Processing Joins with User-Defined Functions

Volker Gaede and Oliver Günther
Institut für Wirtschaftsinformatik
Humboldt-Universität zu Berlin
Spandauer Str. 1
10178 Berlin, Germany
{gaede,guenther}@wiwi.hu-berlin.de

Abstract

Most strategies for the computation of relational joins (such as sort-merge or hash-join) are facing major difficulties if the join predicate involves complex, user-defined functions rather than just simple arithmetic comparisons. In this paper, we identify a class of user-defined functions that can be included in a join predicate, such that a join between two sets R and S can still be computed efficiently, i.e., in time significantly less than $O(|R| \cdot |S|)$. For that purpose, we introduce the notion of the ϕ-function, an operator to process each set element separately with respect to the user-defined function(s) being used. Then any particular join query containing those functions can be computed by a variation of some traditional join strategy. After demonstrating this technique on a spatial database example, we present the results of a theoretical analysis and a practical performance evaluation.

Keywords

functional join, query processing, user-defined predicates, z-ordering, ϕ-function, extensible and object-oriented database system

1 Introduction

One of the most common operations in relational database systems is the relational join. Ullman [Ull88] gives the following definition:

Definition 1 (θ-Join) *The θ-join of relations R and S on columns i and j, written $R \overset{\bowtie}{}_{i\theta j} S$, where θ is an arithmetic comparison operator ($=, <$, and so on), is those tuples in the Cartesian product of R and S where the i-th component of R stands in relation θ to the j-th component of S.*

If θ corresponds to equality, the resulting operation is often called an *equijoin*. For the efficient computation of equijoins there exists a variety of strategies, including:

1. *Nested loop*: For each tuple $r \in R$, test each tuple $s \in S$ whether $r.i = s.j$. If yes, add the joined tuple (r, s) to the result.

2. *Sort-merge*: Sort R on column i and S on column j. Merge the sorted tables while testing for tuples $r \in R$ and $s \in S$ where $r.i = s.j$. For each matching pair, add the joined tuple (r, s) to the result.

3. *Hash-join*. Assume a hash table H on R's join column $R.i$. Use S's join column $S.j$ to hash each tuple $s \in S$ into H. If the corresponding hash bucket is not empty, test each tuple $r \in R$ in the bucket whether $r.i = s.j$. If yes, add (r, s) to the result.

4. *Index-supported*: Assume an index I on $R.i$. Sort S on $S.j$. For each different value v of $S.j$, use the index I to find all tuples $r \in R$ where $r.i = v$. For each corresponding tuple $s \in S$, add (r, s) to the result.

5. *Join index*: Precompute the join and store the tuple IDs of matching tuples in a special join index relation $JI = (ID_R, ID_S)$. The join between R and S or any subsets thereof can then be obtained by a simple lookup operation.

Note that in those descriptions, the roles of R and S may be exchanged, depending on efficiency considerations. Note also that some (but not all) of these strategies remain applicable for θ being an arithmetic comparison operator other than equality. For a more detailed overview see, for example, [ME92] or [Gra93].

While the equijoin is still the most common join operation, the relevance of more complex θ-operators is increasing steadily. Non-standard database applications such as CAD, geography, or multimedia require more advanced ways to connect different data sets, including spatial and temporal operators. To handle these requirements, an increasing number of commercial database systems allow users to introduce their own application-specific data types and operators. This kind of extensibility is also an essential feature of object-oriented database systems; it is often called *behavioral object-orientation* [Dit91].

Once introduced, these user-defined functions may of course be included in queries, e.g., as a θ-operator in a join query. Consider the following example that refers to two user-defined functions called `adjacent` and `area` (here we use the syntax of the object-oriented query language ZQL[C++], as defined in [Bla91]):

Query 1

```
set<NewObject> * result;
result = SELECT NewObject(b, r)
FROM Biotope * b IN Biotope_extent, Road * r IN Road_extent
WHERE adjacent(b, r)
&& area(b) >= 50;
```

With regard to efficient query processing, the main problem is that user-defined operators are black boxes for the optimizer. Several proposals have been made to address this problem. Graefe and Maier present an approach where a request can be sent to a class or object to reveal any relevant execution information [GM88]. Aberer and Fischer [AF93] recently proposed a scheme to optimize methods based on equivalences within a method algebra. The revelation scheme of the P/FDM system [JG91] is only applicable if the a canonical form of the function (e.g., its source code) is available to the optimizer. If the source code is not available, the optimizer usually resorts to a simple nested loop algorithm.

Hellerstein and Stonebraker [HS93] recently proposed an optimization scheme for expensive predicates based on ranking. Their approach tackles the problem in a practical

fashion but leaves space for more optimization, especially of more complex join predicates (see section 6).

Except for nested loop and join index, none of the other traditional strategies seems to be immediately applicable to joins with more complex, user-defined θ-operators. In the sequel of this paper, we will investigate how these strategies can actually be adapted to a wider range of joins than just joins involving simple arithmetic comparisons. For that purpose, in Section 2 we define a class of joins that is more general than Ullman's θ-join and that can also be applied in an object-oriented setting. We then define a subset of this class that can be processed efficiently, using variations of the tradional join processing strategies described above. In Section 3, these concepts are demonstrated on an example from spatial data management. Section 4 contains a brief theoretical analysis of the different strategies, and Section 5 verifies these results by means of a practical performance evaluation. Section 6 discusses some implementation issues, and Section 7 summarizes our results and gives directions for future research.

2 Functional Joins

In order to generalize Ullman's join definition, we propose the following definition of a *functional join.*

Definition 2 (Functional Join) *Let R_i $(1 \leq i \leq n)$ denote a nonempty set, and a_j $(1 \leq j \leq l)$ a (possibly complex) attribute of some R_i. Further let $\Phi(...)$ be a functional predicate defined on the domains $dom(a_1) \times dom(a_2) \times ... \times dom(a_l) \times c_1 \times ... \times c_m$ with c_k $(0 \leq k \leq m)$ from some arbitrary domain O_k $(O_0 = \emptyset)$. Then the functional join of the sets $R_1, ..., R_n$, denoted by*

$$\bowtie_{\Phi(a_1,a_2,...,a_l,c_1,...,c_m)} (R_1, ..., R_n)$$

is defined as those elements of the Cartesian product $R_1 \times R_2 \times ... \times R_n$ where $\Phi(...)$ is satisfied.

Several remarks:

1. The sets R_i may in particular be relations and the Φ-predicate may be an arithmetic comparison. In that case, one obtains Ullman's definition of a θ-join for $l = n = 2$ and $m = 0$.

2. With the variables c_k one can parameterize a given Φ-predicate. For example, in order to find all pairs of objects in a spatial database whose distance is less than d, it suffices to define *one* Φ-predicate and represent d as a parameter c_k.

3. The *Ojoin* defined by Shaw and Zdonik [SZ90] in the context of object-oriented databases is also a special case of a functional join. We recall that the *Ojoin* is essentially the Cartesian product of two collections R and S of objects, followed by a selection of tuples:

   $$Ojoin(R, S, A_1, A_2, p) = \{< A_1 : r, A_2 : s > \mid r \in R \wedge s \in S \wedge p(r, s)\}$$

   where p is a first order predicate defined over objects from R and S.

Our intention now is to identify a maximal subset of Φ-predicates for which the functional join can be processed *efficiently*, i.e., in time significantly less than $O(\prod_i |R_i|)$. We will call this subset ϕ-predicates ($\phi \subset \Phi$). The key idea is that for those ϕ-predicates one can compute certain data for the participating sets R_i separately. For any particular join that contains the ϕ-predicate, this data can then be used to reduce the complexity of the join query significantly. For this purpose it is necessary to undergo the following transition from a ϕ-predicate to a corresponding ϕ-function.

Definition 3 (ϕ-predicate, ϕ-function) *The Φ-predicate $\Phi(a_1, ..., a_l, c_1, ..., c_m)$ is called a ϕ-predicate iff for some j $(1 \leq j \leq l)$ there exists a function*

$$\phi_j : a_1, ..., a_{j-1}, a_{j+1}, ..., a_l, c_1, ..., c_m \rightarrow \{\xi_j^1, ..., \xi_j^k\} \ (0 \leq k \leq D_j)$$

such that $\Phi(...)$ is true iff $a_j \in \{\xi_j^1, ..., \xi_j^k\}$. ϕ_j is called a ϕ-function, and D_j is called the grade of ϕ_j.

Several details should be noted:

1. There may be more than one possible value of a_j, such that $\Phi(...)$ is true ($k > 1$), but the total number of possible values has to be bounded ($k \leq D_j$).
2. $k = 0$ iff there exists no suitable a_j.
3. We restrict our consideration to $\xi_j^1, ..., \xi_j^k$ denoting discrete non-complex values (in contrast to objects).

For simplicity we restrict our future presentation to the case $n = 2$ and call the participating sets R and S (rather than R_1 and R_2). A generalization to higher values of n is straightforward unless noted otherwise.

In the next section, we will demonstrate the use of the ϕ-function on a spatial database example. Note, however, that the applicability of ϕ-functions is not restricted to spatial databases. Generally speaking, ϕ-functions are useful if some transformation is necessary to evaluate a given join predicate. Examples include:

1. CAD/CAM databases: Given a set of (possibly complex) parts, one wants to find all pairs of parts that fit together. Instead of testing all possible pairs, one could apply the ϕ-function to characterize possible matches and then search for them.
2. Scientific databases: Given some spectra, one wants to find all matching spectra that are shifted by a given constant for a given accuracy. The ϕ-function could transform these spectra into a search-able representation.
3. Encrypted databases: In some applications, data maybe encrypted and must decrypted to be evaluated. Instead of decrypting each tuple to perform a requested predicate evaluation, one could transform the interesting tuples by means of the ϕ-function and then search the result.

3 An Example

3.1 Z-Ordering

A spatial access method well-known in the field of geographic databases is the *z-ordering* proposed by Orenstein [OM88]. Z-ordering is based on the successive regular subdivision of a given area into smaller rectangles or squares. Orenstein uses the Peano order to sort the cells of the resulting grid, and a bit-interleaving scheme to attach a unique identifier (the *z-value*) to each one of them. A given (n-dimensional) object can then be approximated by a collection of cells and represented as the set of corresponding z-values (fig. 1). Note that z-ordering does not completely preserve spatial proximity. For a comprehensive discussion of z-values and their possible usage in the context of query processing see, for example, [OM88] or [Gae93].

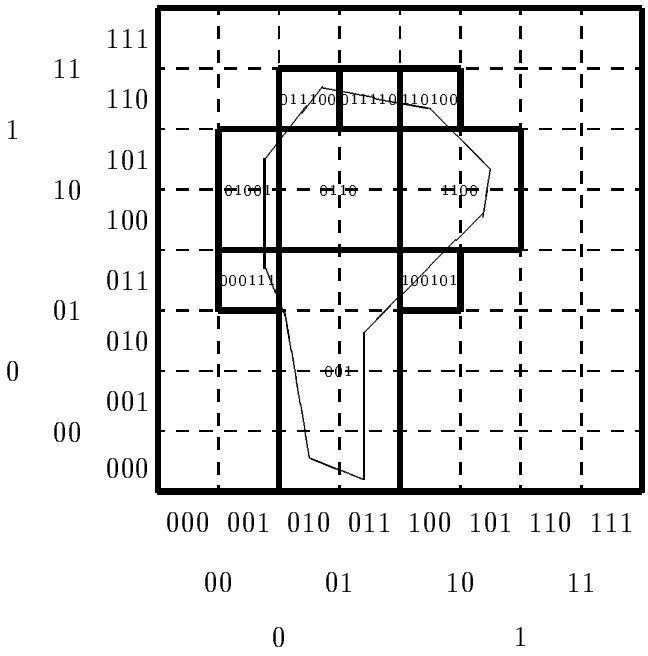


Figure 1: A two-dimensional object decomposed into a number of cells with their corresponding z-values.

One of the advantages of z-ordering compared to many other spatial access methods is that it is possible to compute certain functions directly from the z-values. For example, a common query in geographic applications concerns the adjacency of objects of two different sets, such as:

Find all apartments of the city that are adjacent to a factory.

Query 2

```
SELECT a, f
FROM Apartment * a IN city_extent, Factory * f IN industry_extent
WHERE adjacent(a, f);
```

or more general

Query 3

```
SELECT r, s
FROM R r, S s
WHERE adjacent(r, s);
```

This query is a functional join of two sets R and S with `adjacent` being the Φ-predicate. Due to the spatial nature of the `adjacent` operator, it is also called a *spatial join* [BKS93, Gü93]. If the operands are regular cells, as they result from a z-order partitioning of space, `adjacent` fulfills the criteria for being a ϕ-predicate, which can be shown as follows. Two cells are considered *adjacent* (or *neighbors*) if they share at least one common border (4-adjacency). For each cell $O_i \in R$, respectively its z-value, we can individually compute the z-values of its neighbors. The number of neighbors of a given cell is variable, but bounded. The function ϕ_R to perform this computation therefore is a ϕ-function with $D_R = 4$ (we restrict our discussion to the neighbors of one level; the generalization is straightforward), and the resulting set of z-values corresponds to the set $\{\xi_R^1, ..., \xi_R^k\}$. A function ϕ_S to compute the neighbors of each cell $O_i \in S$ can be defined analogously. Due to the commutativity of `adjacent`, ϕ_R and ϕ_S are in fact identical.

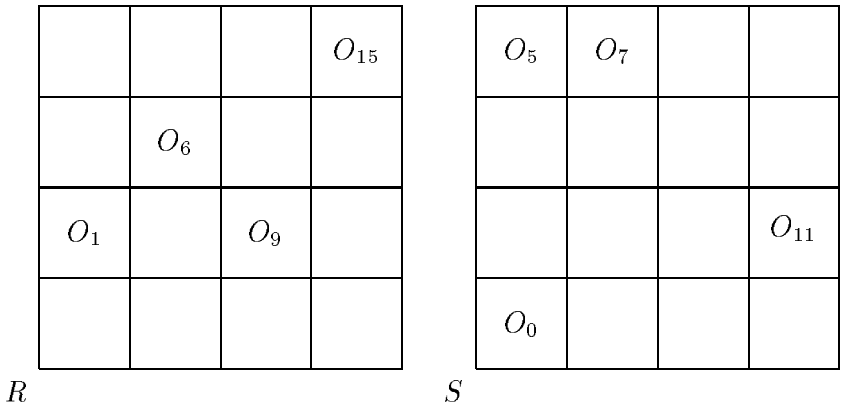


Figure 2: Two given sets R and S to be joined as specified in query 3.

3.2 Sort-Merge Join

One can now easily transform the problem into a format that allows one to apply a variant of the sort-merge join algorithm. For that purpose it is necessary to assign a

unique identifier to each adjacency between two given cells. This can be achieved by bit-interleaving the z-values of the neighbors in question, as shown on the example given in figure 2.

The neighbors of cell O_6 (0x0110)[1] are the cells O_3, O_4, O_7, O_{12} (0x0011, 0x0100, 0x0111, 0x1100). We generate a unique number for each individual adjacency relationship by taking the first digit of the smaller numeric value of the two z-values, e.g.,

```
interleave(6,3) = interleave(0x0110, 0x0011)
                = interleave(0x0011, 0x0110)
                = 0x00011110
                = 30
```

These values can be stored in sorted tables T_R and T_S, together with backpointers to the corresponding cells (see figure 3). One can then apply the sort-merge algorithm to compute the join: If and only if two numbers occur in both tables, the corresponding cells are neighbors. Note that it is easily possible to extend this algorithm to handle more than two sets.

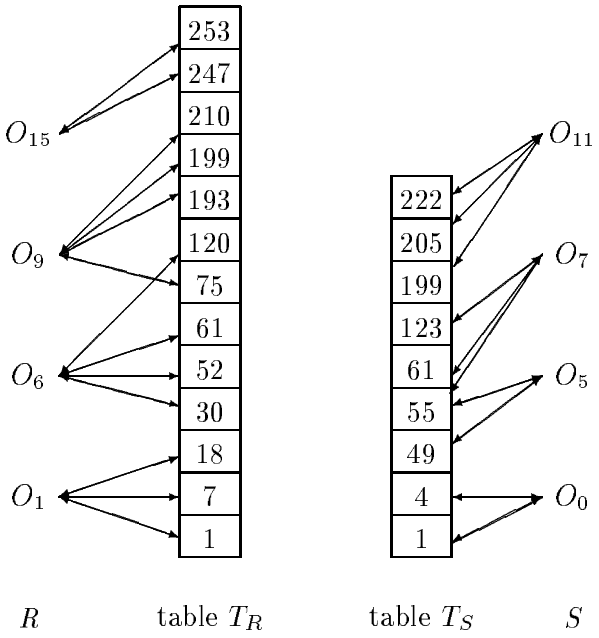


Figure 3: Result of the precomputation for the given sets R and S and the `adjacent` function. The connection between objects and computed values is maintained by backpointers.

The tables T_R and T_S can be computed directly from the corresponding index files without touching the objects. If these files are sorted by z-value, one can obtain the

[1]The prefix $0x$ denotes the binary representation of the z-value.

tables by appending the computed values, followed by a constant number of exchanges per z-value. We therefore assume linear time complexity for computing and inserting the resulting values. With the parameters

n, m	number of objects in R and S, respectively
D	grade of the ϕ-function

we obtain the following cost formula[2]

$$\begin{aligned} C &= \underbrace{O(n)+O(m)}_{\text{scanning the index file}} + \underbrace{O(D\cdot n)+O(D\cdot m)}_{\text{computation time}} + \underbrace{O(D\cdot n)+O(D\cdot m)}_{\text{insertion time}} + \underbrace{O(D\cdot n)+O(D\cdot m)}_{\text{merge time}} \\ &= O(D\cdot(n+m)) \end{aligned}$$

3.3 Hash-Join

Hash-join algorithms have been shown to be competitive with sort-merge and index-supported joins, and if the whole hash-table can be held in main memory, they are usually superior. Next we adopt the hash-join algorithm to compute the neighborhood query with the hash function being identity.

Since the cardinality of the given sets is equal ($n = m$), there is no advantage of processing one of the two relations first. We start with set R and compute the possible neighbors of the given cells. The possible neighbors of O_6, for example, are O_3, O_4, O_7, and O_{12}. Inserting these values into a table with backpointers to the objects results in the structure shown in figure 4. Assuming that the table can be held in primary memory, one could then find the neighbors of cells in S by applying the same hash function to each element in S.

The time complexity of this strategy is

$$\begin{aligned} C &= \underbrace{O(n)}_{\text{scanning the index file of R}} + \underbrace{O(D\cdot n)}_{\text{computation time}} + \underbrace{O(D\cdot n)}_{\text{insertion time}} + \underbrace{O(m)}_{\text{scanning the index file of S}} + \underbrace{O(m)}_{\text{lookup time}} \\ &= O(m+Dn) \end{aligned}$$

3.4 Index-Supported Technique

If an index exists on the z-value columns, it is also possible to use an index-supported technique. In this case, the index would be used in a similar manner as the hash function in the hash-join algorithm. If h denotes the height of the index tree, we obtain the following time complexity:

$$\begin{aligned} C &= \underbrace{O(n)}_{\text{scanning the index file}} + \underbrace{O(D\cdot n)}_{\text{computation time}} + \underbrace{O(D\cdot n)}_{\text{insertion time}} + \underbrace{O(h\cdot D\cdot n)}_{\text{search time}} \\ &= O(h\cdot D\cdot n) \end{aligned}$$

[2] We assume in this formula that all operations must be carried out for both R and S. This may not always be necessary, in which case one may obtain a lower cost such as $O(n + D\cdot m)$ using sort-merge.

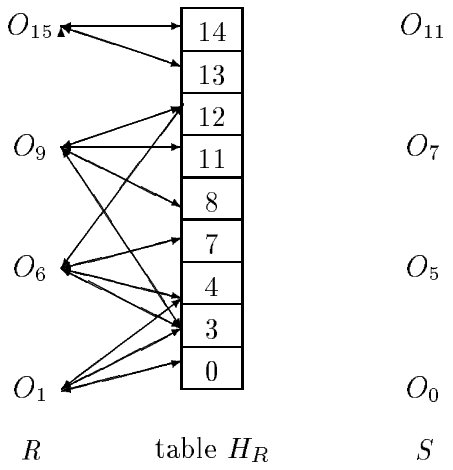


Figure 4: Table H_R containing the precomputed values with backpointers to the cells O_i. Each value of S now has to be hashed into H_R as well.

4 Theoretical Analysis

In this section, we will provide a detailed analysis of four algorithms to compute a functional join between two sets R and S of objects. The strategies investigated are:

1. sort-merge,
2. hash-join,
3. index-supported, and
4. nested loop.

Our analysis of the join index strategy has not been included in this paper because its applicability highly depends on the update ratio, whereas the complexity C_Φ of the ϕ-function has, of course, no impact on the join performance. Further details can be found in a forthcoming technical report.

4.1 Parameters

X	given set ($X \in \{R, S\}$)
P	page size (Kbytes)
k	key size (bytes)
p	block pointer size (bytes)
τ_X	record size in set X (bytes)
D	grade of the ϕ-function
$N = \|R\|$	number of objects in R
$M = \|S\|$	number of objects in S
ζ_X	number of index entries for set X
π	average page occupancy factor
κ	compression factor
α	selectivity of the join
μ_X	average number of records stored on a disk page $\lceil \frac{P\pi}{\tau_X} \rceil$
μ_ζ	average number of index records stored on a disk page
C_ϕ	time to compute ϕ-function
C_{cmp}	time for a comparison
C_{in}	time to insert an item
C_{IO}	time per I/O operation
C_{lkup}	time to look up one item
C_{mv}	time to move a record in memory
C_{mrg}	time to merge two or more sets
C_{srt}	time for sorting

4.2 Basic Formulae

In this section we provide the basic formulae to be used in the sequel.

1. *Sorting time*:
   The time to sort n records is assumed to be

$$C_{srt}(n) \quad = \quad n \cdot \log_2(n) \cdot C_{cmp} + C_{mv} \cdot n$$

2. *Record access*:
   To access k records randomly distributed in a file of n records stored in m pages, the expected number of page accesses is given by the Yao formula

$$\begin{aligned} Y(k, m, n) &= m \cdot \{1 - \prod_{i=1}^{k} \frac{n - (n/m) - i + 1}{n - i + 1}\} \\ &= m \cdot \{1 - \prod_{i=1}^{k} (1 - \frac{n}{m(n - i + 1)})\} \\ &\approx m \cdot \{1 - \exp(-k/m)\} \end{aligned}$$

   Note that this formula assumes that the scheduling of the page accesses is optimal, i.e., no page is read more than once. Note also that the number of accessed pages is nearly independent of the number of records.

3. *Selectivity*:

   The selectivity of the join is defined as

$$\alpha = \frac{|R \bowtie_\Phi S|}{|R| \cdot |S|}$$

   The selectivity for one set, say R, is defined as

$$\alpha_R = \frac{|R \ltimes S|}{|R|}$$

   It is not possible to provide an exact formula for the relationship between α_X and α, since this strongly depends on the given data. Nevertheless, it is possible to deduce the following inequality:

$$max(\frac{\alpha_R}{|S|}, \frac{\alpha_S}{|R|}) \leq \alpha \leq \alpha_R \cdot \alpha_S$$

4. *Number of index entries and index size*:

   (a) dense index (each object has its own index entry):

$$\zeta_X = |X|$$

   (b) clustered index:

$$\zeta_X = |X| \frac{\tau_X}{P \cdot \pi}$$

   Here we assume only one pointer per block.

   The number of disk pages necessary to store the index is:

$$\zeta_X \frac{(k+p)}{P \cdot \pi} = \frac{\zeta_X}{\mu_\zeta}$$

5. *Insertion time*:
   The time to insert n records with Ξ different ordering keys is:

$$C_{in}(n, \Xi) = C_{lkup}(n, \Xi) + n \cdot C_{mv}$$

   If the first and the second argument are equal (i.e., $n = \Xi$), we will omit the latter.

6. *Lookup time*:

$$C_{lkup}(n, \Xi) = n \cdot \log_2(\Xi) \cdot C_{cmp}$$

7. *Page accesses:*
The number of page accesses required to find k records on m pages with n records each, using a B$^+$-tree with height h, can be computed as follows.

$$I_b(h,k,m,n) \quad = \quad \sum_{i=0}^{h-1} Y_i$$

with

$$\begin{aligned} Y_{-1} &\stackrel{def}{=} k \\ Y_0 &\stackrel{def}{=} Y(k,m,n) \\ Y_i &\stackrel{def}{=} Y(Y_{i-1}, \lceil \frac{m}{\mu_{\zeta_X}^i} \rceil, \frac{m}{\mu_{\zeta_X}^{i-1}}) \end{aligned}$$

Note that in this formula it is assumed that the average number of index records on each page is equal to the fan-out, and that the root page of the index tree resides in main memory.

(a) clustered index (assumption: $h = 2$):

$$\begin{aligned} I(2,k,m,n) = I_b(2,k,m,n) &= Y_0(Y_{-1},m,n) + Y_1(Y_0, \frac{m}{\mu_\zeta}, m) \\ &= \underbrace{Y(k,m,n)}_{Y_0} + Y(Y_0, \frac{m}{\mu_\zeta}, m) \end{aligned}$$

(b) dense index (assumption: $h = 3$):

$$\begin{aligned} I(3,k,m,|X|) &= Y(k,m,|X|) + \underbrace{Y(k, \frac{|X|}{\mu_\zeta}, |X|)}_{Y_0} + Y(Y_0, \frac{|X|}{\mu_\zeta^2}, \frac{|X|}{\mu_\zeta}) \\ &= Y(k,m,|X|) + I_b(2,k, \frac{|X|}{\mu_\zeta}, |X|) \end{aligned}$$

8. *Merge time (without insertion)*:
The time to merge n sorted records is:

$$C_{mrg}(n) \quad = \quad n \cdot C_{cmp}$$

9. *Average number of pages required to store the objects in X*:

$$||X|| \quad = \quad \frac{|X|}{\mu_X}$$

4.3 Cost Formulae

4.3.1 Sort-Merge

Algorithm 1 (Sort-Merge)

```
1  for each index entry in IR do
2     compute the phi-function and
3     insert the resulting values in table tr;
4  done;
5  for each index entry in IS do
6     compute the phi-function and
7     insert the resulting values in table ts;
8  done;

9  Merge both tables and
10 insert for each matching tuple the corresponding pointers
11 in the result table (if they are not already there);

12 Perform a semi-join for the result table with R;
13 Perform a semi-join for the result table with S;
```

Based on the formulae given in the previous section, we obtain the following cost for the various steps.

1. Step 1 (line 1-4, line 5-8):

$$C^X_{1gc|s}(\zeta_X) \quad = \quad \lceil \frac{\zeta_X}{\mu_\zeta} \rceil C_{IO} + \zeta_X \cdot D \cdot C_\phi + C_{in}(\zeta_X(D+1), \kappa_X \zeta_X(D+1))$$

2. Step 2 (line 9-11): Since it is likely that during the insertion a number of ξ values occur more than once, we introduce the parameter $\kappa \leq 1$ to denote that the number of items which must be compared during the merge step is smaller than the total number of items inserted.

$$C^{R\&S}_{2mg}(\zeta_R, \zeta_S, |R|, |S|) \quad = \quad C_{mrg}((\kappa_R \zeta_R + \kappa_S \zeta_S)(D+1)) + C_{in}(\alpha |R| \cdot |S|)$$

3. Step 3 (line 12, line 13):

$$C^X_{sj} \quad = \quad \lceil Y(\alpha_X \cdot |X|, \lceil \frac{|X|}{\mu_X} \rceil, |X|) \rceil \cdot \mu_X \cdot C_{cmp} + \alpha_X \cdot |X| \cdot C_{mv}$$

The total cost of Algorithm 1 is then

$$C_{mg} \quad = \quad C^R_{1gc|s} + C^S_{1gc|s} + C^{R\&S}_{2mg} + C^R_{sj} + C^S_{sj}$$

4.3.2 Hash-Join

Algorithm 2 (Hash-Join)

```
1  for each index entry in IR do
2      compute the phi-function and
3      insert the resulting values in table tr;
4  done

5  for each index-entry of IS do
6      perform a table lookup;
7      if value found then
8           insert page pointers in result table;
9      endif
10 done

11 Perform a semi-join for the result table with R;
12 Perform a semi-join for the result table with S;
```

1. Step 1 (line 1-4):

$$C_{1hgc|s}(\zeta_R) \quad = \quad \lceil \frac{\zeta_R}{\mu_\zeta} \rceil \cdot C_{IO} + \zeta_R \cdot D \cdot C_\phi + C_{in}(\zeta_R(D+1), \kappa_R \zeta_R(D+1)) = C^R_{1gc|s}(\zeta_R)$$

2. Step 2 (line 5-10):

$$C_{2hg}(\zeta_S, |S|, |R|) \quad = \quad \lceil \frac{\zeta_S}{\mu_\zeta} \rceil \cdot C_{IO} + C_{lkup}(\zeta_S, \kappa_R \zeta_R(D+1)) + C_{in}(\alpha |S| \cdot |R|)$$

3. Step 3 (line 11 or 12): same as for Algorithm 1.

The total cost of Algorithm 2 is:

$$C_{hg} \quad = \quad C^R_{1gc|s} + C_{2hg} + C^R_{sj} + C^S_{sj}$$

4.3.3 Index-Supported

Algorithm 3 (Index-Supported)

```
1  for each index entry in IR do
2      compute the phi-function and
3      insert the resulting values in table tr;
4  done

5  for each value in the table tr do
6      use index of S to find matching
7      objects and their page pointer;
8      Insert these page pointers in the result table;
9  done

10 Perform a semi-join for the result table with R;
11 Perform a semi-join for the result table with S;
```

1. Step 1 (line 1-4): same as for Algorithm 2

2. Step 2 (line 5-9):

$$C_{2is}(\zeta_S) \quad = \quad \lceil I(h, \kappa_R \zeta_R (D+1), \lceil \frac{\zeta_S}{\mu_\zeta} \rceil, \zeta_S) \rceil \cdot C_{IO} + C_{in}(\alpha |S| \cdot |R|)$$

   It is likely that the first argument of the Yao function (k) is greater than the last one (n), especially if the relationship is complex, i.e., D is large. In such a case the number of necessary page accesses is m, and thus nearly equal to the hash-join method.

3. Step 3 (line 10, line 11): same as for Algorithm 1

The total cost of Algorithm 3 is then:

$$C_{is} \quad = \quad C^R_{1gc|s} + C_{2is} + C^R_{sj} + C^S_{sj}$$

4.3.4 Nested Loop

Algorithm 4 (Nested Loop)

```
1  Load index of set R;
2  Load index of set S;

3  for each element r of set R do
4      foreach element s of set S do
5          if phi(s, r) then
6              place in result relation;
7          endif
8      done
9  done

10 Perform a semi-join for the result relation with R;
11 Perform a semi-join for the result relation with S;
```

1. Step 1 (line 1, line 2):

$$C^X_{1ngc|s}(\zeta_X) \quad = \quad \lceil \frac{\zeta_X}{\mu_\zeta} \rceil C_{IO}$$

2. Step 2 (line 3-9):

$$C^{R\&S}_{2ng}(\zeta_R, \zeta_S, |R|, |S|, \alpha) \quad = \quad (\zeta_R \cdot \zeta_S) C_\phi + C_{in}(\alpha |R| \cdot |S|)$$

3. Step 3 (line 10, line 11): same as for Algorithm 1

Total cost:

$$C_{ng} \quad = \quad C^R_{1ngc|s} + C^S_{1ngc|s} + C^{R\&S}_{2ng} + C^R_{sj} + C^S_{sj}$$

4.3.5 Comparison (Join)

We recall the total cost formulae

$$\begin{aligned}
C_{mg} &= C^R_{1gc|s} + C^S_{1gc|s} + C^{R\&S}_{2mg} + C^R_{sj} + C^S_{sj} && \text{(sort-merge)}\\
C_{hg} &= C^R_{1gc|s} + C_{2hg} + C^R_{sj} + C^S_{sj} && \text{(hash-join)}\\
C_{is} &= C^R_{1gc|s} + C_{2is} + C^R_{sj} + C^S_{sj} && \text{(index-supported)}\\
C_{ng} &= C^R_{1ngc|s} + C^S_{1ngc|s} + C^{R\&S}_{2ng} + C^R_{sj} + C^S_{sj} && \text{(nested loop)}
\end{aligned}$$

For the comparison nested loop (C_{ng}) vs. sort-merge (C_{mg}), we obtain:

$$\begin{aligned}
C_{ng} - C_{mg} &= -\zeta_R D C_\phi - C_{in}(\zeta_R(D+1), \kappa_R\zeta_R(D+1))\\
&\quad -\zeta_S D C_\phi - C_{in}(\zeta_S(D+1), \kappa_S\zeta_S(D+1))\\
&\quad -C_{mrg}((\kappa_R\zeta_R + \kappa_S\zeta_S)(D+1))\\
&\quad +\zeta_R\zeta_S C_\phi\\
&\geq -(\zeta_R + \zeta_S)\cdot\\
&\quad \underbrace{(D\cdot C_\phi + (D+1)\cdot(2\cdot\log_2(\max(\kappa_R\zeta_R, \kappa_S\zeta_S)\cdot(D+1))\cdot C_{cmp} + C_{mv}))}_{\Omega}\\
&\quad +\zeta_R\zeta_S\cdot C_\phi\\
&= \zeta_R\zeta_S\cdot C_\phi - (\zeta_R + \zeta_S)\cdot\Omega
\end{aligned}$$

I.e., if the values of ζ_R and ζ_S are sufficiently large, the sort-merge strategy is superior to the nested loop approach. For very small sets or for a selection, the sort-merge strategy is usually not competitive to nested loop. However, the larger the participating sets are, the more advantageous the sort-merge strategy becomes.

For the comparison nested loop (C_{ng}) vs. hash-join (C_{hg}), we obtain:

$$\begin{aligned}
C_{ng} - C_{hg} &= -\zeta_R D C_\phi - C_{in}(\zeta_R(D+1), \kappa_R\zeta_R(D+1)) - C_{lkup}(\zeta_S, \kappa_R\zeta_R(D+1))\\
&\quad +\zeta_R\zeta_S C_\phi\\
&= -\zeta_R \underbrace{(DC_\phi + (D+1+\frac{\zeta_S}{\zeta_R})\log_2(\kappa_R\zeta_R(D+1))\cdot C_{cmp} + (D+1)\cdot C_{mv})}_{\Lambda}\\
&\quad +\zeta_R\zeta_S C_\phi\\
&= \zeta_R\zeta_S C_\phi - \zeta_R\cdot\Lambda
\end{aligned}$$

The formula shows that the cost of the hash-join is linear in ζ_R (or ζ_S), whereas the nested loop costs are proportional to $\zeta_R\cdot\zeta_S$, and therefore not competitive. A closer inspection also reveals that the major factor Λ is multiplied by ζ_R, unlike the corresponding factor Ω in the case of sort-merge, which is multiplied by the sum $\zeta_R + \zeta_S$.

For the comparison nested loop (C_{ng}) vs. index-supported join (C_{isg}), we obtain:

$$\begin{aligned}
C_{ng} - C_{isg} &= -\zeta_R D C_\phi + C_{in}(\zeta_R(D+1), \kappa_R\zeta_R(D+1))\\
&\quad -\lceil I(h, \kappa_R\zeta_R(D+1), \lceil\frac{\zeta_S}{\mu_\zeta}\rceil, \zeta_S)\rceil\cdot C_{IO}
\end{aligned}$$

$$
\begin{aligned}
& +\zeta_R\zeta_S C_\phi \\
\geq\ & -\zeta_R(D \cdot C_\phi + (D+1)\log_2(\kappa_R\zeta_R(D+1))C_{cmp} + (D+1)C_{mv} + \\
& +h \cdot \lceil \frac{\zeta_s}{\mu_\zeta} \rceil \frac{1}{\zeta_R} \cdot C_{IO}) \\
& +\zeta_R\zeta_S C_\phi
\end{aligned}
$$

Again one finds that for the index-supported strategy the time dependency on ζ_R (or ζ_S) is essentially linear, and that nested loop is therefore not competitive. Note that the total cost of hash-join and of the index-supported approach are almost identical, which is in accordance with our empirical results.

5 Empirical Results

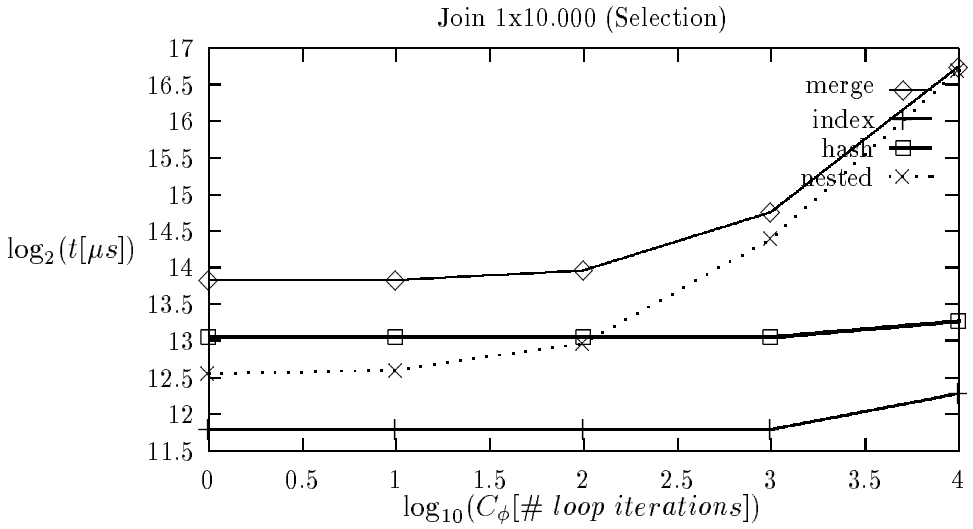
All strategies have been implemented on a Sun 10/40 for a page size of 512 KB. Due to some restrictions in the measurement utilities, the finest time granularity of our measurements is 16 ms.

As can be seen from the detailed analysis, there are several parameters that determine the overall performance of the different join strategies. First, there are of course the cardinalities of the participating sets, $|R|$ and $|S|$. A second important parameter is the time C_ϕ for evaluating the Φ-predicate (or ϕ-function). Third, for the sort-merge, hash-join and index-supported strategies, the parameter D (cardinality of the ξ-set) is important as well for the overall performance. We investigated the time dependency of the join with respect to these parameters and found our theoretical estimations confirmed in general, although there is no exact accordance.

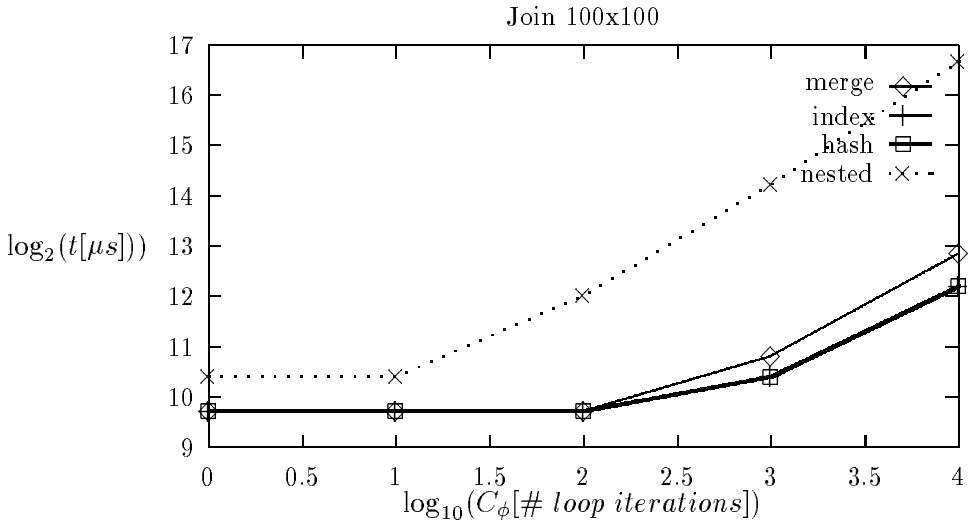
To derive the dependency of the join performance t on the time C_ϕ to compute the predicate Φ, we simulated the variable time consumption of Φ by a simple loop. The following diagrams plot $\log_{10} C_\phi$ (measured in the number of loop iterations) on the x-axis and $\log_2 t$ (measured in **micro**seconds) on the y-axis.

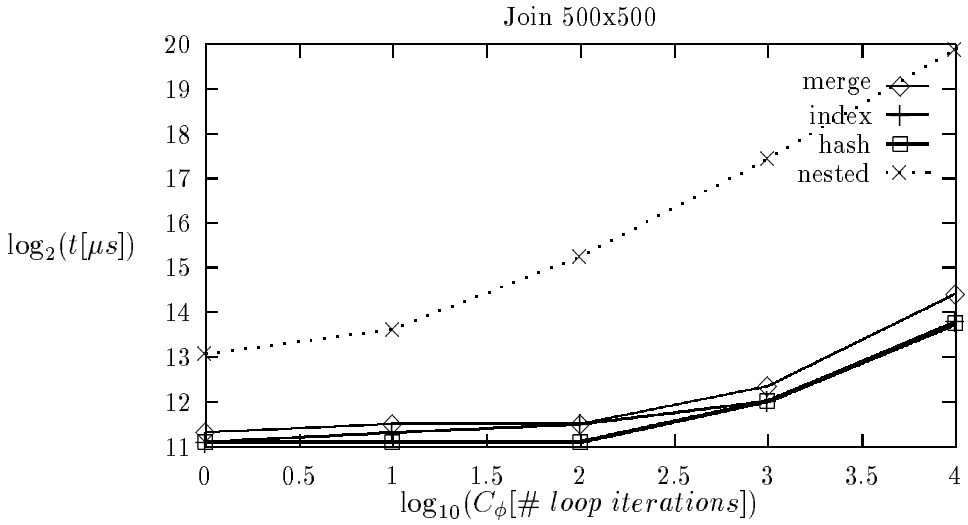
5.1 Dependency on C_ϕ

In this subsection the cardinality of the ξ-sets remains fixed at $D = 5$. The notation "Join $\delta_R \times \delta_S$" denotes that set R has δ_R elements and set S has δ_S elements. The case "Join $1 \times \delta$" can usually be interpreted as a selection operation, depending on the semantics of the Φ-predicate.



As expected, sort-merge is inferior to the other strategies, whereas the index-supported method shows the best performance. Furthermore, the time consumption for the index-supported as well as for the hash-join strategy is nearly independent of C_ϕ. Finally, it is interesting to note that the nested loop strategy is superior to the hash-join method for small values of C_ϕ.

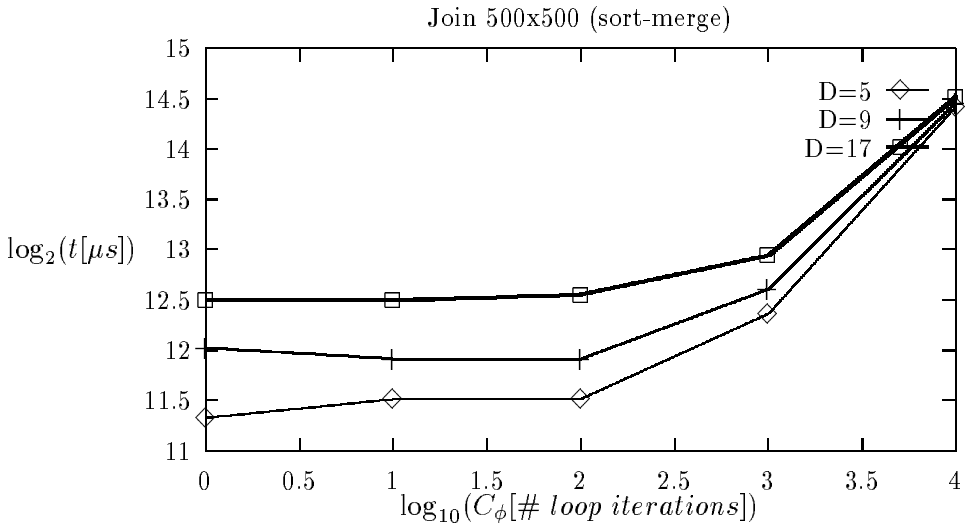


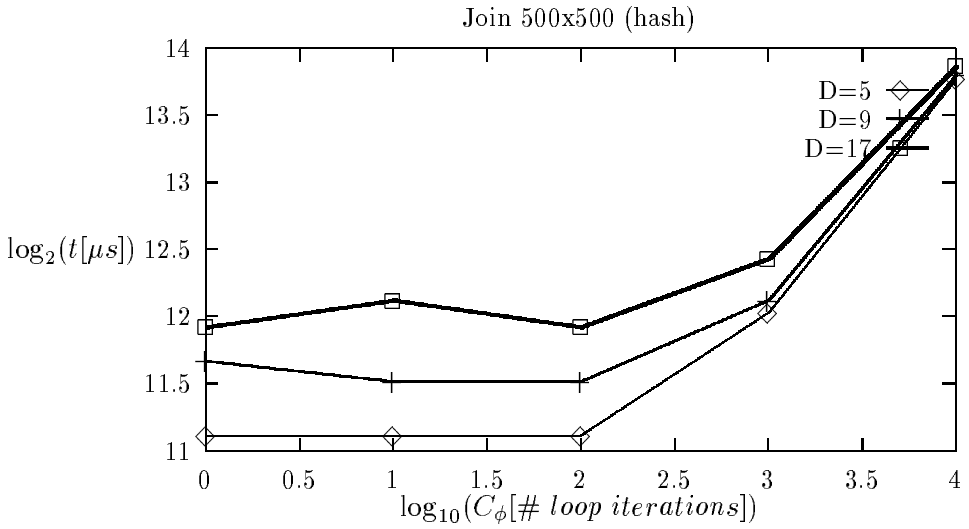


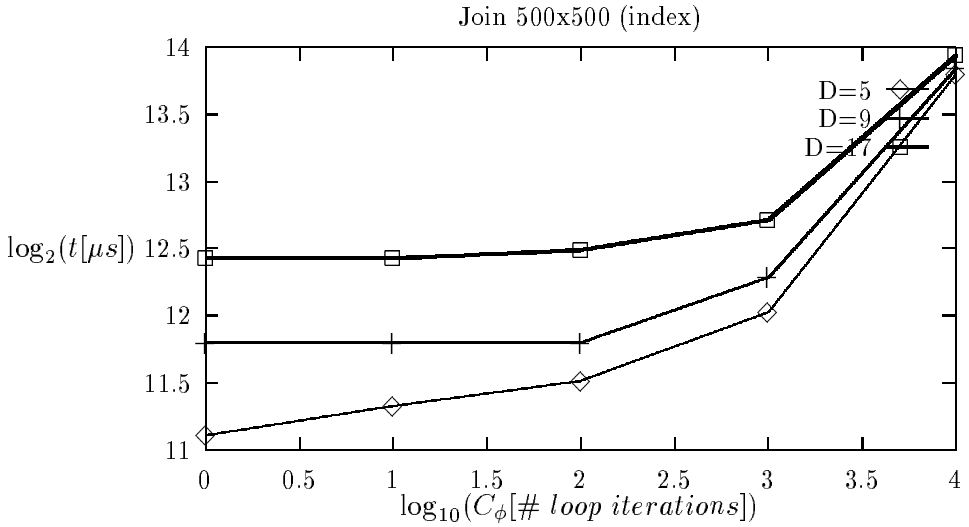
The above diagrams confirm our prediction that the nested loop method is not competitive with the other methods. The sort-merge strategy performs slightly worse than the hash-join or index-supported strategy for large values of C_ϕ. The higher the complexity C_ϕ, however, the more the sort-merge algorithm falls behind, until its performance is only about half that of the hash-join or index-supported strategy.

5.2 Dependency on D

The performance of the strategies also depends on the parameter D, i.e., on the number of ξ-values. The following diagrams depict the performance graphs for the different strategies for $D = 5, D = 9$, and $D = 17$.



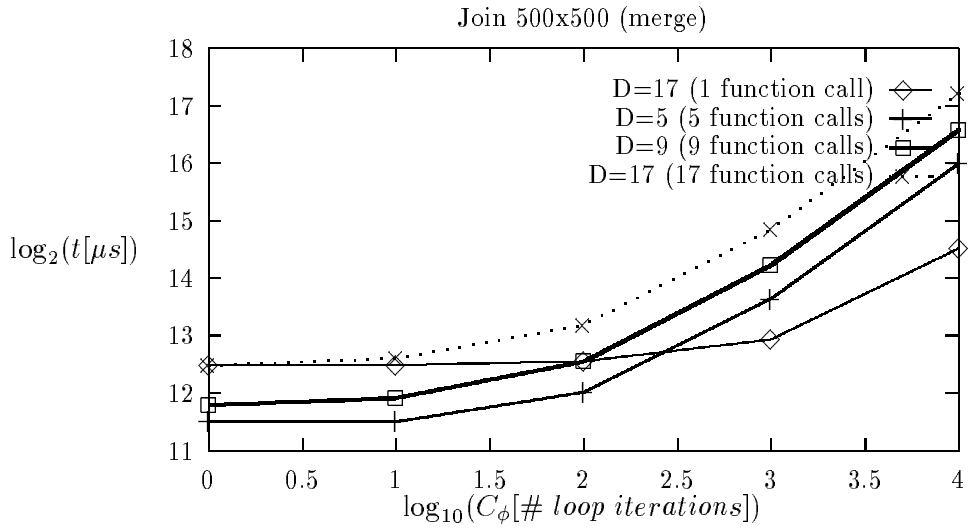




These experimental results confirm the theoretical prediction that the performance of the various strategies depends on the cardinality of the ξ-set, i.e., on D. However, the impact of different D parameters differs from method to method. In all cases the impact of D vanishes with higher values of C_Φ.

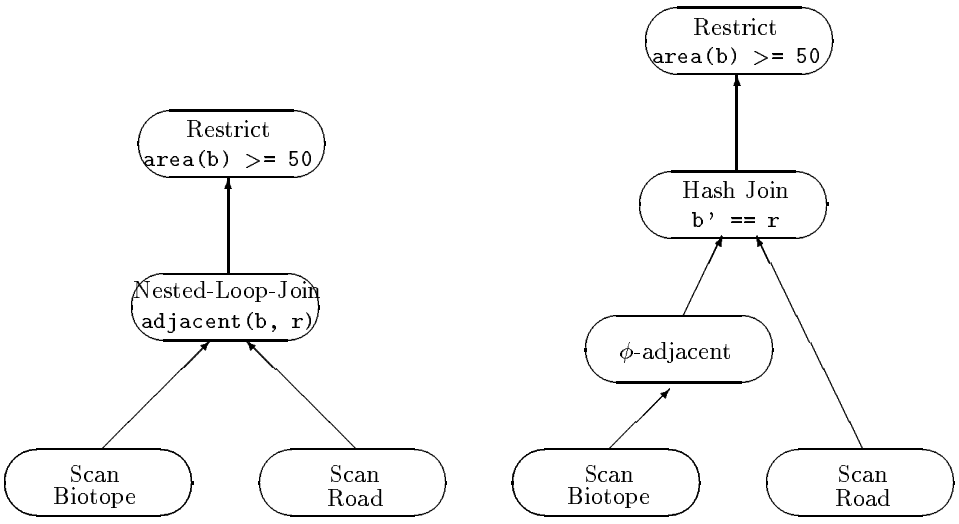
In the above diagrams, we called the ϕ-function only once per parameter. This function call returned the complete set of ξ-values. If this can not be done for a particular ϕ-predicate and one has to call the ϕ-function up to D times, one finds the linear dependency on D confirmed, as can be seen from the following diagram. The case $D = 17$ with just *one* function call has been included for comparison purposes.

In summary, the ϕ-function strategy seems to yield the greatest improvements if (i) the grade D is small, and (ii) the evaluation costs C_Φ are large.



6 A Note on Implementation Issues

The application of predicate migration [HS93] to Query 1 would probably result in the query plan given in figure 5(a), i.e., Postgres would resort to a simple nested loop. Since the join predicate is not a simple equivalence test anymore, the optimizer would not be able to propose a better strategy. With the notion of the ϕ-function, the optimizer would obtain the execution plan given in figure 5(b), which enables the query execution engine to exploit the more efficient hash join.



(a) Without ϕ-Function (b) With ϕ-Function

Figure 5: Possible plans of query 1 after predicate migration.

Several details should be noted in this context:

First, the transition from figure 5(a) to figure 5(b) is possible due to the following equivalence

$$R \bowtie_{\Phi} S = \phi(R) \bowtie_{equal} S$$

In order for the optimizer to take advantage of the concept of ϕ-functions, a set of equivalences for the Φ-predicate must be provided as well as the different ϕ-functions. This set of equivalences can be made available to the optimizer in a Revelation-like fashion [GM88], e.g., by sending a message to the Φ-function to reveal possible equivalences.

Second, as can be seen from the execution plan, the ϕ-function is a stream operator similar to the well-known sort operator. This similarity supports our claim that the concept of ϕ-functions can be incorporated rather easily into current extendible database systems as long as the interface is well-defined and known.

Finally, to assess the cost of the ϕ-function, the grade D and the cost per tuple should also be known. As should be noted from the preceding discussion, it is not anymore sufficient to carry the notion of extensibility only to the query interface. Having an extensible query interface also influences all other parts of the database including, of course, the query execution engine.

7 Conclusions

In this paper, we considered the problem of efficient join processing in the presence of user-defined functions. The key idea is to apply a two-step algorithm to process a join between two sets R and S: First, each set element is processed separately with respect to the user-defined function(s) being used. Then any particular join query containing those functions can be computed by a variation of some traditional join strategy, such as sort-merge or hash-join. We provided a detailed theoretical and practical analysis and comparison of the different strategies and found that it is possible to reduce the time consumption significantly even for simple predicates. The higher the (computational) complexity of the user-defined function(s), the more it is worthwhile to use the techniques proposed in this paper. As the performance results show, it is possible to reduce the total time consumption by up to several orders of magnitudes.

It should be mentioned, however, that this approach also requires the user to do some extra work: The ϕ-function needed to compute the ξ-values has to be provided by the user, based on the corresponding Φ-predicate. In our experience, this conversion has always been a rather simple exercise for the programmer of this predicate. Furthermore, we found most user-defined functions could easily be converted into a format fulfilling the requirements of a ϕ-function. Work on a general methodology to define a greater class of user-defined functions that can be transformed into a ϕ-function is currently in progress.

References

[AF93] Karl Aberer and Gisela Fischer. Object-oriented query processing: The impact of methods on language, architecture and optimization. Technical Report GMD no. 763, GMD-IPSI, Darmstadt, Germany, 1993.

[BKS93] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient processing of spatial joins using r-trees. In *Proc. of the 1993 ACM Int. Conf. on the Management of Data, SIGMOD Record*, volume 22, pages 237–246, June 1993.

[Bla91] José A. Blakeley. Object query module. Technical report, Texas Instrument Incorporation, 1991. DARPA Open Object-Oriented Database Preliminary Module Specification.

[Dit91] Klaus R. Dittrich. Object-oriented database systems: The notion and the issues. In *On Object-Oriented Database Systems*, pages 3–10. Springer-Verlag, 1991.

[Gae93] Volker Gaede. Z-ordering and its application in the context of query processing. Technical Report FAW-TR-93016, FAW Ulm, Germany, 1993.

[GM88] Götz Graefe and David Maier. Query optimization in object-oriented database system: A prospectus. In *Advances in Object-Oriented Database System*, pages 358–363. Springer Verlag, September 1988. Lecture Notes in Computer Science, 334.

[Gra93] Götz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[Gü93] Oliver Günther. Efficient computation of spatial joins. In *Proc. 9th Int. Conf. on Data Engineering*, 1993.

[HS93] Joseph M. Hellerstein and Michael Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 267–276, 1993.

[JG91] Zhuoan Jiao and Peter M. D. Gray. Optimisation of methods in a navigational query language. In *Proc. of the 2nd Int. Conf. on Deductive and Object-Oriented Databases*, pages 22–35, 1991.

[ME92] Priti Mishra and Margaret H. Eich. Join processing in relational databases. *ACM Computing Surveys*, 24(1):63–113, March 1992.

[OM88] Jack A. Orenstein and Frank Manola. Probe: Spatial data modeling and query processing in an image database application. *IEEE Transactions on Software Engineering*, 14:611–629, May 1988.

[SZ90] Gail M. Shaw and Stanley B. Zdonik. A query algebra for object-oriented databases. In *Proc. IEEE Conf. on Data Engineering*, pages 154–162, February 1990.

[Ull88] Jeffrey Ullman. *Principles of Database and Knowledge Base Systems*, volume 1. Computer Science Press, 1988.

[Val87] P. Valduriez. Join indices. *ACM Transaction on Database Systems*, 12(2), 1987.