Evaluation of Main Memory Join Algorithms for Joins with Subset Join Predicates

Sven Helmer
helmer@pi3.informatik.uni-mannheim.de

Guido Moerkotte
moer@pi3.informatik.uni-mannheim.de

Fakultät für Mathematik und Informatik, University of Mannheim, Germany

Abstract

Current data models like the NF^2 model and object-oriented models support set-valued attributes. Hence, it becomes possible to have join predicates based on set comparison. This paper introduces and evaluates two main memory algorithms to evaluate efficiently this kind of join. More specifically, we concentrate on subset predicates.

1 Introduction

Since the invention of relational database systems, tremendous effort has been undertaken in order to develop efficient join algorithms. Starting from a simple nested-loop join algorithm, the first improvement was the introduction of the merge join [1]. Later, the hash join [2, 7] and its improvements [20, 23, 28, 39] became alternatives to the merge join. For overviews see [27, 37] and for a comparison between the sort-merge and hash joins see [13, 14].

A lot of effort has also been spent on parallelizing join algorithms based on sorting [10, 25, 26, 34] and hashing [6, 12, 36]. Another important research area is the development of index structures that allow to accelerate the evaluation of joins [16, 22, 21, 29, 40, 42].

All of these algorithms concentrate on simple join predicates based on the comparison of two atomic values. Predominant is the work on equi-joins, i.e., where the join predicate is based on the equality of atomic values. Only a few articles deal with special issues like non-equi-joins [9], non-equi-joins in conjunction with aggregate functions [5], and pointer-based joins [8, 38]. An area where more complex join predicates occur is that of spatial database systems. Here, special algorithms to support spatial joins have been developed [3, 15, 18, 24, 30].

Despite this large body of work on efficient join processing, the authors are not aware of any work describing join algorithms for the efficient computation of the join if its predicate is based on set comparisons like set equality ($=$) or subset equal ($\subseteq$). These joins were irrelevant in the relational context since attribute values had to be atomic. However, newer data models like NF^2 [32, 35] or object-oriented models like the ODMG-Model [4] support set-valued attributes, and many interesting queries require a join based on set comparison. Consider for example the query for faithful couples. There, we join persons with persons in condition of the equality of their children attributes. Another example query is that of job matching. There, we join job offers with persons such that the set-valued attribute *required-skills* is a subset of the persons' set-valued attribute *skills*. We could give plenty of more queries involving joins based on set comparisons but we think these suffice for motivation. Note that all the examples work on possibly large sets of objects, but with limited set cardinality. We believe that this is the most common case found in practice. This belief is backed by our observations on real applications for object-oriented databases.

The rest of the paper is organized as follows. In the next section, we introduce some basic notions needed in order to develop our join algorithms. Sections 3 introduces and evaluates several join algorithms where the join predicate is subset equal. Algorithms for other join predicates can be found in the full paper [17]. Section 4 concludes the paper.

Proceedings of the 23rd VLDB Conference
Athens, Greece, 1997

2 Preliminaries

2.1 General Assumptions

For the rest of the paper, we assume the existence of two relations R_1 and R_2 with set-valued join attributes a and b. We do not care about the exact type of the attributes a and b – that is whether it is e.g. a relation, a set of strings, or a set of object identifiers. We just assume that they are sets and that their elements provide an equality predicate.

The goal of the paper is to compute efficiently the join expression

$$R_1 \bowtie_{a \subseteq b} R_2$$

More specifically, we introduce a join algorithm based on hashing and compare its performance with a nested-loop strategy. Two other join algorithms, sort-merge and a tree-based one, and set equality predicates are described in [17].

For convenience, we assume that there exists an (injective) function m which maps each element within the sets of $R_1.a$ and $R_2.b$ to the domain of integers. The function m is dependent on the elements' type of the set-valued attributes. For integers, the function is identity, for strings and other types, techniques like folding can be used. From now on, we assume without loss of generality that the type of the sets is integer. If this is not the case, the function m has to be applied before we do anything else with the set elements.

2.2 Set Comparison

The costs of comparing two sets by $\subseteq$ differ significantly depending on the used algorithm. Hence, we first discuss some of the alternatives for comparing sets. Consider the case in which we want to evaluate $s \subseteq t$ for two sets s and t. We could check whether each element in s occurs in t. If t is implemented as an array or list, then this algorithm takes $O(|s| \cdot |t|)$. For small sets, this might be a reasonable strategy. For large sets, however, the comparison cost of this simple strategy can be significant. Hence, we consider further alternatives for set comparison.

One obvious alternative to evaluate efficiently $s \subseteq t$ is to have a search tree or a hash table representation of t. Since we assume that the representation of set-valued attributes is not of this kind but instead consists in a list or an array of elements, we abandon this solution since the memory consumption and cpu time needed in order to construct this indexed representations are too expensive in comparison to the methods that follow.

Another alternative to implement set comparison is based on sorting the elements. Assuming an array representation of the elements of the set, we denote the i-th element of a set s by $s[i]$. We start with comparing the smallest elements. If $s[0]$ is smaller than $t[0]$, there is no chance to find $s[0]$ somewhere in t. Hence, the result will be false. If $s[0]$ is greater than $t[0]$, then we compare $s[0]$ with $t[1]$. In case $s[0] = t[0]$, we can start comparing $s[1]$ with $t[1]$. The following algorithm implements this idea:

```
if(s->setcard > t->setcard) return false;
i=j=0;
while(i < s->setcard && j < t->setcard)
  if(s[i] > t[j])       j++;
  else if (s[i] < t[j]) return false;
  else /*  (s[i] == t[j]) */
      { i++; j++; }
if(i==s->setcard) return true;
return false;
```

The first line implements a pretest based on cardinality. The remaining lines implement the idea outlined above. The run time of the algorithm is $O(|s| + |t|)$. Again, since we do not assume that the sets are sorted, we have a run time complexity of $O(|s| \log |s| + |t| \log |t|)$.

The third alternative we considered for implementing set comparisons is based on signatures. This algorithm first computes the signature of each set-valued attribute and then compares the signatures before comparing the actual sets using the naive set comparison algorithm. This gives rise to a run time complexity of $O(|s| + |t|)$. Signatures and their computation are the subjects of the next section. Furthermore, the next section introduces some basic results that will be needed for tuning some of the hash join algorithms.

2.3 Signatures

2.3.1 Introduction

A signature is a bit field of a certain length b–called the signature length. Signatures are used to represent or approximate sets. For our application, it suffices if we set one bit within the signature for each element of the set whose signature we want to compute. Assuming a function m_{sig} that maps each set element to an integer in the interval $[0, b[$, the signature can be computed by successively setting the $m_{sig}(x)$-th bit for each element x in the set. For a set s, let us denote the signature of s by $sig(s)$.

Similar to hashing, we cannot assume that the bits set for the elements of a set are really distinct. But still, the following property holds:

$$s \subseteq t \implies sig(s) \subseteq sig(t)$$

where $sig(s) \subseteq sig(t)$ is defined as

$$sig(s) \subseteq sig(t) := sig(s) \& \neg sig(t) = 0$$

with & denoting *bitwise and* and $\neg$ denoting *bitwise complement*. Hence, a pretest based on signatures can be very fast since it involves only bit operations. Again, we do not assume that the elements of the sets of the relation's tuples contain their signatures, all the subsequent join algorithms which use signatures have to construct them. Hence, their cost will highly depend on the quality of the algorithm used to implement m_{sig}. Here, we can measure the quality by the probability that the reverse direction of the above implications do not hold. Such a case is called a *false drop*. The probability of false drops is calculated in the next subsection.

The function m_{sig} can be implemented in several different ways. We investigated two principle approaches. The first approach uses a random number generator whose seed is the set element. The resulting number gives the bit to be set within the signature. In our implementation, we used two different random number generators: *rand*() of the C-library and *DiscreteUniform* of the GNU-library. The second approach just takes the set element modulo the signature length. The advantage of the former is a possible slight reduction of the false drop probability, the advantage of the latter is a much better run time.

2.3.2 False Drop Probability

Consider two sets s and t and their signatures $sig(s)$ and $sig(t)$. If for a predicate θ $sig(s)\theta sig(t)$, we call this a *drop*. If additionally $s\theta t$ holds, we call this a *right drop*. If $sig(s)\theta sig(t)$ and $\neg(s\theta t)$, we call this a *false drop*. False drops exist, because by hashing the data elements and superimposing their signatures, it is possible that two different sets are mapped onto the same signature.

The false drop probabilities for subset predicates have been studied [11, 19, 31, 33] and can be approximated by the following general formula [19]:

$$d_{f\subseteq}(b,k,r_s,r_t) \quad \approx \quad (1-e^{-\frac{k}{b}r_t})^{k\cdot r_s} \qquad (1)$$

Here, b denotes the signature length, k the number of bits set per set element (in this paper, we will assume $k = 1$), $r_s = |s|$ the cardinality of the set on the left side of the predicate $s\theta t$, and $r_t = |t|$ the cardinality of the set t on the right side.

3 Join Predicates with $\subseteq$

This section discusses algorithms to compute

$$R_1 \bowtie_{R_1.a\subseteq R_2.b} R_2$$

for two relations R_1 and R_2 with set-valued attributes a and b. Obviously, these algorithms will also be useful for computing joins like $R_1 \bowtie_{R_1.a\supseteq R_2.b} R_2$, $R_1 \bowtie_{R_1.a\subset R_2.b} R_2$, and $R_1 \bowtie_{R_1.a\supset R_2.b} R_2$. For the latter two only slight modifications are necessary. This section is organized as follows. The next subsection contains a description of the nested-loop join and the hash join. The last subsection covers the evaluation of these join algorithms.

3.1 Algorithms

3.1.1 Nested-Loop Joins

There are three different possible implementations of the nested-loop join. Each alternative is based on a different implementation of the $\subseteq$ predicate. The first alternative applies the naive implementation, the second applies the implementation based on sorting, and the third alternative utilizes signatures. For details on the different implementations for $\subseteq$ see Sec. 2.2.

For space reasons, we cannot give the performance evaluation of the different nested-loop join algorithms (see [17]) but we repeat the most important conclusion that we draw from the evaluation: the signature-based set comparison performs best and the naive implementation of $\subseteq$ performs worst. Hence, the signature-based variant will be used for further comparison.

We give a rough estimation of the running time of the signature-based nested-loop join algorithm. Let C_{sig} be the costs to compare two signatures, $C_{set}(\overline{r})$ the costs for comparing two sets ($\overline{r}$ stands for the average set cardinality), $C_{create}(\overline{r})$ the costs for creating a signature, and P_{hit} the probability that two (arbitrary) signatures taken from R_1 and R_2 match. P_{hit} is equal to the sum of the selectivity between R_1 and R_2 and the false drop probability $d_{f\subseteq}$. Note that costs $C_{create}(\overline{r})$ are linear in $\overline{r}$. The complexity of $C_{set}(\overline{r})$ depends on the algorithm that is used (cf. section 2.2). We neglected the costs for the result construction, since they are the same for all join algorithms. Then the costs for the hash join are

$$\begin{aligned} C_{NL} \quad \approx \quad & (|R_1| + |R_2|) \cdot C_{create}(\overline{r}) + \\ & |R_1| \cdot |R_2| \cdot C_{sig} + \\ & |R_1| \cdot |R_2| \cdot P_{hit} \cdot C_{set}(\overline{r}) \qquad (2) \end{aligned}$$

The first term reflects the creation costs of the signatures for all tuples in R_1 and R_2. The second term accounts for the comparison costs of every signature of the tuples in R_1 with every signature of the tuples in R_2. The third term incorporates the testing of all drops encountered. For small values of P_{hit} C_{NL} is characterized by the term $|R_1| \cdot |R_2| \cdot C_{sig}$.

3.1.2 Signature-Hash Join

Using signatures to hash sets seems easy for evaluating joins based on set equality. For subset predicates this is not so obvious. In principle, two different approaches exist. First, we could redundantly insert every tuple with a set-valued attribute a into the hash table for every possible subset of a. Since this results in a tremendous storage overhead, we abandoned this solution. The second approach inserts every tuple once into the hash table. Now the problem is to retrieve for a given tuple all those tuples whose set-valued attribute is a subset of the given tuple. More specifically, given a signature of the set-valued attribute of a tuple, the question is how to generate the hash keys for all subsets. This problem can be solved by restricting the hash functions used. A hash function has *degree n*, if it always produces a bit pattern of length n. Let h be a hash function of degree n. A k-restriction is a bit pattern of length n where at most k bits are set. We call a hash function of degree n *$\subseteq$-capable* if for all sets s, all its subsets t, and all k-restrictions the following holds:

$$s \subseteq t \implies (k \& h(s)) \& \neg(k \& h(t)) = 0.$$

Note that signatures are $\subseteq$-capable.

Assuming a $\subseteq$-capable hash function h, we can step quickly through all the hash values of the subsets of a given set s by an algorithm applied by Vance and Meier in their blitzsplit join ordering algorithm [41]:

```
a = h(s);
s = a & -a;
while(s) {
   s = a & (s - a);
   process(s); }
```

Although the inner part of the loop encompasses only a small number of machine instructions, it is executed exponentially often in the number of bits set within the hash value $h(s)$. This suggests to keep the number of bits set small. However, for signatures this would result in a high false drop probability. The problem can be avoided by using only part of the signature for the above subset traversal algorithm. Assuming a $\subseteq$-capable hash function this poses no difficulties, since for example truncation to the last d bits is possible.

In order to derive a hash join for the subset equal predicate, we proceed in several steps:

1. building the hash table (including the computation of the hash values)
2. tuning the parameters
3. the actual join algorithm
4. fine tuning of the algorithm
5. discussion of implementation details

Building the hash table.

We consider two alternative approaches. In the direct approach, we consider a signature of length b, but take only the lowest d bits as a direct pointer into the hash table. (The reasons for this become obvious in the next step.) From this it follows that the hash table must have a size of 2^d. Let us denote the lowest d bits of the signature of a set-valued attribute a of some tuple t by $partsig(t)$.[1] Every tuple t of the relation to be hashed is now inserted into the hash table at address $partsig(t)$.

Tuning the parameters.

The signature size b and the partial signature size d are crucial tuning parameters, since they determine the number of hash table entries and the collision chain lengths, which in turn have an impact on the running time of the join algorithm. If b becomes very small, i.e. much smaller than the set cardinality, then we can expect every bit to be set. Hence, many collisions occur. If b becomes very large, say 1,000 times the set cardinality, we can expect that almost none of the d bits of the partial signature is set. Again, many collisions occur. Hence, both extremes collect all tuples in one collision chain.

The problem is now to determine the optimal value of b. Our partial signature contains the most information content possible, if on the average $d/2$ bits are set. Since we assume that the bits which are set are distributed uniformly over the signature, this amounts to determine the partial signature size so that half of the bits are set in the full signature.

In section 2.3.2, we gave the formula to determine the number of bits set as

$$w = b(1 - (1 - \frac{1}{b})^r)$$

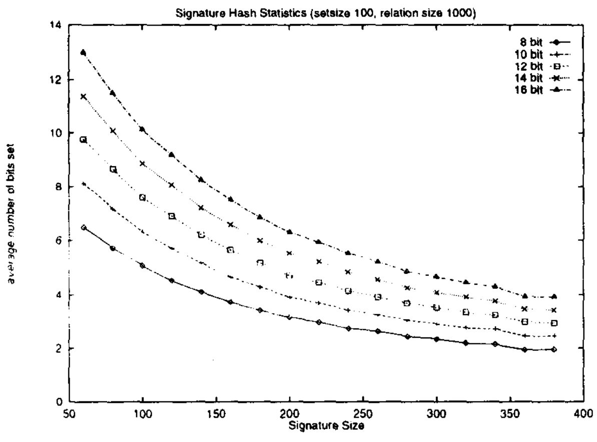
where b is the signature size and r is the (average) set cardinality. For a given set cardinality, the problem is now to determine the optimal value for b. This b can be derived analytically as

$$b_{opt} = \frac{1}{1 - (\frac{1}{2})^{\frac{1}{r}}} \qquad (3)$$

A formal derivation of this result can be found in [17].

Hence, if we determine b_{opt} by this formula, we can expect that half of our d bits in the partial signature will be set and that the collision chain length is minimal. Let us give at least the results of one experiment validating this analytical finding. For a set cardinality of 100, the optimal value of b can be computed as

[1]For computing the signature and the partial signature, we consider only the modulo approach, since the approaches using random number generators are too slow.



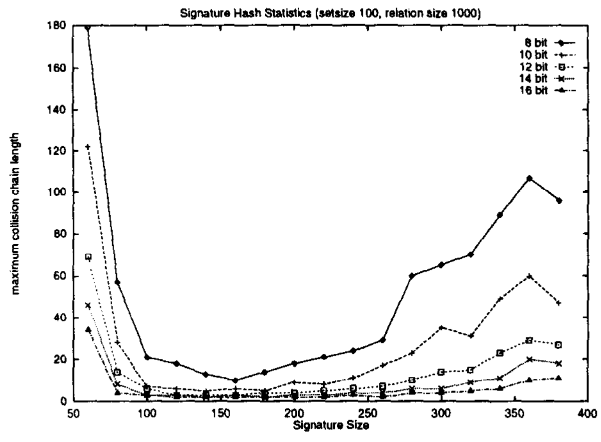


Figure 1: Tuning partial signature sizes

$b_{opt} = 144.77$. Figure 1 gives the experimental results. On the left-hand side, the number of bits set in the partial signature depending on the signature size is given. Each curve corresponds to a different partial signature size d. It follows that the according hash table has 2^d entries. For $d = 8$, we find that for some number a little smaller than 150, 4 bits are set. This is exactly what we expect after our analysis. On the right-hand side, the maximum collision chain lengths for different partial signature lengths are given. We find that for values around 145, the collision chain lengths become minimal. Hence, we can use formula 3 to tune our hash tables. Further, we find that the accurate value for the partial signature size is not too important. In order to be very close to the minimum collision chain length, a value from 130-200 will do.

The signature-hash join.

After having built hash tables with reasonably short collision chains, we come to the problem of computing the join $R_1 \bowtie_{R_1.a \subseteq R_2.b} R_2$. We first transform this problem into computing the join of $R_2 \bowtie_{R_2.b \supseteq R_1.a} R_1$. Then, the hash table is constructed for R_1. The last step consists in finding all the joining tuples of R_1 for each tuple in R_2. For a given tuple $t_2 \in R_2$, we have to find all the tuples t_1 in R_1 such that $t_2 \supseteq t_1$. We do so by generating with the algorithm above—all the partial signatures for all subsets of t_2 and look for these partial signatures within the hash table. For all tuples found, we perform (1) a comparison of the full signature and, if necessary, (2) an explicit set comparison to take care of false drops.

We refer to the approach where the hash table size is 2^d for a given partial signature size by the *direct approach* since the signatures are directly used as hash keys. However, we did not want to be bound to hash table sizes of 2^d only. Hence, we added a modulo computation in order to allow for arbitrary hash table sizes. Let n be the chosen hash table size. Then, the tuples t are inserted into the hash table at address $partsig(t) \bmod n$. For retrieving the joining tuples, we apply also the *mod* n to the partial signatures S of the subsets. We refer to this *indirect approach* by *mod* and to the direct approach by *dir*.

Theoretical analysis and fine tuning

Let us now have a closer look at the tuning parameters. Obviously, the signature length and the partial signature length heavily influence the performance of the hash join. The shorter the partial signature, the longer are the collision chains (witness Fig. 1). The longer the partial signature, the more bits are set within it. Hence, the more hash table entries have to be checked for each tuple. On the average $2^{\frac{d}{2}}$ entries have to be checked if we set the signature length to b_{opt}. Hence, a large d gives small collision chains but results in many hash table lookups.

Let us formalize these thoughts. If we assume that the signature values are uniformly distributed, the average length of a collision chain is equal to

$$\bar{l}_{coll} = \frac{|R_1|}{2^d} \tag{4}$$

During join processing we have to look up all matching signatures in R_1, i.e. all signatures which are a subset of $t_2.b$ for every tuple $t_2 \in R_2$. As previously mentioned, after optimizing the signatures we expect

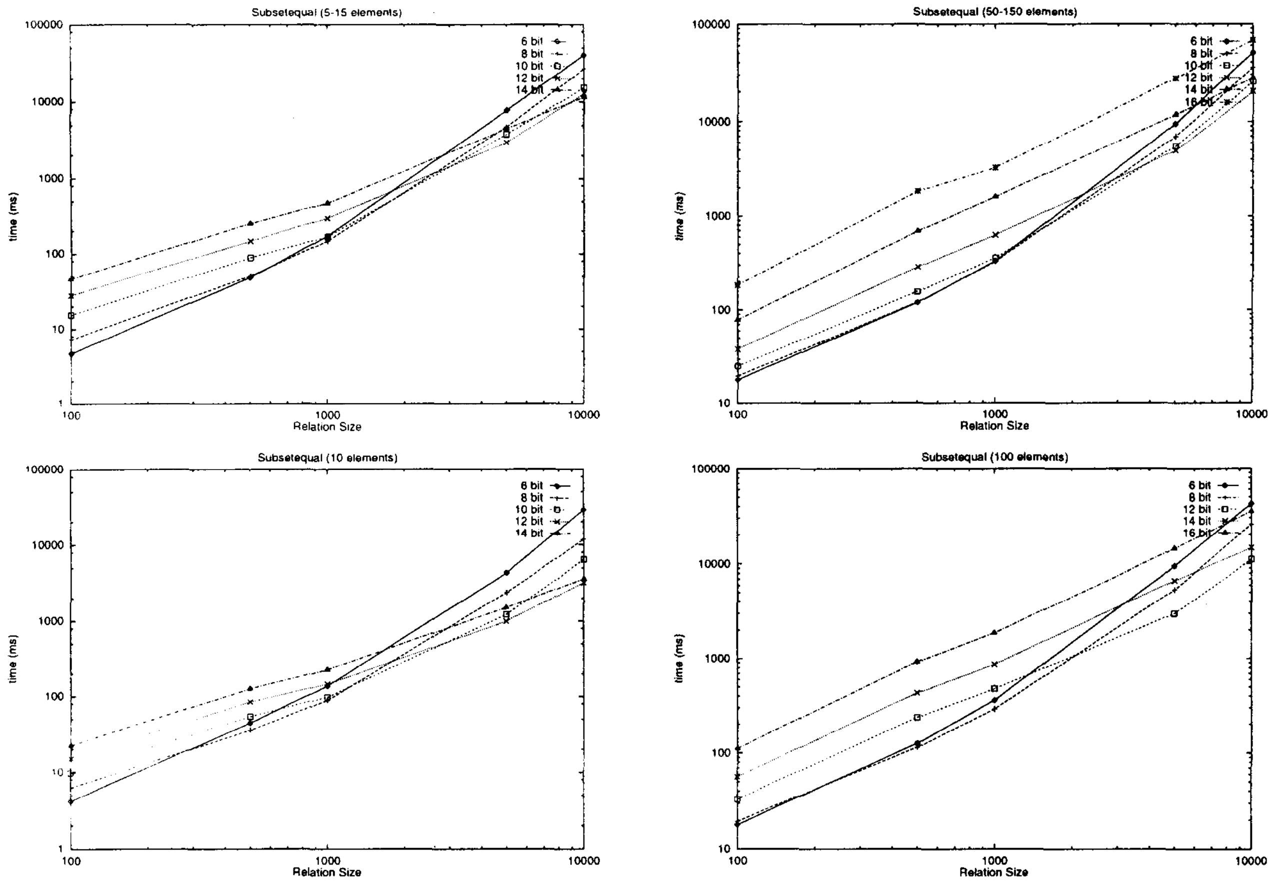


Figure 2: Performance of the hash join depending on the partial signature size

half of the bits to be set. So the mean number of hash lookups per set are

$$\overline{C}_{look} = 2^{\frac{d}{2}} \tag{5}$$

For the total number of lookups and accesses to items in the collision chain per tuple $t_2 \in R_2$ we get

$$\overline{C}_{totlook} = \overline{l}_{coll} \cdot \overline{C}_{look} = \frac{|R_1|}{2^{\frac{d}{2}}} \tag{6}$$

One could assume that the larger d becomes, the smaller the lookup costs per set become. This is, however, not the whole truth. Obviously, d cannot be indefinitely large, because of memory limits. Also d should not be larger than $\log_2 |R_1|$, because an average collision chain length below one does not improve the lookup cost significantly. A very large d does however increase the mean number of hash lookups, which is more crucial. See figure 2 that displays results of our experiments confirming these arguments. It follows that a hash table in the order of $|R_1|$ is reasonable and should also be feasible.

Let us now approximate the costs for the hash join (C_{insert} are the costs for inserting an item into a hash table, usually $C_{insert} = O(1)$ for collision chains).

$$\begin{aligned} C_H \approx\; & (|R_1| + |R_2|) \cdot C_{create}(\overline{r}) + \\ & |R_1| \cdot C_{insert} + \\ & |R_2| \cdot \frac{|R_1|}{2^{\frac{d}{2}}} \cdot C_{sig} + \\ & |R_2| \cdot |R_1| \cdot P_{hit} \cdot C_{set}(\overline{r}) \end{aligned} \tag{7}$$

For hash join we have an additional term for creating the hash table. The third term, determining the costs for the signature comparisons, has also changed (cf. (2)). Again, we omitted the costs for the construction of the result.

Implementation details

For the computation of signatures, for each element of the set one has to determine which bit to be set. One approach is to use the set element as a seed in a random number generator. We used two different random number generators (*rand* from the C-library and

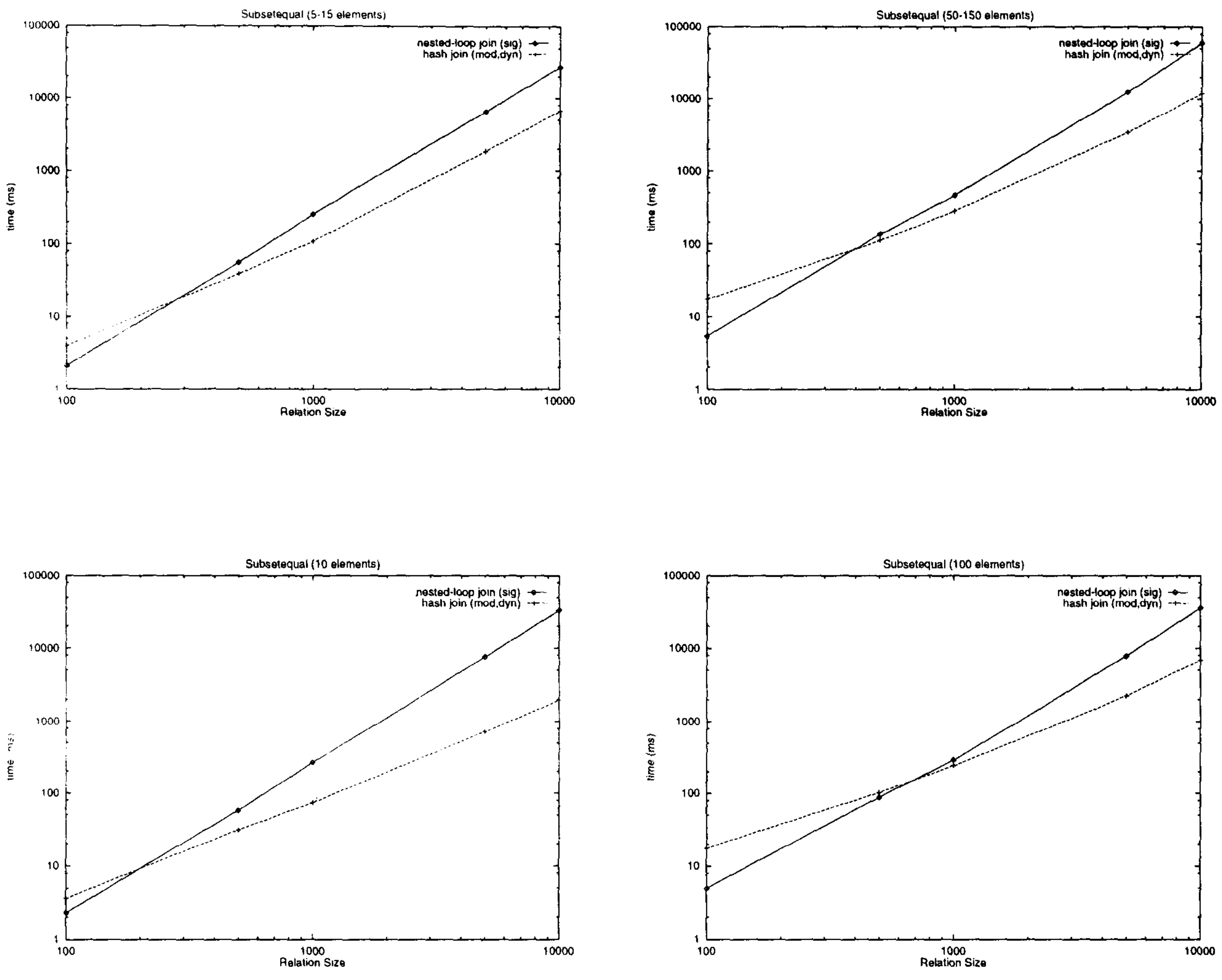


Figure 3: Performance of different join algorithms for $\subseteq$

DiscreteUniform from the GNU-library). We also investigated the simple approach where the set element is mapped to the bit to be set by the modulo function. We found that the random number generators resulted in a more uniform distribution of the bits to be set and, hence, in smaller collision chains. However, the decrease in collision chain length could not compensate for the high run time of the random number generator itself. As a consequence, for main-memory join algorithms random number generators do not pay. For a join algorithms working with disks, the situation might look different.

In our implementation, we allocated the signatures individually. Some tuning concerning this point is possible. For the set comparison we relied on the (whole) signature. That is, for each qualifying collision chain the given signature was compared to every signature of the elements within the collision chain. For every drop, the naive set comparison algorithm is applied. We also experimented with the sort-based comparison, but the faster comparison does not compensate for the sorting of the set elements. This is mainly due to the low false drop probabilities encountered.

3.2 Evaluation

3.2.1 Theoretical evaluation

Comparing hash join to nested-loop join we immediately see that hash join has additional costs for creating a hash table and different costs for comparing the signatures (see (2) and (7)). As seen before, it makes sense to choose the hash table size in the order of the cardinality of R_1. Hence, we set

$$d = \log_2 |R_1|$$

Then the costs for comparing signatures become equal to

$$|R_2| \cdot \sqrt{|R_1|} \cdot C_{sig}$$

for hash join as opposed to

$$|R_2| \cdot |R_1| \cdot C_{sig}$$

for nested-loop join. How will this influence the performance of the algorithms?

Small values of P_{hit} and small set cardinalities increase the impact of the signature comparison costs ($|R_2| \cdot |R_1| \cdot C_{sig}$ for nested-loop join and $|R_2| \cdot \sqrt{|R_1|} \cdot C_{sig}$ for hash join, respectively) on the total costs. Let us look closer at this case, in which the hash join outperforms the nested-loop join. The value of P_{hit} depends almost exclusively on the selectivity between R_1 and R_2, because the false drop probability can be held very low. For small selectivities, however, hash join is better than nested-loop join, as many unnecessary comparisons are avoided. Obviously, small set cardinalities result in a decrease of the signature creation costs $C_{create}(\overline{r})$ and set comparison costs $C_{set}(\overline{r})$. Hence, the proportion of the term $(|R_1| + |R_2|) \cdot C_{create}(\overline{r})$ and the term $|R_2| \cdot |R_1| \cdot P_{hit} \cdot C_{set}(\overline{r})$ in the total costs decreases. If we assume $|R_1| = |R_2| = n$, then the comparison of the signatures can be achieved in $O(n^{1.5})$ for the hash join in contrast to $O(n^2)$ for the nested-loop join. So the larger the relations are, the greater the performance gain of the hash join will be (under the assumption that P_{hit} and the set cardinalities are small).

Vice versa large selectivities and large set cardinalities increase the influence of the signature creation costs $((|R_1| + |R_2|) \cdot C_{create}(\overline{r}))$ and the set comparison costs $(|R_1| \cdot |R_2| \cdot P_{hit} \cdot C_{set}(\overline{r}))$ on the total costs. Therefore the performance gain of the hash join will not be as large, because the fraction of the total costs concerning the signature comparison costs $(|R_2| \cdot \sqrt{|R_1|} \cdot C_{sig})$ decreases.

3.2.2 Experimental evaluation

We also evaluated the nested-loop join and hash join experimentally. Due to time constraints we had to keep the benchmarks simple (see table 1 for the chosen parameters). Both the element distribution and the set cardinality distribution was uniform. The selectivity was implicitly defined by the element and set cardinality distribution. It was fairly small (lower than 0.0008 in all cases) and we did not investigate it further.

Parameter		*Values*
set card.	without var.	10, 100
	with var.	5-15, 50-150
relation card.		100, 500, 1000 5000, 10000

Table 1: Parameters for benchmarks

Figure 3 shows the performance evaluation of our join algorithms. More specifically, it contains the run time of the nested-loop algorithm with signature-based set comparison and the hash join with the tuned parameters set according to the above considerations for the signature length and the partial signature length. The hash join allocates dynamically a signature object for each tuple to be hashed. This signature object in turn allocates the signature bit vector dynamically. Only one signature object is allocated for all tuples of the outer relations. It is cleared before re-usage. This makes sense since clearing is less expensive than dynamic allocation. If the signatures are allocated statically, we can gain a factor of three.

As predicted in 3.2.1, the speedup for using hash join decreases for large set cardinality when compared to nested-loop join as the costs for creating signatures and comparing sets increase. For relations containing 10.000 tuples, the hash join saves a factor of up to 10 for small sets (containing 10 elements) and a factor of up to 5-6 for large sets (containing 100 elements). Also, because of the low selectivity used in the benchmarks, for large relation cardinalities, the hash join algorithm performs better than the nested-loop algorithm. For larger set cardinality the break-even point moves to the right as can be clearly seen in figure 3.

4 Conclusion

For the first time, this paper investigates join algorithms for join predicates based on set comparisons. More specifically, this paper treats subset predicates. It has been shown that remarkably more efficient algorithms exist than a naive nested-loop join. Even the signature nested-loop join results in an order of magnitude improvement over the naive nested-loop join. The hash join surpasses the signature nested-loop join only by a factor of 5-10 depending on various parameters. Although this is a result that is not to be neglected, the question arises whether even better alternatives exist. This is one issue for future research. Other problems need to be solved as well. First, join algorithms whose join predicate is based on non-empty intersection have to be developed. Second, all the algorithms presented are main memory algorithms. Hence, variants for secondary storage have to be developed. Also the different tuning parameters will have to be adjusted for secondary storage variants. For example, lowering collision chain entries and false drops by using random number generators and larger signatures might pay. Third. parallelizing these algorithms is an interesting issue by itself.

Acknowledgments

We thank B. Rossi and W. Scheufele for carefully reading a first draft of the paper. We also thank the anonymous referees for many useful comments. Especially one of them, whose hints improved the theoretical part of the paper quadratically (if not even exponentially).

References

[1] M. W. Blasgen and K. P. Eswaran. On the evaluation of queries in a relational database system. Technical Report IBM Research Report RJ1745, IBM, 1976.

[2] K. Bratbergsengen. Hashing methods and relational algebra operations. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 323–333, 1984.

[3] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 237–246, 1993.

[4] R. Cattell, editor. *The Object Database Standard: ODMG-93.* Morgan Kaufmann, 1996. Release 1.2.

[5] S. Cluet and G. Moerkotte. Efficient evaluation of aggregates on bulk types. In *Proc. Int. Workshop on Database Programming Languages*, 1995.

[6] D. DeWitt and R. Gerber. Multiprocessor hash-based join algorithms. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 151–164, Stockholm, Sweden, 1985.

[7] D. DeWitt, R. Katz, F. Ohlken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 1–8, 1984.

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

[9] D. DeWitt, J. Naughton, and D. Schneider. An evaluation of non-equijoin algorithms. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 443–452, Barcelona, Spain, 1991.

[10] D. DeWitt, J. Naughton, and D. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Int. Conf. on Parallel and Distributed Information Systems*, Miami Beach, Fl, 1991.

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

[12] S. Fushimi, M. Kitsuregawa, and H. Tanaka. An overview of the systems software of a parallel relational database machine: GRACE. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 209–219, 1986.

[13] G. Graefe. Sort-merge-join: An idea whose time has(h) passed? In *Proc. IEEE Conference on Data Engineering*, pages 406–417, Houston, TX, 1994.

[14] G. Graefe, A. Linville, and L. Shapiro. Sort versus hash revisited. *IEEE Trans. on Data and Knowledge Eng.*, 6(6):934–944, Dec. 1994.

[15] O. Günther. Efficient computation of spatial joins. In *Proc. IEEE Conference on Data Engineering*, pages 50–59, Vienna, Austria, Apr. 1993.

[16] T. Härder. Implementing a generalized access path structure for a relational database system. *ACM Trans. on Database Systems*, 3(3):285–298, 1978.

[17] S. Helmer and G. Moerkotte. Evaluation of main memory join algorithms for joins with set comparison join predicates. Technical Report 13/96, University of Mannheim, Mannheim, Germany, 1996.

[18] E. Hoel and H. Samet. Benchmarking spatial join operations with spatial output. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 606–618, Zurich, 1995.

[19] Y. Ishikawa, H. Kitagawa, and N. Ohbo. Evaluation of signature files as set access facilities in oodbs. In *Proc. of the 1993 ACM SIGMOD*, pages 247–256, Washington D.C., 1993.

[20] M. Kamath and K. Ramamritham. Bucket skip merge join: A scalable algorithm for join processing in very large databases using indexes. Technical Report 20, University of Massachusetts at Amherst, Amherst, MA, 1996.

[21] C. Kilger and G. Moerkotte. Indexing multiple sets. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 180–191, Santiago, Chile, Sept. 1994.

[22] W. Kim, K. C. Kim, and A. Dale. Indexing techniques for object-oriented databases. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 371–394, Massachusetts, 1989. Addison Wesley.

[23] M. Kitsuregawa, M. Nakayama, and M. Takagi. The effect of bucket size tuning in the dynamic hybrid GRACE hash join method. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 257–266, 1989.

[24] M.-L. Lo and C. Ravishankar. Spatial hash-joins. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 247–258, Montreal, Canada, Jun 1996.

[25] R. Lorie and H. Young. A low communication sort algorithm for a parallel database machine. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 135–144, 1989. also published as: IBM TR RJ 6669, Feb. 1989.

[26] J. Menon. A study of sort algorithms for multiprocessor DB machines. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 197–206, Kyoto, 1986.

[27] P. Mishra and M. H. Eich. Join processing in relational databases. *ACM Computing Surveys*, 24(1):63–113, 1992.

[28] M. Nakayama, M. Kitsuregawa, and M. Takagi. Hash-partitioned join method using dynamic destaging strategy. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 468–478, 1988.

[29] P. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Record*, 24(3):8–11, Sep 1995.

[30] J. Patel and D. DeWitt. Partition based spatial-merge join. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 259–270, Montreal, Canada, Jun 1996.

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

[32] M. Roth, H. Korth, and A. Silberschatz. Extending relational algebra and calculus for nested relational databases. *ACM Trans. on Database Systems*, 13(4):389–417, Dec 1988.

[33] R. Sacks-Davis and K. Ramamohanarao. A two level superimposed coding scheme for partial match retrieval. *Information Systems*, 8(4):273–280, 1983.

[34] B. Salzberg, A. Tsukerman, J. Gray, M. Stewart, S. Uren, and B. Vaughan. FastSort: an distributed single-input single-output external sort. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 94–101, 1990.

[35] H.-J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, 1986.

[36] D. Schneider and D. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 469–480, Brisbane, 1990.

[37] L. D. Shapiro. Join processing in database systems with large main memories. *ACM Trans. on Database Systems*, 11(3):239–264, 1986.

[38] E. J. Shekita and M. J. Carey. A performance evaluation of pointer-based joins. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 300–311, 1990.

[39] D. Shin and A. Meltzer. A new join algorithm. *SIGMOD Record*, 23(4):13–18, Dec. 1994.

[40] P. Valduriez. Join indices. *ACM Trans. on Database Systems*, 12(2):218–246, 1987.

[41] B. Vance and D. Maier. Rapid bushy join-order optimization with cartesian products. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 35–46, Toronto, Canada, 1996.

[42] Z. Xie and J. Han. Join index hierarchies for supporting efficient navigation in object-oriented databases. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 522–533, 1994.