Evaluation of Main Memory Join Algorithms for Joins with Set Comparison Join Predicates

Sven Helmer *Guido Moerkotte*

Lehrstuhl für praktische Informatik III
Fakultät für Mathematik und Informatik
Universität Mannheim
Seminargebäude A5
68131 Mannheim
Germany
helmer|*moer*@pi3.informatik.uni-mannheim.de

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 several main memory algorithms to evaluate efficiently this kind of join. More specifically, we concentrate on the set equality and the 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 [19, 22, 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, 21, 20, 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, 24, 17, 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 the join predicate is based on set comparisons like set equality (=) or subsetequal ($\subseteq$). 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.

The rest of the paper is organized as follows. In the next subsection, we introduce some basic notions needed in order to develop our join algorithms. Sections 3 and 4 introduce and evaluate several join algorithms where the join predicate is set equality and subsetequal. Section 5 concludes the paper.

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 don't 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 expressions

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

and

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

More specifically, we introduce join algorithms based on sorting and hashing and compare their performance with a simple nested-loop strategy. In addition to this, we describe a tree-based join algorithm and evaluate its performance.

For convenience, we assume that there exists a 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 of the type of the elements 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 = or $\subseteq$ differ significantly depend on the algorithm used. 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 $O(|s| * |t|)$. Set equality can then be implemented by testing $s \subseteq t$ and $t \subseteq s$, giving rise to a factor of two. For small sets, this might be a reasonable strategy. For large sets, however, the comparison cost with 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 abandoned 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, and denoting the i-th element of a set s by $s[i]$, the following algorithm implements set comparison $s = t$, if the sets are sorted:

```
if(s->setsize != t->setsize)
  return false;
for(int i=0; i < setsize; i++) {
  if (s[i] != t[i])
    return false;
}
return true;
```

Two comments should to be made. First, note that we introduced a pretest by testing the cardinality of the sets to be equal. This kind of pretest is used in every set comparison algorithm we implemented– also in the above mentioned trivial ones. Second, when applying this sort-based algorithm for set equality within our join algorithms, we do not assume that the elements of the set are sorted. Instead, the sort is performed by the join algorithms explicitly. This way, the comparison with other join algorithms is not biased by additional assumptions. Note that this algorithm runs in $O(|s|)$. Since we do not assume that the sets are sorted, we have to add $O(|s| \log |s| + |t| \log |t|)$ for sorting s and t.

A predicate of the form $s \subseteq t$ can also take advantage of sorting the sets. Again, 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->setsize > t->setsize)
  return false;
i=j=0;
while(i < s->setsize && j < t->setsize) {
  if(s[i] > t[j]) {
    j++;
  } else if (s[i] < t[j]) {
      return false;
  } else { /*  (s[i] == t[j]) */
      i++;
      j++;
  }
}
if(i==s->setsize)
  return true;
return false;
```

Note that the run time of this algorithm is $O(|s| + |t|)$. Again, since we do not assume that the sets are sorted, we have 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. Based on $m_{sig}(x)$ we can now give the algorithm to compute the signature $sig(s)$ for a set s.

```
sig = 0;
for(int i=0; i < s.setsize; i++)
   sig |= 1 << m_sig(s[i]);
return sig;
```

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

$$\begin{aligned} sig(s) = sig(t) &\quad\Longleftarrow\quad s = t \\ sig(s) \subseteq sig(t) &\quad\Longleftarrow\quad s \subseteq t \end{aligned}$$

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 ¬ 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 algorithm used to implement m_{sig} and its quality. 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 reduction of the false drop probability, the advantage of the latter is a much better run time. This trade-off will be investigated experimentally below.

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 signatures.

The false drop probabilities for subset and superset predicates have been studied [11, 18, 31, 33] and can be approximated by the following general formula [18]:

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

$$d_{f\supseteq}(b,k,r_s,r_t) \approx (1-e^{-\frac{k}{b}r_s})^{k\cdot r_t} \quad (2)$$

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

This leaves it to us to compute the false drop probability $d_{f=}(b,k,r_s,r_t)$ for set equality. This is the probability that for any two sets s and t, the signatures $sig(s)$ and $sig(t)$ are equal, despite the fact that $s \neq t$:

$$Pr(sig(s) = sig(t)|s \neq t)$$

Roberts has thoroughly examined the theoretical background of false drop probabilities [31]. We will use some of his results to derive a formula for $d_{f=}(b,k,r_t,r_s)$.

The probability $p_1(b,k,r_t,r_s)$ that z specific bits are set to ‘1’ (while ignoring the remaining $b-z$ bits) in a signature can be approximated by [31]:

$$p_1(b,k,r_t,z) \approx (1-(1-\frac{k}{b})^{r_t})^z \quad (3)$$

The probability $p_0(b,k,r_t,r_s)$ that the other $b-z$ specific bits are set to ‘0’ (while not looking at those z bits) in a signature can be estimated by the analogous formula:

$$p_0(b,k,r_t,z) \approx (1-\frac{k}{b})^{r_t\cdot(b-z)} \quad (4)$$

The probability $d_{f=}(b,k,r_t,r_s)$ that exactly z bits are set to '1' and the other $b-z$ bits are set to '0' in a signature can now be approximated by

$$d_{f=}(b,k,r_t,r_s) \approx p_0(b,k,r_t,w_q) \cdot p_1(b,k,r_t,w_q) \quad (5)$$

where w_q corresponds to the expected weight of r_s, i.e. the number of bits set in the signature of the query set. It can be computed as

$$w_q = b(1-(1-\frac{k}{b})^{r_s}) \quad (6)$$

This is an upper bound for the false drop probability, because

$$d_{f=}(b,k,r_t,r_s) = p_0(b,k,r_t,r_s) \cdot p_1(b,k,r_t,r_s) - p_0(b,k,r_t,r_s) \cap p_1(b,k,r_t,r_s).$$

3 Join Predicates with Set Equality

3.1 Algorithms

This section describes the algorithms we evaluated for implementing a join of the form $R_1 \bowtie_{a=b} R_2$ where R_1 and R_2 are relations with set-valued attributes a of R_1 and b of R_2. We discuss several variants of three major approaches. First, we briefly evaluate three variants of the nested-loop approach. Subsequently, the sort-merge join and the tree-join are described. Last, we introduce the hash join variants.

3.1.1 Nested-Loop Joins

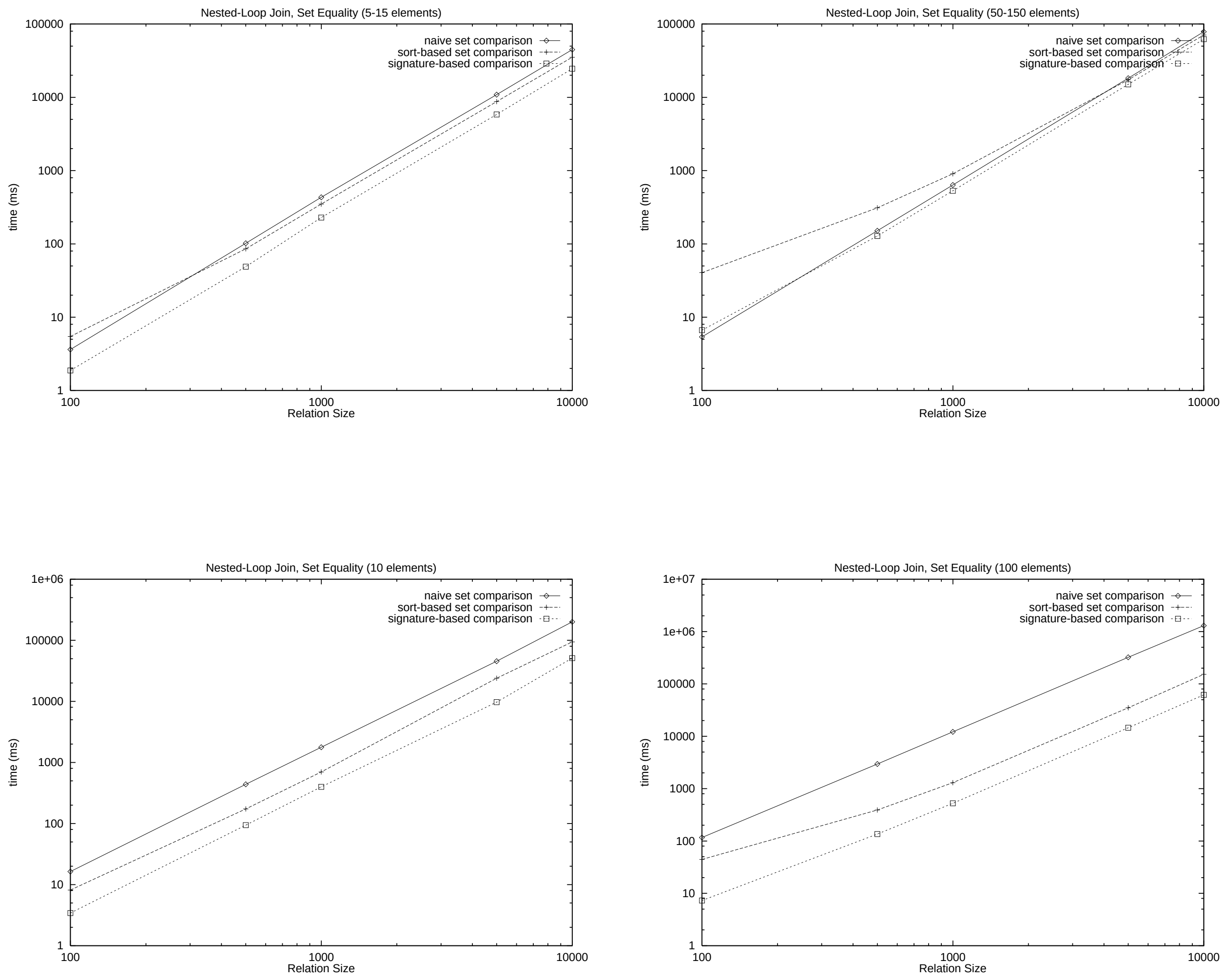
The nested-loop join is implemented easily by two nested loops ranging over the tuples of the inner and outer relation, respectively. We implemented the nested-loop join algorithm with the above described three different set comparison operations: the naive one, the one based on sorting the sets, and the comparison based on signatures.

In Figure 1, these three variants are evaluated. The top row shows the results for varying set sizes, the bottom row gives the results for fixed set sizes. Hence, the pretest on set sizes only accelerates the set comparison for the top row. In most cases–except for small relations and varying set sizes– the fastest comparison is that by signatures. Hence, we will use this a reference point for comparing subsequent algorithms.

3.1.2 Sort-Merge Join

The main idea of sort-merge join algorithms is to sort the relations to be joined on their join attributes and to merge subsequently the two sorted relations. If the join attributes have a simple, ordered domain, this approach is well known. However, ordering set-valued attributes seems less obvious on the first sight. The idea is to use a lexicographical ordering on sets.

More specifically, our sort-merge join sorts the relations in two steps. In a first step, the sets within the join attributes are sorted. In a second step, the relations themselves



Nested-Loop Join, Set Equality (5-15 elements)

Nested-Loop Join, Set Equality (50-150 elements)

Nested-Loop Join, Set Equality (10 elements)

Nested-Loop Join, Set Equality (100 elements)

Figure 1: Performance of different nested-loop join algorithms

In the top row, the number of elements per set varies uniformly between 5 and 15 for the left-hand side of the figure and between 50 and 150 for the right-hand side of the figure. In the bottom row, the number of elements is always 10 for the left-hand side figure and 100 for the right-hand side figure. Along the x-axes relation sizes are varied. The y-axes shows the cpu-time in milliseconds necessary for completing the join as measured on a Sun Sparc Station 20 with 64MB Main Memory. Note the logarithmic scale on both axes.

are sorted on their join attributes. We use a lexicographic ordering for sorting the tuples according to the set-valued join attributes. For example, the sets

$$\{1,4,7\}, \{1,5,7\}, \{2,5\}$$

are lexicographically ordered, whereas

$$\{1,5,7\}, \{1,4,7\}, \{2,5\}$$

are not.

Now, the merge phase proceeds as follows. For the outer relation–say R_1–exist two pointers (*low* and *high*) to tuples within it. These pointers indicate the lowest and the highest tuples within the sorted relations that are equal. A third pointer (j) is used to range over the inner relation–say R_2. The join attributes of the first two tuples t_1 and t_2 of the relations to be joined are compared. If the set a of t_1 precedes lexicographically the set b of t_2, then we can advance the *low* pointer of R_1. If the set b of t_2 precedes lexicographically the set a of t_1, then we advance the pointer (j) of R_2. If they are equal, a result tuple can be built. Further, we have to check subsequent tuples in R_1 whether they have the same a value. We do so by advancing the second pointer (*high*) and checking separately each tuple it points to until we meet a tuple whose a value is unequal to the according $t_2.b$ value. If, after such a run of R_1 tuples with equal attributes the next tuple in R_2 does not match, all these tuples can be skipped.

This idea is easily implemented in the following algorithm:

```
low=high=j=0;
flag = false;
while(low < R1.size && j < R2.size ) {
  while(low < R1.size && j < R2.size &&
        R1[low].a != R2[j].b ) {
    if(flag) {
      flag = false;
      low = high;
      continue;
    }
    if(R1[low].a < R2[j].b)
      low++;
    else if (R1[low].a > R2[j].b)
      j++;
  }
  if(low >= R1.size || j >= R2.size)
    break;
  /* build result tuple */
  high = low + 1;
  while(high < R1.size && R1[high].a = R2[j].b) ) {
    /* build result tuple */
    high++;
  }
  j++;
  flag = true;
}
```

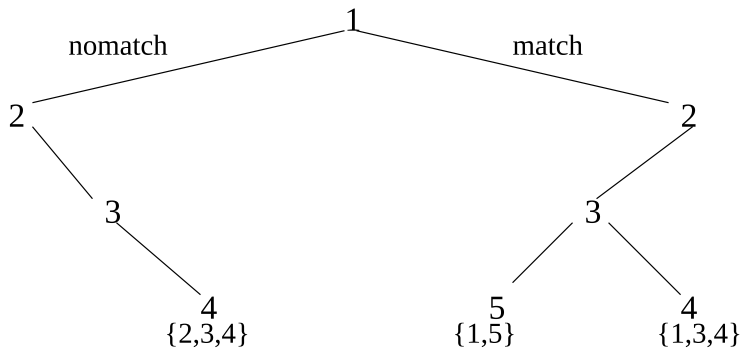
Since sorting had to be done before the application of this algorithm, the used set comparison algorithm is the one based on sorted sets. The performance evaluation of this algorithms can be found in Section 3.2

3.1.3 Tree Join

The basic idea behind a hash join is to use a temporary index. Instead of a hash table, other index structures could also be used. In this section, we consider a binary tree as the temporary index structure for a tree-based join algorithm. Let us first introduce the data structure by means of an example. Consider a relation containing the following three tuples with a set-valued attribute A:

1. $T1.A = \{1, 3, 4\}$
2. $T2.A = \{1, 5\}$
3. $T3.A = \{2, 3, 4\}$

The according tree is



Every node in the tree consists of three pointers and a key. The key contains a set element found in some of the relation's tuples. For every tuple, if the element occurs in the set and no further elements are found, a pointer points to a list containing this tuple. If it contains further elements, a *match* pointer (to the right) points to a subtree containing the tuple. If the key element does not occur in the tuple considered, the *nomatch* pointer (to the left) points to a subtree containing the tuple.

Before inserting the tuples into the tree, the sets in their set-valued attributes are sorted. We do not sort the relations as we did for the sort-merge approach. Just the elements within the set-valued attributes are sorted. Assume that t denotes the tuple to be inserted. Then, the insertion proceeds as follows:

```
void insert(Tuple* t) {
  TreeNode* place = root;
  TreeNode* oneup = 0;

  for(int i = 0; i < t->setsize; i++) {
    int elem = t->A[i];

    while(place->key < elem) { // go down to find the place of insertion
      if(place->nomatch == 0) {  // can't go down further, insert node
        place->nomatch = new TreeNode(elem);
      }
      oneup = place;
      place = place->nomatch;
    }

    if( place->key == elem ) {  // this is the place to go
      if( i == (t->setsize-1) ) {
         place->entries = new TreeCollEntry(t,place->entries);
         break;
      }
      if( place->match == 0 ) {
        place->match = new TreeNode(t->A[i+1]);
      }
      oneup = place;
      place = place->match;
    } else { // the new tuple must be inserted befor this node
      // (place->key > elem)
      if(place == oneup->nomatch) {
        TreeNode* newNode = new TreeNode(elem,0,place);
        oneup->nomatch = newNode;
        place = newNode;
        i--;                 // to create the match entry
      } else {
        // (place == oneup->match)
        TreeNode* newNode = new TreeNode(elem,0,place);
        oneup->match = newNode;
        place = newNode;
        i--;                // to create the match entry
      }
    }
  }
}
```

The algorithm proceeds as follows: it goes down the tree until it finds a place where the key is greater or equal to the next element of the set-valued attribute to be considered. Then, two cases have to be distinguished. First, the key is equal to the element to be inserted. If so–and if we are at the end of the set–we just insert the tuple into the entries list. Otherwise we have to go further down the match pointer in order to insert the next element of the set. If the set element is greater than the key, then this means that the element had to be inserted before the current node. This is done by creating an according node and inserting it between the predecessor of the current node and the current node. From there on, the search for an insertion place restarts.

To find for a given tuple all the joining tuples contained within the tree is now an easy task. We look successively at each element within the sorted set, starting with the smallest element and entering the tree at the root. If the current element is equal to the key of the current node, we go down the *match* branch. If it is not equal to the key of the current node, we proceed in the *nomatch* branch. We proceed the same way with the next element in the set until all elements have been processed. The tuples within the current *entries* list are the joining tuples.

3.1.4 Hash-Loop Join

The idea of the hash join algorithm is to build a hash table for the inner relation hashed on its join attribute. For set-valued attribute values, there exist several alternatives for the hash function. The simplest is to hash the sets by their cardinality. This yields a good distribution only if the join attribute values consist of sets of varying cardinality. In this case, a partitioning by set sizes might be useful. However, since in general this is not applicable, we considered two other alternatives:

1. direct hashing
2. signature hashing

In the first alternative, each set is directly mapped to some hash key. Since our assumption is that the set elements have already been mapped to the integer domain, we add up all elements in the set. The result is then taken modulo the hash table size to yield the hash value of the set. Within subsequent figures, this alternative is labeled by *sum*. The set comparison used in this algorithm is the naive one. As a variant of *sum*, we implemented one where the set comparison is based on sorting the sets. However, since the collision chain lengths are always quite short, sorting does not pay for this approach (see below).

The second alternative is based on signatures. Several aspects have to be considered when using signatures for hashing:

1. choice of the right signature length,
2. allocation of the memory for the signature,
3. computation of signatures, and
4. mapping signatures to hash values.

Signature Length The performance of signatures depends on their size. For a fixed set size, the larger the signature size, the smaller the false-drop probability. Hence, a large signature seems to be the best. However, there are two aspects which complicate the issue. First, the signature must be mapped onto a hash value. The larger the signature, the more "complex" the mapping becomes. The effect is that collisions of hash values become more likely. Second, computing the signatures just for computing the hash values seems a waste. If we store them together with the tuples in the hash table, then we are able to perform a signature comparison as a pretest on the collision chains within the hash tables. This results in a performance enhancement. If the signatures become too large, the storage overhead is no further neglectable. Hence, we have a trade-off between lowering the false-drop probability on one hand and minimizing storage overhead on the other hand. Considering the formula for false-drop probabilities in case of set equality as a comparison predicate, we can use a signature of 32 bits to have a false-drop probability below 1.4e-06 for up to 60 elements per set. Beyond 60 elements, the false drop probability increases dramatically. Hence, we use 64 bits for sets containing up to 150 elements. The resulting false-drop probability is less than 2.1e-09. These probabilities only hold if the variance of the set size is zero.

Allocating Memory for the Signature There exist two extreme cases for allocating memory for signatures. At one end, each tuple provides a couple of bytes (4 or 8) that are able to contain the signature. At the other end, a signature object can be created dynamically and attached to the tuple. Further, the signature bits themselves are allocated by the signature object. This results in two dynamic memory allocations. An alternative in the middle would be to allocate the signature object together with the signature bits at once. We implemented the two extreme situations in order to derive an upper bound on the performance loss of dynamic signature creation.

Computing Signatures There exist several possibilities to compute the signature. We investigated the alternatives described in section 2.3. We use the following scheme to denote the different alternatives:

ran for computing the signatures using the C-library *rand()* function

dis for computing the signatures using the GNU-library *DiscreteUniform* function

mod for computing the signatures by simply taking the elements modulo the signature length.

Mapping Signatures to Hash Values If the signature is within the 32 bit bound of an integer, it is mapped to a hash value by taking it modulo the hash table size. If it is longer, we consider two alternatives. In the first alternative, we truncate the signature into 32 bit pieces and fold these pieces by using the bit-wise exclusive or. The second alternative just considers the lower 32 bits of the signature and ignores the rest. This part of the signature is then taken modulo the hash table size in order to give the hash value. This approach will lead to the notion of *partial signature (size)* that is investigated thoroughly in the next section (see Sec. 4.3).

To show that the signature length plays a central role for the collision chain length, consider the following numbers. For each of the alternatives, the maximum collision chain length for some typical experiments are given. In each experiment, 500 tuples were inserted into the hash tables. The set sizes were varied.

Set Size	sum	32bit sig			64bit sig, fold			64bit sig, 32bit truncate		
		dis	ran	mod	dis	ran	mod	dis	ran	mod
10	6	5	6	6	7	7	8	9	7	7
9-11	4	6	6	7	7	7	6	8	8	8
44	5	7	8	7	5	5	6	5	5	5
40-48	6	7	7	7	5	5	5	7	4	5
100	4	119	128	128	7	6	6	8	6	9
90-110	5	109	126	126	7	5	6	8	6	8

Although these are only example experiments and some deviation for other experiments can be expected, a couple of observations can be made. First, if the signature is too small, like 32 bits for set sizes of about 100, the collision chain becomes very long. The reason is that there is a high probability for every bit in the small signature to be set. Second, the quality of the random number generator plays a minor role for the collision chain length. On the average, for the examples shown, using a random number generator leads to slightly shorter collision chains than using a modulo value to derive the bit to be set. Nevertheless, since this operation is much faster than a random number generator call, algorithms based on the modulo operation outperform (in terms of run time) those based on a random number generator (see below). Further, this it is not true in general, that random number generators lead to shorter collision chains than a simple *mod* operation. Third, folding performs slightly better than taking only a portion of the signature and ignoring the rest. Fourth, the *sum* alternative seems to perform quite well. This is not true in general. If the relations become large, the probability of a collision becomes quite high, resulting in long collision chains.

Implemented Alternatives The following is a list of the implemented alternatives. The first algorithm uses the idea of summing up all elements, all the others are signature-based. Since the signature length plays a crucial role in performance, we allow for arbitrary signature length. However, the dynamic allocation of small signatures imposes some overhead. Hence, we have two further implementations for fixed signature lengths of 32 bit and 64 bit. The signature can be built using either a random number generator or a simple modulo operation. We tested two random number generators (*rand*, *DiscreteUniform*). A signature of more than 32 bit is not easily mapped to a table entry number by a simple modulo function. Hence, we use folding or truncation to map a signature to an 32 bit unsigned int. Then, from this number we can derive the table entry number by a taking it modulo the hash table size. These dimensions give rise to numerous alternatives. The implemented alternatives are summarized in the following list:

sum sum all elements in set

sumS sum all elements in set, sort sets for fast comparison

ran [32] 32 bit signature using *rand*

dis [32] 32 bit signature using *DiscreteUniform*

mod [32] 32 bit signature using *mod*

ran [64x] 64 bit signature using *rand*, folding (bit exclusive or) to 32 bit

dis [64x] 64 bit signature using *DiscreteUniform*, folding (bit exclusive or) to 32 bit

mod [64x] 64 bit signature using *mod*, folding (bit exclusive or) to 32 bit

ran [64t] 64 bit signature using *rand*, use only lowest 32 bits for hashing (truncate)

dis [64t] 64 bit signature using *DiscreteUniform*, use only lowest 32 bits for hashing (truncate)

mod [64t] 64 bit signature using *mod*, use only lowest 32 bits for hashing (truncate)

ZF [nn,32] arbitrary signature lengh, *mod* for building signature, folding (bit exclusive or) to 32 bit, *nn* denotes the actual signature length

ZT [nn,32] arbitrary signature length, *mod* for building signature, use only lowest 32 bits for hashing (truncate), *nn* denotes the actual signature length

Evaluation of Alternatives We ran several experiments in order to evaluate the performance of the different alternatives. Thereby we had several goals in mind:

1. verify that sorting for fast set comparison does not pay;
2. take a look at the collision chain length of the variants;
3. select the right approach to generate the signatures;
4. quantify the difference between folding and truncation;
5. quantify the overhead for dynamic signature allocation, and
6. select the best algorithm(s) to compare them with non hash-based algorithms, i.e., reduce the number of alternatives.

During all the experiments, the hash table size was chosen to be equal to the number of tuples. This is not necessary, since also much smaller hash table sizes result in similar performance, but for us it had the advantage that the average collision chain length is always equal to 1. Hence, the experiments on the hash tables are normalized and we only need to discuss the maximal collision chain length.

Figure 2 suggests to use no sorting of the sets for fast set comparisons, at least as long as the collision chain lengths remain small. As we will see (Fig. 3), this is not true for the *sum* alternative. This is the reason why the alternative where set comparison is based on sorting comes close to the alternative without sorting. For the other hash join variants, sorting clearly does not pay.

Figure 3 shows the collision chain length of the different hash tables. The maximum collision chain length is 20 for all but the *sum* alternative, even if the relations become very large. Hence, the signature-based variants proof to be superior. The figure on the right-hand side shows that folding is superior to truncation. For us this is bad news, since this will be the variant that has to be used for the $\subseteq$ join predicate. Fortunately, the signature sizes used for this figure are untuned. Using a better signature size leads to in much shorter collision chain lengths (see Fig. 10).

Figure 4 shows the performance of the 32 bit and 64 bit signature algorithms for the different alternatives to produce the signature. As expected, although their collision chains are typically shorter, the alternatives using random number generators need much more time than the *mod* alternative. Hence, this overhead does not pay in a main memory environment.

Figure 5 concentrates on 64 bit signatures where either folding or truncation is used. Given the shorter collision chains, folding is superior to truncation. This is due to the much shorter collision chains. However, as mentioned above, the truncation can be tuned.

Figure 6 quantifies the overhead of a dynamic versus a static implementation of signatures. As expected, there is an overhead for dynamic signature creation. The only question that remains is how big it is. The answer can now be given easily. For small sets–and, hence, a small signature size–the overhead is about a factor of 2-3. For larger sets, the overhead becomes neglectable ($< 10\%$) since the computation of the signatures consumes much more cpu time than their creation.

Figure 7 shows the performance of the most interesting alternatives. For small sets, the *mod* variant with static memory allocation performs best. For large sets, the *sum*

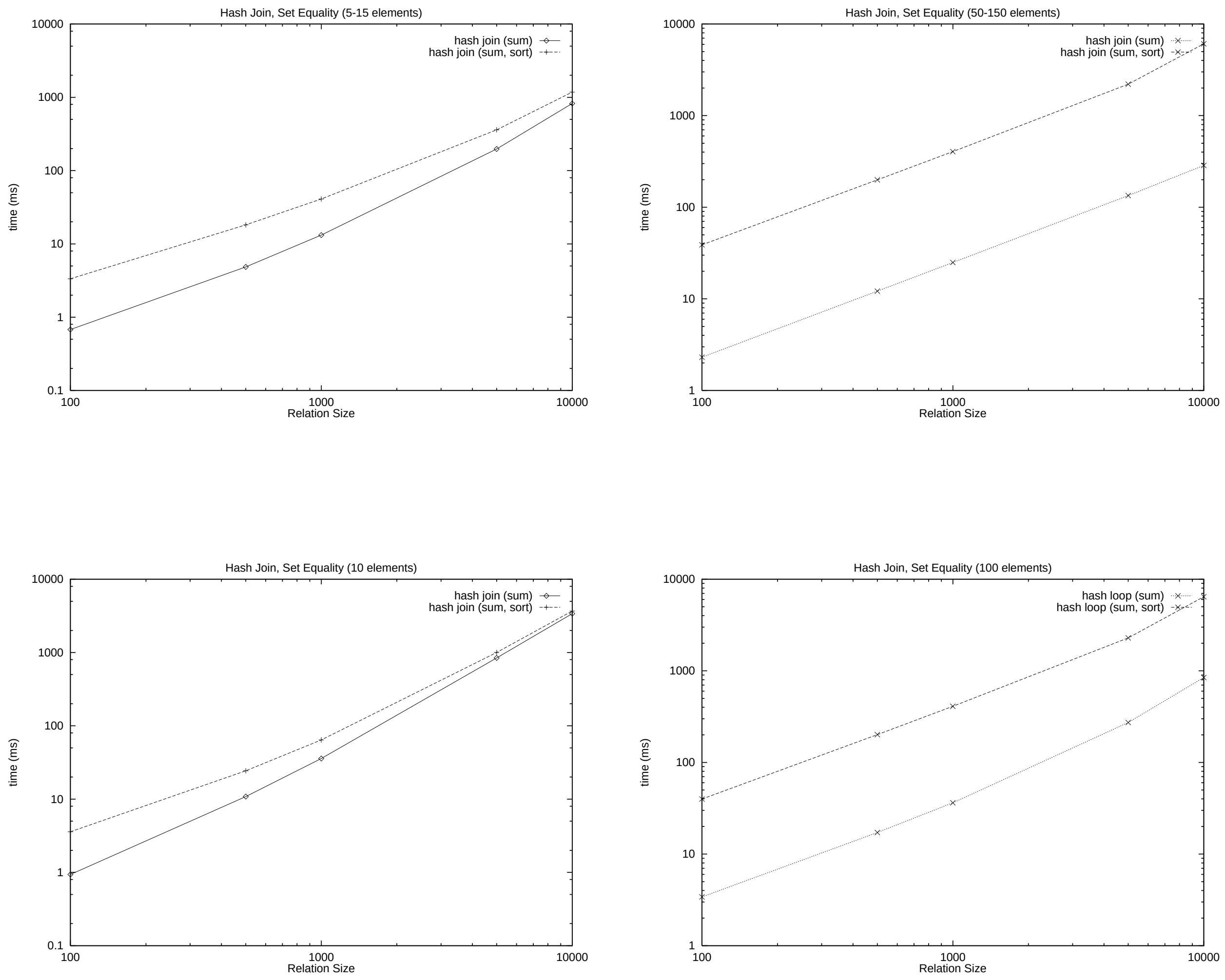


Figure 2: Sort-based versus naive set comparison

variant might be an alternative. However, we do not recommend its usage due to its possible degradation. Note that we also included the curves for those alternatives that build on dynamic signature allocation. Even if we have to join two relations with 10,000 tuples each and within every single tuple the set-valued attribute contains 100 elements, we can perform the join within one or two seconds, depending on the memory allocation strategy. A comparison with other approaches like nested-loop and sorting is discussed in the next section.

3.2 Evaluation

Figure 8 shows the performance figures of the best nested-loop variant using set comparison based on signatures, the sort-merge join, the tree join, and the *sum* and *mod* variants of the hash-join. For the latter, we used the variant with static signature allocation. This is quite reasonable, since the query optimizer already determines the query evaluation

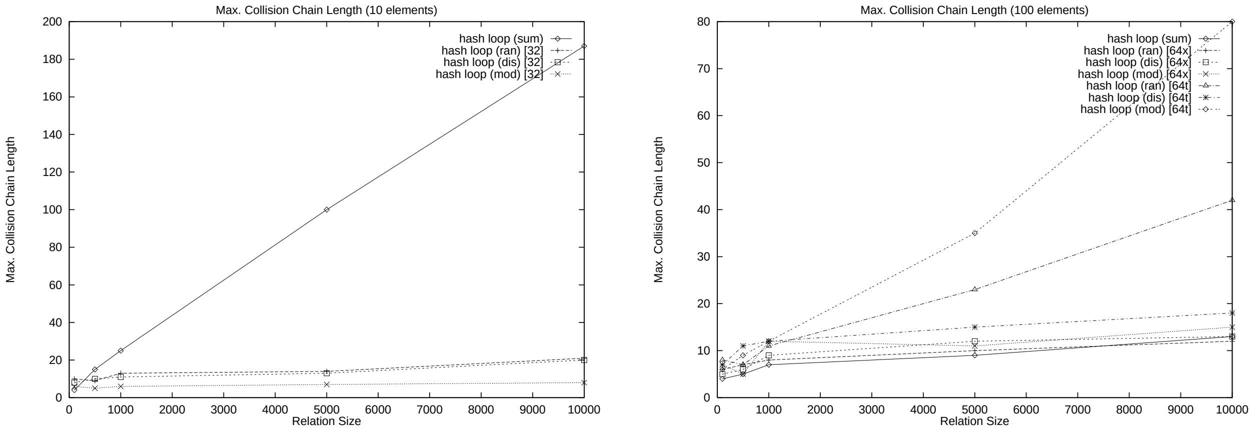


Figure 3: Maximal collision chain lengths of different hashing schemes

plan and hence fixes the join algorithm. Further, given that a signature hash join is to be performed, some space for the signature can easily be allocated in the tuples representing the intermediate relation. This then saves the dynamic memory allocation for the signatures. Remember also, that the overhead for dynamic signature allocation was quite small for larger sets.

The conclusion that can be drawn from Figure 8 and previous figures is that the *mod* variant is the method of choice. It performs best (or close to best) in all cases. Only for large sets, the *sum* alternatives prove to be superior. However, as mentioned above, the chance of degradation still remains. The nested-loop variant seems to be an alternative for small relations only. This is not surprising due to its complexity. If for some reason a hash-based alternative is not applicable within a given application, the sort-merge join and the tree join remain reasonable alternatives. From a run time perspective, the tree join performs slightly better for large relations and small sets. However, one should not neglect its tremendous storage overhead. Hence, the sort-merge join remains the alternative of choice if hash-based algorithms are not applicable.

For us, the most important conclusion was that there exist algorithms which are much more efficient than the nested-loop variant. Further, the efficiency of the alternative algorithms is surprisingly good. Joining two relations with 10,000 tuples each based on the equality of their set-valued attributes consisting of 100 elements within a second or two seems to be a reasonable result, especially when we consider that the best nested-loop variant needs almost one and a half minute for the same task.

4 Join Predicates with $\subseteq$

This section discusses algorithms to compute

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

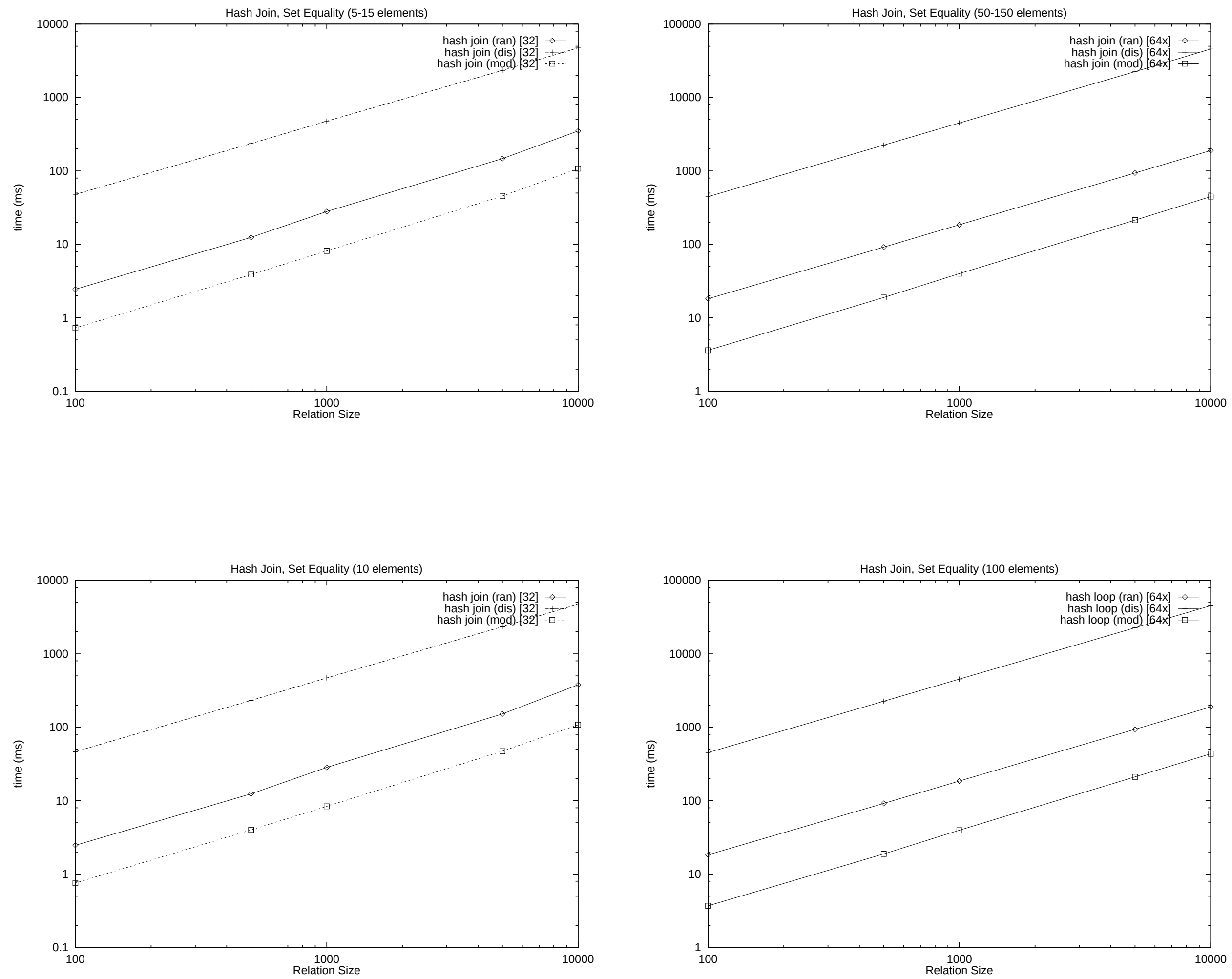


Figure 4: Alternative approaches to generate the signature

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 algorithms. It starts with the nested-loop algorithm. Then, the sort-merge join and the tree join follow. Finally, the hash join is discussed. The last subsection contains the evaluation of the different join algorithm for $\subseteq$.

4.1 Algorithms

4.1.1 Nested-Loop Joins

There are again three different possible implementations of the nested-loop join. Each alternative is based on a different implementation of the $\subseteq$ predicate. The first alternative

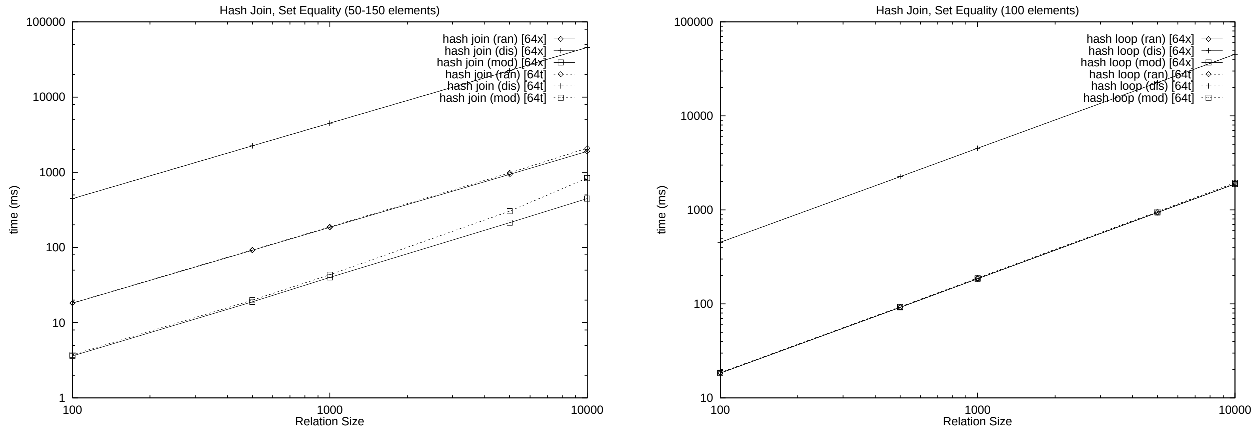


Figure 5: Folding and truncation

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.

Figure 9 shows the performance figures for the different nested-loop variants. Again, 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 with other $\subseteq$-join algorithms.

4.1.2 Sort-Merge Join

The situation here is very similar to the one encountered when performing a merge join for a $\leq$ predicate. There is no way for a simple merge of the two sorted relations suffices. The only thing we can do is break early the inner loop ranging over the second relation R_2. Assume that we want to find all those tuples in relation R_2 with an attribute value less than a. Then, as soon as we found the first tuple whose attribute value is greater than a, we can break the loop on R_2, since all subsequent attribute values will be even higher. For this, it does not make sense to sort the first relation R_1.

Similar situations occur when evaluating a join with the $\subseteq$ predicate. Let us assume that we want to find all matching tuples for a set s. Further assume that R_2 is sorted lexicographically. As soon as the smallest element in s is smaller than the smallest element in the attribute of a tuple in R_2, we can stop the loop, since this element will not be contained in any subsequent sets of tuples of R_2. A similar argument applies to the maximum element of s. All the tuples in R_2, whose maximum is smaller than that of s can be skipped. This idea is captured in the subsequent code fragment.

```
// R1->sort();  sorting the first relation doesn't make much sense
for(int i=0; i < relsize; i++) {  // instead we just sort the sets
  R1[i]->sort();
}
```

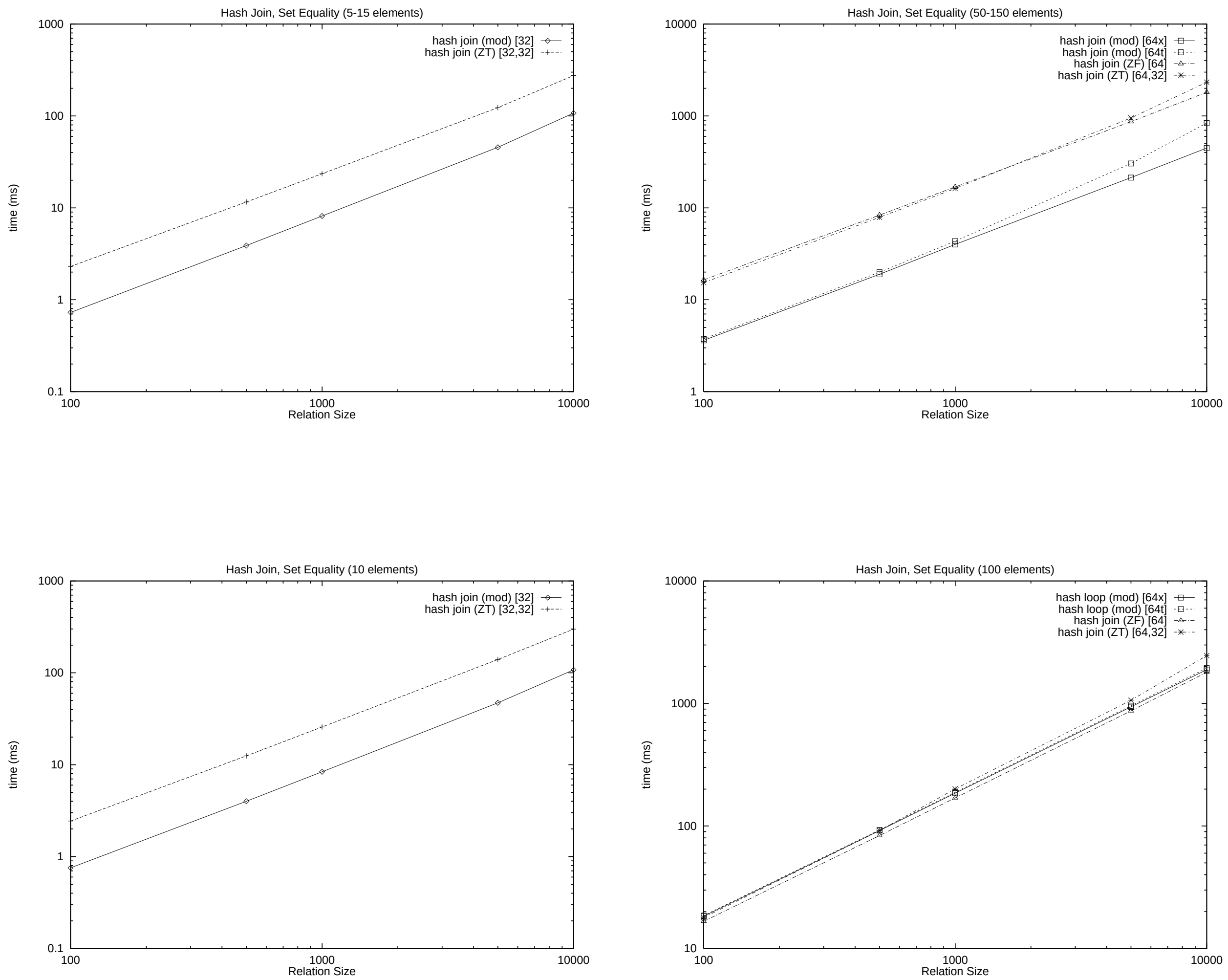


Figure 6: Overhead of dynamic signature creation

```
R2->sort(); // the second relation is sorted, includes sorting the sets
for(int t1=0; t1 < R1->relsize; t1++) {
for(int t2=0; t2 < R2->relsize; t2++) {
  if( R1[t1]->A[0] < R2[t2]->A[0] )
    break;                        // break if minimum in t2 is too big
  if( R1[t1]->A[ R1[t1]->setsize - 1 ] > R2[t2]->A[ R2[t2]->setsize - 1] )
    continue;                    // skip if maximum in T2 is too small
  if( R1[t1]->subseteqsorted(r->R[t2]) ) {
    // build result tuple
  }
}}
return res;
```

Unlike for the set equality join, there is no gain in the complexity of the run time of this

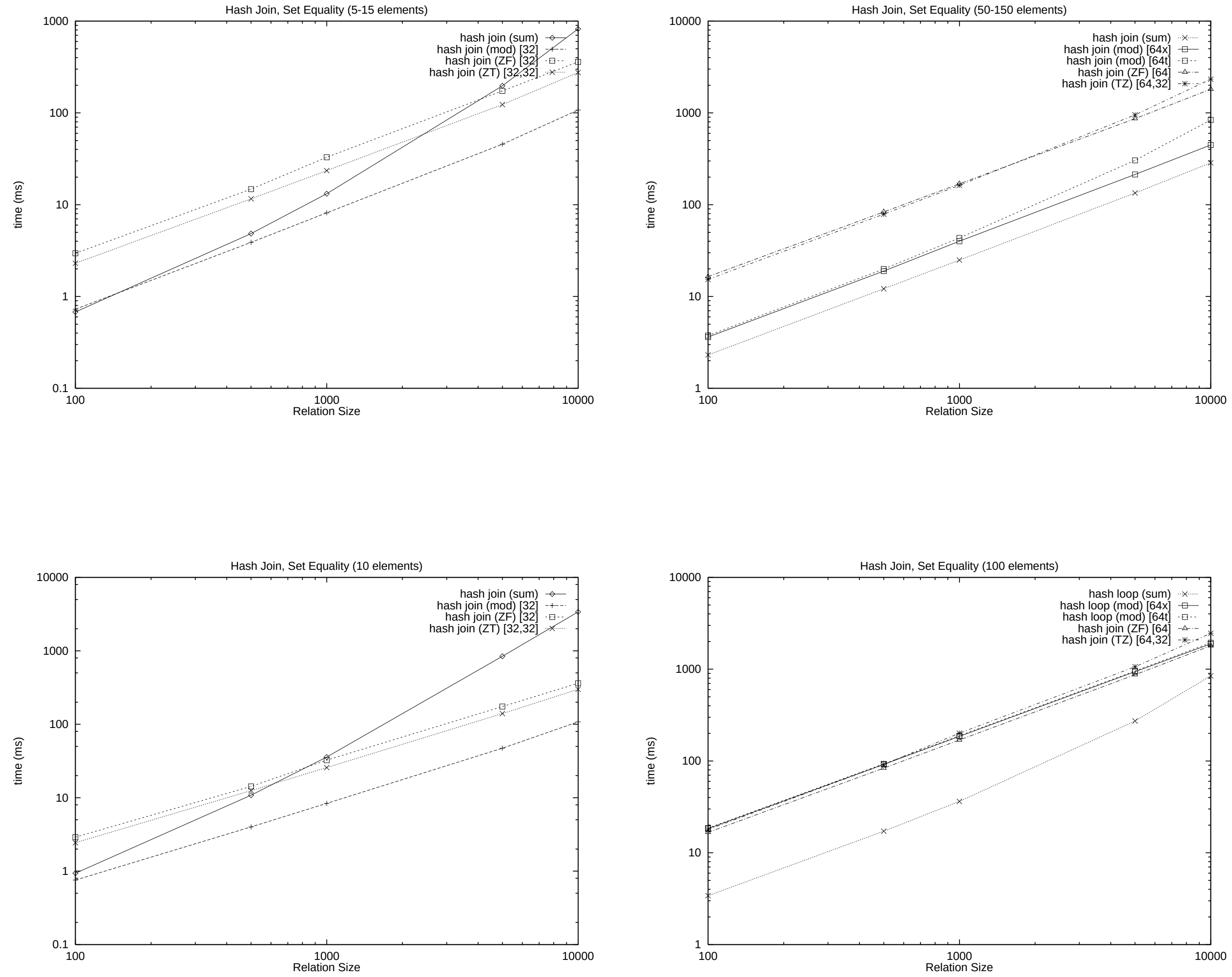


Figure 7: Comparison of the best alternatives

algorithm. It is still $O(n^2)$. Nevertheless, there is one new simple pretest (the continue case) which eliminates some evaluations of the expensive *subseteqsorted* predicate, which in turn is cheaper than the naive evaluation of $\subseteq$. Further, the *break* case implies that the inner loop has to be evaluated on only half of R_2 on the average. Hence, this alternative looks worthwhile to evaluate.

4.2 Tree Join

The tree join for set equality can easily be adapted to the subset predicate. We only have to change the retrieval algorithm. In order to find all the subsets of a given set within the tree, we advance simultaneously through the set and the tree. The *retrieve* procedure takes a tuple t, an index i indicating how far we proceeded already through its set-valued attribute and a tree node. If we are at the end of the tree, nothing has to be done. Otherwise, if we processed all the elements of a set, we can build the result tuples from

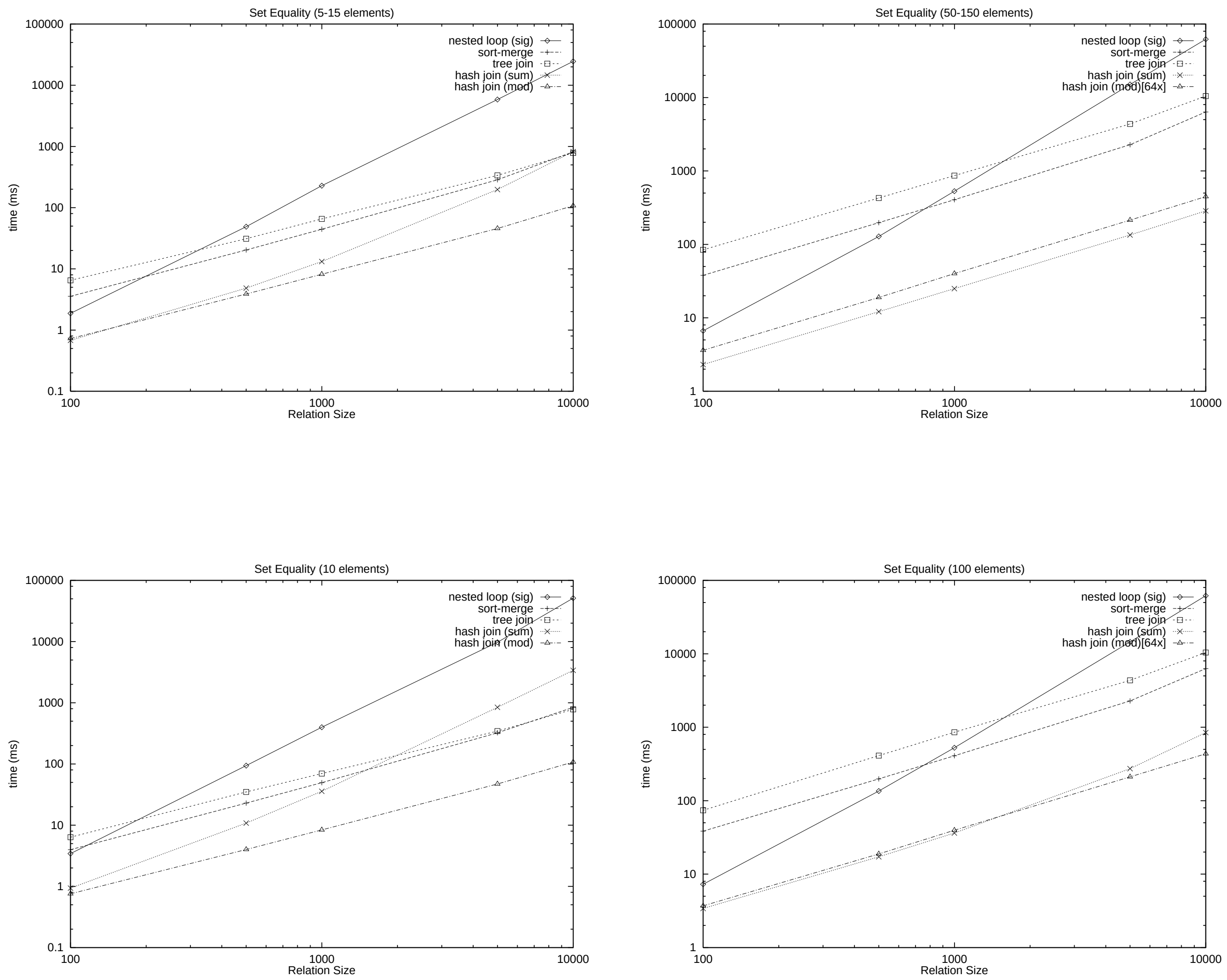


Figure 8: Performance of different join algorithms

the entries in the current node and all the nodes reachable from this node. If we are not yet done–that is, did not consider all elements of the set–we take a look at the key of the current node. If this key matches the current set element, we can advance our pointer i into the set and go into the *match*-branch of the current node. If the key is less than the current set element, we continue by investigating the *match* and *nomatch* branch. If the key is greater than the current set element, we certainly will not find any qualifying tuple below the current node and can quit the *retrieve* procedure. The code for this procedure is

```
retrieve(Tuple* t, int i, TreeNode* place) {
   if(place == 0)
     return;
   if(i == t->setsize) {
     // build result tuples from current node and subnodes
```

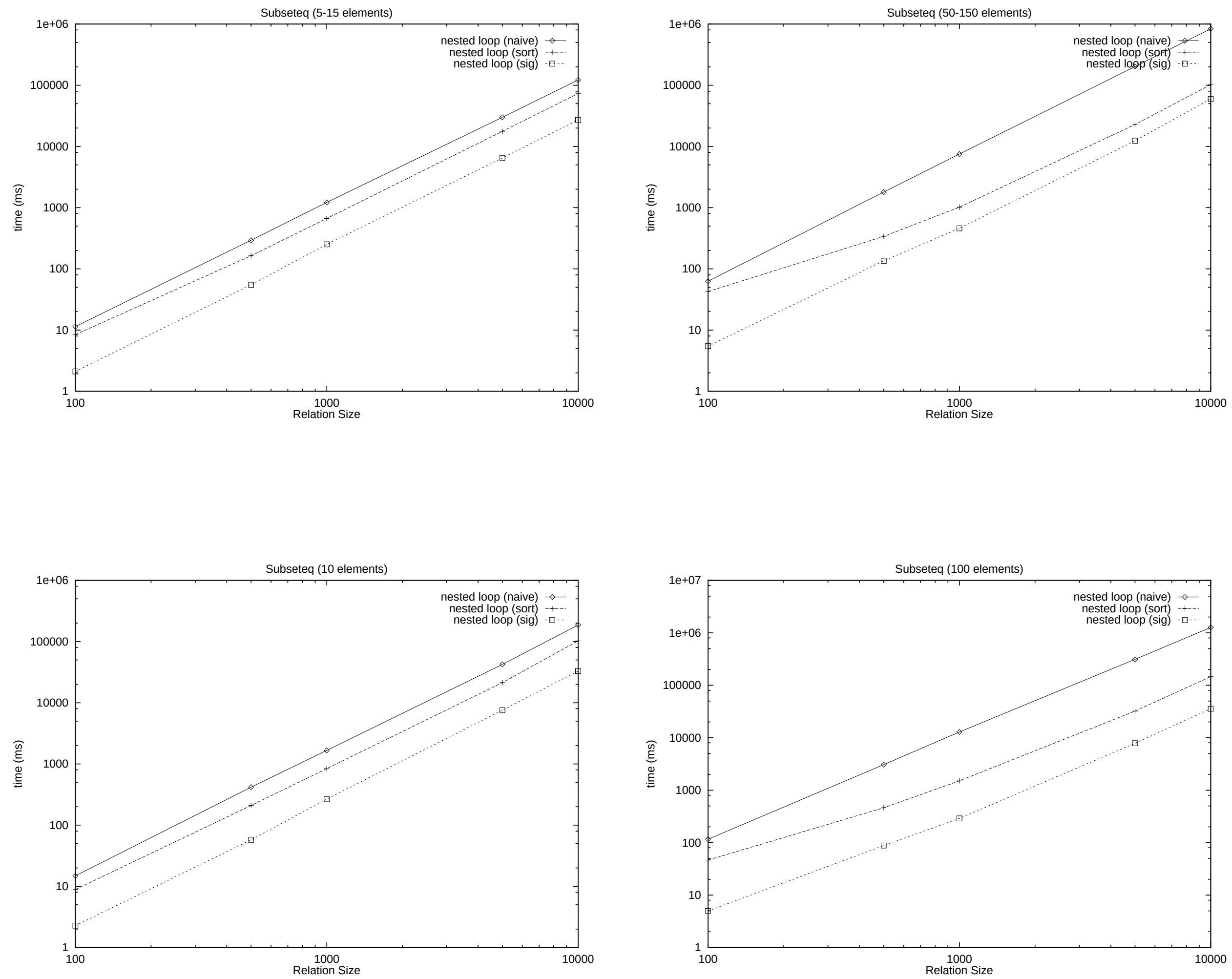


Figure 9: Performance of different nested-loop join algorithms

```
}
if( place->key < t->A[i] ) {
  if(place->match != 0)
     retrieve(t, i, place->match);
  if(place->nomatch != 0)
     retrieve(t, i, place->nomatch);
  return;
} else if( place->key == t->A[i] ) {
  if( i == (t->setsize - 1) ) {
   // build result tuples from current node and all successor nodes
   // in the match branch.
  } else {
    return retrieve(t, i+1, place->match);
```

```
      }
    } else {
      return;
    }
  }
```

It is important to note that the *retrieve* procedure investigates more than one branch within the tree. The complete join algorithm proceeds as follows: first, the tree is built for the inner relation. Then, for each tuple in the outer relation, the *retrieve* procedure is used to construct the joining tuples. For both relations, the set-valued attributes are sorted before hand. Sorting the relations themselves according to some lexicographical ordering is not necessary.

4.3 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, one could insert redundantly every tuple with a set-valued attribute a for every possible subset of a into the hash table. Since this results in an exponential 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? Assuming it to be possible, this generation results in an exponential runtime in the length of the signature. Hence, for small signatures, this approach seems reasonable whereas for large signatures it is unfeasible. Since the signature depends on the size of the sets, and 30 and more bits for a signature are common, generating all the signatures of possible subsets is unfeasible. This problem can be avoided by using only part of the signature. In order to derive a hash join for the subsetequal 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

Building the hash table 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] We call this the *partial signature*. This corresponds to the fixed prefix/suffix partitioning technique for signatures [23], except that we neglect the other part. Every tuple t of the relation to be hashed is now inserted into the hash table at address $partsig(t)$.

[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.

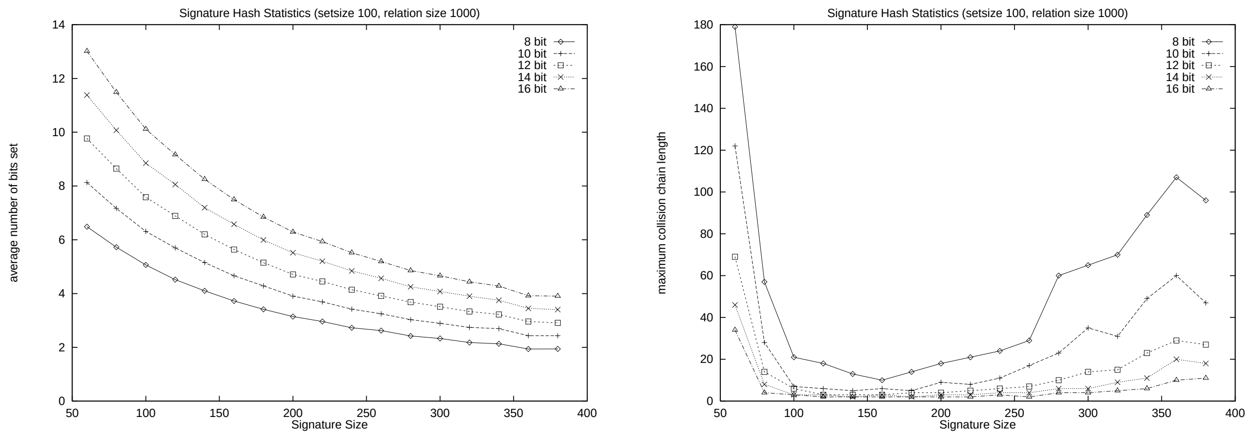


Figure 10: Tuning partial signature sizes

Tuning the parameters The signature size b and the partial signature size d are crucial tuning parameters since they determine via the collision chain lengths and via another factor (see the next step) the run time of the join algorithm. If b becomes very small, that is much smaller than the set size, then we can expect that every bit is set. Hence, many collisions occur. If b becomes very large, say 1,000 times the set size, we can expect that almost none of the d bits of the partial signature is set, if d is in a reasonable range from 6-16. Again, many collisions occur. 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 size. For a given set size, the problem is now to determine the optimal value for b. This b can be derived analytically as

$$b_{opt} = \frac{1}{1 - (\frac{1}{2})^{\frac{1}{r}}} \tag{7}$$

(See the appendix for a formal derivation.)

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 size of 100, the optimal value of b can be computed as $b_{opt} = 144.77$. Figure 10 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 to 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 7 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 build 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 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 (1) perform a comparison of the full signature and, if necessary, (2) an explicit set comparison to take care of false drops.

The partial signatures are exactly those bit patterns that have at most the bits of the signature of t_2 set. As an example consider that t_2 has the partial signature 0101. Then, we look up the entries 0000, 0100, 0001 and 0101 within the hash table. The generation of these partial signatures for subsets is done by using the idea of Vance and Maier [41]. If M is the partial signature of t_2, then the partial signatures S of all subsets can be generated by performing $S = M \& (S - M)$ until all partial signatures have been generated. (For further details see [41].)

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 $partsig(t)$ *mod* 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*.

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. 10). 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 illustrate this point by some experiments. Figure 11 gives the performance for several values for d–the partial signature length. As we see, for every relation size there seems to exist another optimal value for d. In general, the partial signature length should be the largest d such that 2^d is less than or equal to the cardinality of the hashed relation. For relations sizes larger than 1,000 we have to add an extra bit. (Adding one bit or not

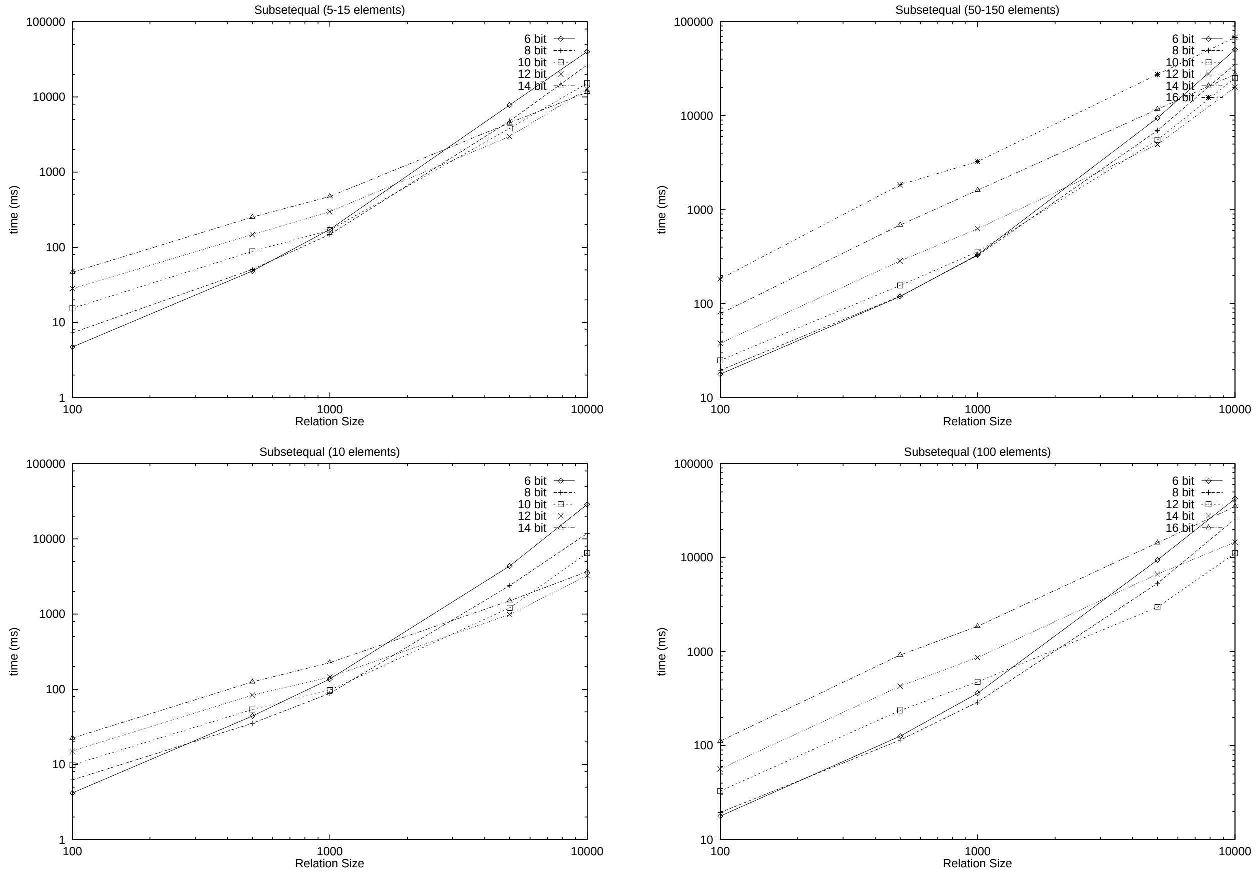


Figure 11: Performance of the hash join depending on the partial signature size

is not really crucial, since the performance increase is about 10%.) Given this fact, the indirect approach does not make much sense anymore, since a hash table size in the order of the relation size seems feasible.

The same factor, that is the number of hash table entries that have to be looked up in order to find all the joining tuples for a given tuple, makes it worth to consider deviations from b_{opt}. If we chose a signature length larger than b_{opt}, then the average number of bits set within the partial signature decreases. Though more collisions may occur, the number of lookups decreases. Experiments show that adding about 30% results in a better performance of at least 30% and at most a factor of 2.

4.4 Evaluation

Figure 12 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, the sort-merge and the tree join, as well as the hash join with the tuning 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 then allocates the signature bit vector dynamically. Only one signature is allocated for all tuples of the outer relations. It is cleared before reusage.

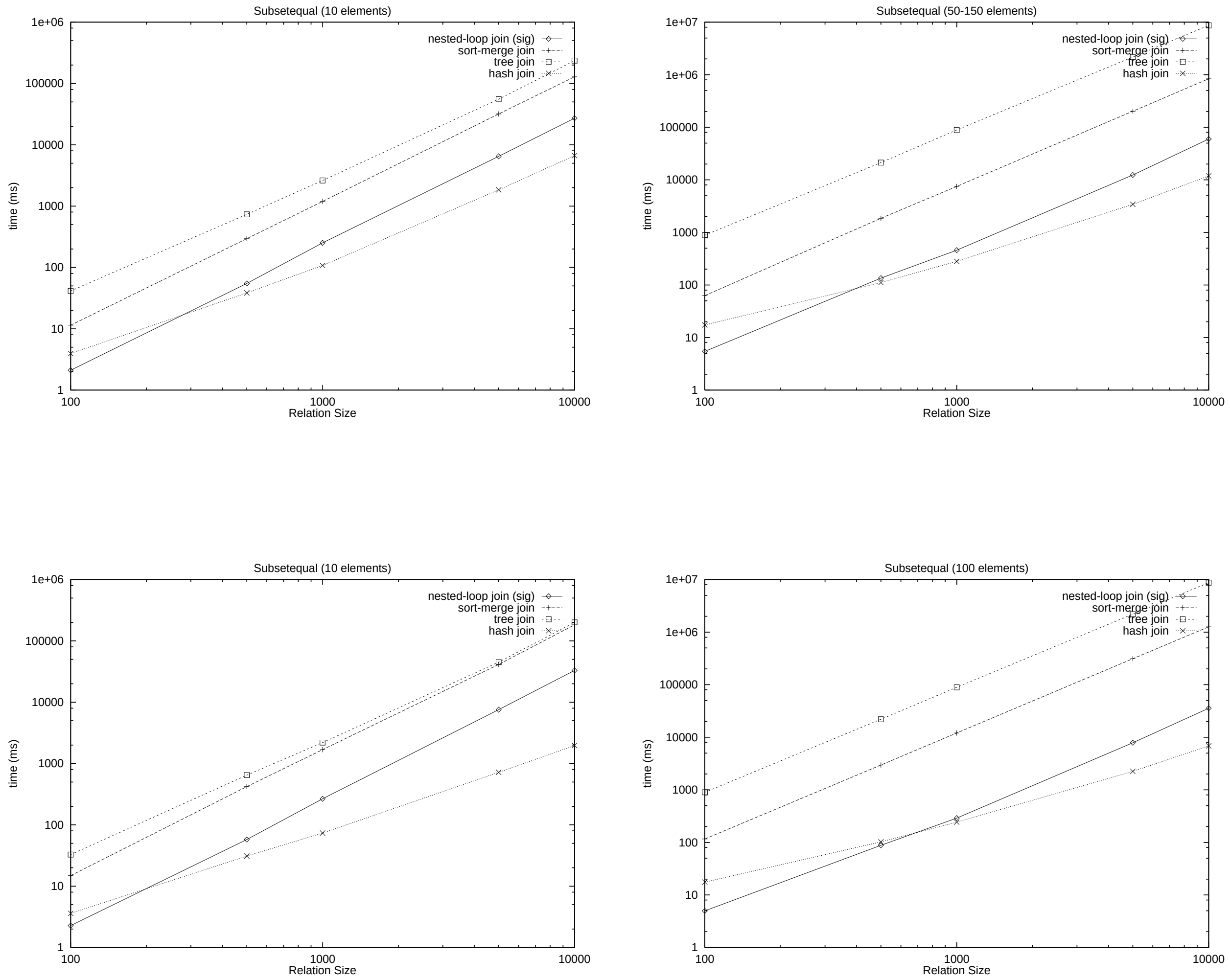


Figure 12: Performance of different join algorithms for $\subseteq$

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 shown before. Obviously, for larger relation sizes, the hash join algorithm performs best. Otherwise, the nested-loop algorithm is superior. However, the advantage of the hash join is not as big as might be expected after the results for set equality. This is due to the exponential number of hash table lookups. Nevertheless for relations containing 10.000 tuples, the hash join saves a factor of up to 10 for small sets and a factor of up to 5-6 for large sets.

The reason why the speed up for large sets is smaller than for small sets is that the signature comparison becomes more expensive with growing signatures. Since the signature comparison is executed only a quadratic number of times in the signature-based nested-loop join but an exponential number of times in the signature-hash join, the performance of the latter degrades more severe.

5 Conclusion

For the first time, this paper investigates join algorithms for join predicates based on set comparisons. More specifically, this paper treats the set equality and subset predicates. It has been shown that much 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. For set equality, the hash join gives two more orders of magnitude over the signature hash join resulting in very good performance. For the subsetequal predicate, the performance of the signature nested-loop join still yields an order of magnitude improvement. However, the hash join for the subsetequal predicate improves over the signature hash join only a factor of 5-10 depending on various parameters. Although this is a result that is not to be neglected, the question arises whether a better alternative exists. This is one issue for future research but there are more. First, join algorithms where the 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. Third, parallelizing these algorithms is an interesting issue by itself.

Acknowledgement We thank B. Rossi and W. Scheufele for careful reading of a first draft of the paper.

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] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] D. Lee and C. Leng. Partitioned signature files: Design issues and performance evaluation. *ACM Trans. on Information Systems*, 7(2):158–180, Apr. 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.

A Optimizing the parameters for the false drop probability

In this section we show, how the calculation of the optimal value for the number of bits in a signature can be derived.

When joining two relations, we do not distinguish between a query set (having r_s members on the average) and the sets in the database (having r_t members on the average). We assume, that all sets participating in the join contain r_t elements on the average.

$$\begin{aligned} d_{f=}(b,k,r_t,r_t) &\approx p_0(b,k,r_t,r_t)\cdot p_1(b,k,r_t,r_t) && (8)\\ &\approx \left(1-\left(1-\frac{k}{b}\right)^{r_t}\right)^t\cdot\left(1-\frac{k}{b}\right)^{r_t\cdot(b-t)} && (9)\end{aligned}$$

with t being the average number of bits set in a signature:

$$t = b\cdot\left(1-\left(1-\frac{k}{b}\right)^{r_t}\right) \tag{10}$$

thus

$$d_{f=}(b,k,r_t,r_t) \approx \left(\left(1-\left(1-\frac{k}{b}\right)^{r_t}\right)^{\left(1-\left(1-\frac{k}{b}\right)^{r_t}\right)}\cdot\left(1-\frac{k}{b}\right)^{r_t\cdot\left(1-\frac{k}{b}\right)^{r_t}}\right)^b \tag{11}$$

The derivative $d'_{f=}(b,k,r_t,r_t)$ of $d_{f=}(b,k,r_t,r_t)$ with respect to k yields (subsituting $x = 1-\frac{k}{b}$ in the result)

$$\begin{aligned} d'_{f=}(b,k,r_t,r_t) &= \left((1-x^{r_t})^{(1-x^{r_t})}\left(x^{r_t}\right)^{(x^{r_t})}\right)^b b\left(\frac{x^{r_t}r_t\ln\left(1-x^{r_t}\right)-x^{r_t}r_t\ln\left(x^{r_t}\right)}{bx}\right) && (12)\\ &= \left((1-x^{r_t})^{(1-x^{r_t})}\left(x^{r_t}\right)^{(x^{r_t})}\right)^b x^{(r_t-1)}r\ln\left(\frac{1-x^{r_t}}{x^{r_t}}\right) && (13)\end{aligned}$$

For $r_t > 0$ and $0 < k < b$:

$$\begin{aligned} d'_{f=}(b,k,r_t,r_t) = 0 &\Leftrightarrow \ln\left(\frac{1-x^{r_t}}{x^{r_t}}\right) = 0 && (14)\\ &\Leftrightarrow \frac{1-x^{r_t}}{x^{r_t}} = 1 && (15)\\ &\Leftrightarrow \frac{1}{2} = x^{r_t} && (16)\end{aligned}$$

Resubsituting $1-\frac{k}{b}$ for x we get

$$1-\frac{k}{b} = \left(\frac{1}{2}\right)^{\frac{1}{r_t}} \tag{17}$$

This means for the optimal values for k and b:

$$k_{opt} = b\left(1-\left(\frac{1}{2}\right)^{\frac{1}{r_t}}\right) \tag{18}$$

$$b_{opt} = \frac{k}{1-\left(\frac{1}{2}\right)^{\frac{1}{r_t}}} \tag{19}$$