrate, for using $k = 57$ partitions in the above example, we compute the partition assignment just as for $k = 2^{\lceil \log_2 57 \rceil} = 64$, and use ($k \mod 57$) to map each partition number into the interval $0, \ldots, 56$. We tested this approach and found that it is effective only for those values of k that lie close to the next larger power of two, e.g., $k \geq \frac{1}{4}2^{\lfloor \log_2 k \rfloor} + \frac{3}{4}2^{\lceil \log_2 k \rceil}$. That is, in the above example, we could effectively use ADCJ for partition numbers between (around) 56 and 64. In contrast, for k between 33 and 55, the partition assignments produced by ADCJ are skewed by applying modulo, and the algorithm performs even worse than for $k = 32$. In summary, using the modulo approach allows us to extend our search for best k from points (powers of two) to larger intervals.

G. ALGORITHMIC SPECIFICATION OF ADCJ AND APSJ

Each of the algorithms that we discussed implements a different partitioning function π. Recall that a partitioning function assigns each set of relation R to one or multiple partitions $R_1, \ldots, R_k$, and each set of S to one or multiple partitions $S_1, \ldots, S_k$. The partitioning functions for APSJ and ADCJ are specified in Algorithm 2 and Algorithm 3, respectively. The algorithms are simple enough so that we use Java notation directly instead of pseudo-code. In each algorithm, the partitioning function is called `mapSetToPartitions`. The function takes three parameters, a bit vector `partitions`, a set of integers `set`, and a relation identifier `relation`. The bit vector is used to return the partition assignment computed for the given `set` of integers. The relation identifier determines whether the `set` originates from relation R or S.

```
double replADCJ(double lambda, int k, double rho) {

  double pr = 1 / (1 + lambda);
  double ps = 1 - Math.pow(lambda / (1 + lambda), lambda);
  return computeReplADCJ(k, pr, ps, 1.0, rho) / (1.0 + rho);
}

double computeReplADCJ(int i, double pr, double ps,
                               double replR, double replS) {
  if(i <= 0)
    return replR + replS;
  if(replR >= replS)
    return computeReplADCJ(i-1, pr, ps, replR * pr, replS * ps) +
           computeReplADCJ(i-1, pr, ps, replR * (1-pr), replS);
  else
    return computeReplADCJ(i-1, pr, ps, replR * (1-pr), replS * (1-ps)) +
           computeReplADCJ(i-1, pr, ps, replR, replS * ps);
}
```

Algorithm 1: Algorithm for estimating the replication factor for ADCJ

```
void mapSetToPartitions(BitVector partitions,
                                /* holds resulting partition assignment */
                        int[] set,
                                /* a set to be assigned to partitions   */
                        int relation) { /* set is from relation R or S */

  // sig is computed as:
  // sig = (int)Math.round(1.0/(1.0 - Math.pow(lambda/(lambda+k-1),
  //                                           1.0/(k-1)/avgR)));
  switch(relation) {
    case R:
      // randomly find a firing hash function using set elements
      for(int j=0; j < set.length; j++) {
        int p = hash(set[j]) % sig;  // hash() is some simple hash function
        if(p < k-1) { partitions.set(p + 1); return; }
      }
      partitions.set(0); // insert into default partition if none fires
      break;
    case S:
      // determine target partitions using all set elements
      for(int j=0; j < set.length; j++) {
        int p = hash(set[j]) % sig;
        if(p < k-1) { partitions.set(p + 1); }
      }
      partitions.set(0); // default partition contains all of S
  }
}
```

Algorithm 2: Adaptive Partitioning Set Join (APSJ) algorithm (optimized for hash functions based on bit-strings)

```
void mapSetToPartitions(BitVector partitions,
                                /* holds resulting partition assignment */
                        int[] set,
                                /* a set to be assigned to partitions   */
                        int relation) { /* set is from relation R or S */

  // given: lambda is the ratio of average set cardinalities
  //        rho    is the relation size ratio
  pr = 1 / (1 + lambda);
  ps = 1 - Math.pow(lambda / (1 + lambda), lambda);
  // start recursion with hash fct index 0 and partNo=0
  computeMap(partitions, offset, 0, set, relation, 0, rho);
}

void computeMap(BitVector partitions, int i /* index of hash fct */,
                       int[] set, int relation, int partNo, double ratio) {

  if(i >= HASH_FCT_NUM) {   // HASH_FCT_NUM = log(k)
    partitions.set(partNo); // set bit number partNo in partition vector
    return;
  }
  boolean h = h(i, set); // compute i-th boolean hash function
  if(ratio <= 1.0 && h)
    computeMap(partitions, i+1, set, relation,
                                partNo, ratio * ps / pr);
  if(ratio > 1.0 && !h)
    computeMap(partitions, i+1, set, relation,
                                partNo, ratio * (1-ps) / (1-pr));
  if(ratio <= 1.0 && (relation == S || !h))
    computeMap(partitions, i+1, set, relation,
                                partNo | (1 << i), ratio / (1-pr));
  if(ratio > 1.0 && (relation == R || h))
    computeMap(partitions, i+1, set, relation,
                                partNo | (1 << i), ratio * ps);
}
```

Algorithm 3: Adaptive Divide-and-Conquer Join (ADCJ) algorithm

Received January 2002; revised November 2002; accepted February 2003