Join and Semijoin Algorithms for a Multiprocessor Database Machine

PATRICK VALDURIEZ and GEORGES GARDARIN
SABRE Project, INRIA and University of Paris VI

This paper presents and analyzes algorithms for computing joins and semijoins of relations in a multiprocessor database machine. First, a model of the multiprocessor architecture is described, incorporating parameters defining I/O, CPU, and message transmission times that permit calculation of the execution times of these algorithms. Then, three join algorithms are presented and compared. It is shown that, for a given configuration, each algorithm has an application domain defined by the characteristics of the operand and result relations. Since a semijoin operator is useful for decreasing I/O and transmission times in a multiprocessor system, we present and compare two equi-semijoin algorithms and one non-equi-semijoin algorithm. The execution times of these algorithms are generally linearly proportional to the size of the operand and result relations, and inversely proportional to the number of processors. We then compare a method which consists of joining two relations to a method whereby one joins their semijoins. Finally, it is shown that the latter method, using semijoins, is generally better. The various algorithms presented are implemented in the SABRE database system; an evaluation model selects the best algorithm for performing a join according to the results presented here. A first version of the SABRE system is currently operational at INRIA.

Categories and Subject Descriptors: H.2.6 [**Database Management**]: Database Machines; H.2.3 [**Database Management**]: Languages—*data manipulation languages* (*DML*); C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures

General Terms: Algorithms, Performance

Keywords and Phrases: Relational algebra, join, semijoin

1. INTRODUCTION

Several relational database machines are currently being developed [1, 4, 14, 30]. The main objective of these projects is to answer complex queries, expressed in a nonprocedural language on a very large database, with better performance than conventional database systems [11, 28]. These machines are generally equipped with a set of processors executing requests in parallel. They are examples of a design approach which tends to distribute the processing power in close proximity to the data storage units. This relieves the main computer of the data processing functions.

Authors' address: SABRE Project, INRIA and University of Paris VI, BP 105, 78150 Le Chesnay, France.

One factor limiting the performance of relational systems is the join operation. The θ-join [13] (or simply join) of two operand relations R and S on attributes A from R, and B from S is the result relation T, obtained by concatenating each tuple in R with each tuple in S, such as, for example, $A\ \theta\ B$, where θ is one of the operators $=, <, \leq, \geq, >, \neq$. Join is an important operation generally needed to answer multirelation queries, but it can be very time consuming. A semijoin is a special case of join, in which the attributes of the result relation belong to only one operand relation.

Several uniprocessor algorithms have been presented and discussed [8, 19, 29]. In such a context, and without indexing, the obvious method of computing joins by nested loops has an execution time proportional to $n^{**}2$ for relations of cardinality n. A better method, based on sorting, can reduce this time to $a \cdot n \log n$, where a is a constant [8]. A still better method, based on hashing [2], can further reduce the time to $b \cdot n$, where b is also a constant. However, this last method allows the performance of semijoins only.

A large collection of parallel algorithms can be derived from monoprocessor algorithms for performing join. We explore here a class of algorithms based on "divide and conquer" principles; this means that data are divided into equal parts, and a similar algorithm is then applied by each processor to join the parts. A first study of this class of algorithms has been done in [9]. In this paper we extend their results to the SABRE database machine architecture, using a larger class of algorithms; we also show the value of performing semijoins before joins in a multiprocessor system.

In summary, this paper presents three implementations of parallel join algorithms—the nested loops, the sort-merge, and the hashing algorithms—in the SABRE multiprocessor system [18], and it more precisely analyzes the algorithms' performance. In Section 2, the architecture model that enables us to study the algorithms is precisely described; it is derived from the SABRE database machine project. The SABRE database machine components involved in computing joins or semijoins are a set of filtering processors associated with the disk units, a cache memory, and a set of join processors. The processors own a local memory which stores data during processing, and are linked together by an interconnection network. In Section 3, the analysis criteria are introduced, including the performance parameters needed for the evaluation of the execution times (I/O, CPU, and message transmission) of the algorithms. The basic assumption is that the operand relations are too large to fit into the cache memory or into the join processors' local memories. In Section 4, three multiprocessor join algorithms are presented: the nested loop join algorithm [15], the sort-merge-join algorithm, and the hashing join algorithm. The execution times of these algorithms are then compared. The results depend mainly on the number of processors, the size of the operand relations, and the proportion of matching tuples in a relation. It turns out that, for a given configuration, each algorithm has an application domain defined by the characteristics of the operand and result relations.

Recent results have shown the value of semijoins for optimizing query execution [5–7, 12]. The length of time for this operation is substantially less than that for a join, and is linearly proportional to the size of the operand and result relations.

The semijoin is useful in distributed relational databases [23, 26] for reducing the time for processing queries involving binary operations, by means of initially selecting relevant data and thereby reducing the size of the operand relations. We show in this paper that it is also very useful for optimizing join processing in a multiprocessor database machine. In Section 5, two equi-semijoin algorithms are presented and compared. A third non-equi-semijoin algorithm is also proposed. Their execution time is generally proportional to the size of the operand and result relations, and inversely proportional to the number of processors. In a uniprocessor environment, a semijoin can be used efficiently to replace the join of relations by the join of their semijoins [29]. Thus comparisons of two join methods, one based directly on the nested loop algorithm [9] and the other performing semijoins before the join, are developed. These comparisons indicate the general advantage of performing joins by using semijoins.

2. ARCHITECTURAL MODEL

The environment in which join and semijoin algorithms are studied is the SABRE database machine [18], composed of a set of processors linked through an interconnection network. The processors exchange commands directly and exchange data via a cache memory. We now give a simplified model which describes the main architectural components of SABRE, focusing on those components useful in implementing joins and semijoins.

The first layer of the architecture is arranged as a set of data filtering processors [3, 16, 27]. These processors, called filters, perform selections (restriction and projection) on data coming from disks. It has been shown in [10] that, if parallel readout disks are employed, the usage of processor-per-disk together with an efficient placement strategy gives the best results, since channel contention is thereby avoided. Furthermore, this usage provides cheap filtering power. Thus, in SABRE, we implement a filter-per-disk device. Filters are very simple processors, close to the general form of automata. They permit realization of asynchronous processing "on the fly"— in other words, filters perform a selection at disk rotation speed on a secondary memory page, which generally corresponds to a track. A selection expression is composed of a set of references that specify the projection attributes and the selection condition. A selection condition is a logical combination of predicates of the form ($Ai\ op\ C$), expressed in normal conjunctive form, where Ai is an attribute of an operand relation, op an operator chosen among $<$, $\leq$, $=$, $\neq$, $\geq$, $>$, and C, a value. The number of predicates which can be taken into account by a filter depends on characteristics of the automata only, and not on the data flow or on the tested expressions. In [16] evaluations show that, for an average memory size of 64K bits, a filter can accept up to 200 predicates.

The second layer of the architecture is a cache memory. The filters are connected to the cache memory through a special bus [17]. We assume that this bus does not provide any contention. Selected data are moved via the bus to the cache memory, divided in pages. In SABRE the basic processing and moving unit is the page. The cache manager should anticipate page reading and use a replacement algorithm to swap the useless pages to the disk. A filter can be

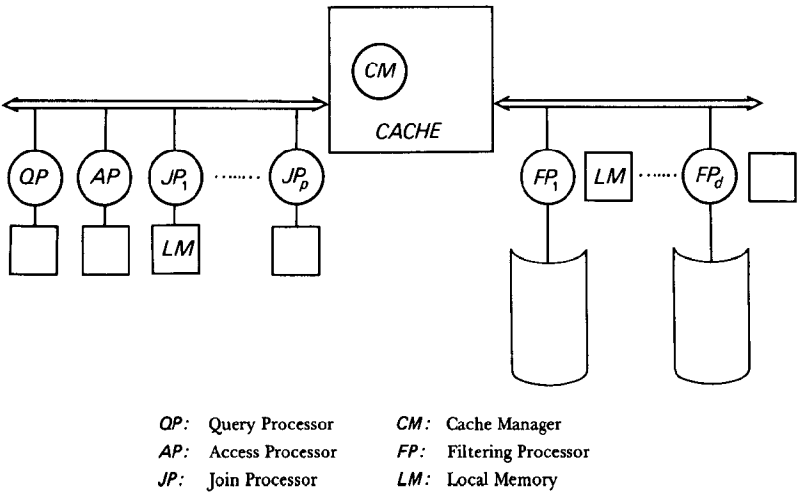

Fig. 1. Simplified architecture model.

associated to the cache memory in order to carry out the necessary selection among pages already in the cache.

The third layer of the architecture is a set of join processors devoted to computing joins of relations. Join processors send reading and writing requests to the cache in order to get pages of relations and to write back result pages. Pages are stored in the local memories of the processors during processing. The join processors are connected with the cache memory by a specific interconnection network. The requirements for this network are the ability to carry out parallel transfers between the cache and any join processor, and the capability to broadcast a page from the cache to several join processors. This kind of interconnection network is feasible [14], and different alternatives have been studied. The current developments of SABRE are based on a French multimicroprocessor called the SM90 [17]. Although this machine has a very fast common bus (eight megabytes per second), we try to be independent of the interconnection network, as it would affect the performance of the whole system. For our purposes, we assume an ideal interconnection network, which would provide the two needed services and introduce no contention.

Finally, the complete architecture of SABRE includes two more processor types (i.e., the access path management processor and the query decomposition processor [18]). These processors are connected to the join processor bus, and share with the join processors the task of executing external requests. The architectural components of SABRE are depicted in Figure 1, where, for simplicity, the interconnetion networks are reduced to buses. Thus, SABRE allows a high degree of intra- and interrequest parallelism by means of a set of specialized processors. In the following, we are going to more precisely focus on intrarequest parallelism for performing joins.

3. ANALYSIS CRITERIA

In the following, we assume that a semijoin or join operation is performed on relations R and S, producing the result relation T. We recall that the semijoin of R by S is the set of tuples of R participating in the join of R and S. The following notation is needed to evaluate the proposed algorithms:

- m, n: number of pages in R and S, respectively;
- $b + 1$: size of local memory (in number of pages) of each join processor (JP);
- p: number of join processors (JP) assigned to the operation;
- c: cache memory size allocated to the join operation (in number of pages);
- d: number of filters assigned to the operation;
- t: number of tuples in a page, assumed to be the same for both R and S;
- JS: join selectivity factor, defined by size (R join S)/($m \cdot n$);
- SR: semijoin of (R by S) selectivity factor which is defined by size(semijoin (R by S))/m;
- SS: semijoin of (S by R) selectivity factor which is defined by size(semijoin (S by R))/n;
- j: average number of distinct join attribute values in a page. Thus, mj and nj are the total numbers of distinct join attribute values in R and S, respectively;
- Nc: number of predicates accepted by a filter;
- Ct: occupation factor of join attributes in a tuple, defined by the size of the join attribute divided by the size of the tuple. Thus, the number of pages of the relation obtained after projecting relation S on join attributes without duplicate elimination is defined by $na = n \cdot Ct$.

The presented algorithms will be compared according to their execution times. The basic performance parameters necessary for the evaluation of the execution times are those detailed above. These parameters allow one to measure the I/O time, the CPU time, and the interprocessor communication time when executing a multiprocessor algorithm. The identified parameters depend on hardware capabilities, and will be fixed for purposes of comparison only.

3.1 I/O Time

In a multiprocessor architecture, two kinds of transfers are considered as I/O operations: page transfers from a mass storage unit and transfers from the cache memory to a processor's local memory. The page transfer time from disk to cache is denoted Tdc, and includes the selection time due to filtering on the fly. The page transfer time from cache to processor is denoted Tcp. The average time of a read by a join processor is now defined by introducing the probability that the requested page is already in the cache memory [9]. The cache manager should maximize this probability by anticipating page reading, which is made easier by the fact that a referenced relation will be read in its entirety. This probability is equivalent to the cache hit ratio, denoted by F. Thus the average time of a read, defined as CR, is

$$\mathrm{CR} = F \cdot T\mathrm{cp} + (1 - F) \cdot (T\mathrm{cp} + T\mathrm{dc}).$$

The average time of a write by a join processor can be evaluated in the same fashion by introducing F′—the probability that there is an available page frame in the cache during the write operation. This probability is supposed to be relatively low and constant. The cache manager uses an algorithm different than LRU, since the main difference with a classical cache memory is that the entire operation is known in advance. Therefore, the cache manager can swap out pages in order to keep F' constant. Thus the average time of a write, defined by CW, is

$$\mathrm{CW} = F' \cdot T\mathrm{cp} + (1 - F') \cdot (T\mathrm{cp} + T\mathrm{dc}).$$

3.2 Communication Time

Page moves are regarded as I/O operations. Therefore, the only communication time which needs be taken into account is from page request and reply messages. This message time, denoted as M, is then added to the I/O time. Thus CR is replaced by CR + M(request) + M(reply), and CW is replaced by CW + M(request) + M(reply). Control messages, such as those used for processor allocation, are few in number and small in size, relative to the page request and reply messages. Therefore they will be ignored.

3.3 CPU Time

Two basic operation times are defined. 0 denotes the time of a simple operation, such as comparing two attributes or computing a hashing function. The time for moving a tuple in a page of local memory is denoted by I. With these two basic operation times, we can compute all other times as follows.

Join time. The pages to be joined may or may not be sorted. If they are unsorted, two methods for joining are possible. The first consists in internally sorting the pages, followed by a merge-type join operation. The second method, which is simpler, compares each tuple in a page with each tuple in another page; it is better than the first method, since it only requires comparison operations, which are cheaper than the tuple movement that internal sorting requires. Thus, the time for joining two unsorted pages, denoted as CJ0, is CJ0 = $0 \cdot t$**2. The time for joining two sorted pages by a merge-type operation is defined as CJ1 = $2t \cdot 0$.

One page sort time. The average time for internally sorting a page of t tuples requires $t \cdot \log_2 t$ comparisons and move operations [22]. It is defined as

$$\mathrm{Cs} = (0 + I)t \cdot \log_2 t.$$

q page merge time. The merging of q sorted pages needs $q(q-1)t$ comparison operations and $q \cdot t$ move operations. By adding the q page reading and writing times, the q page merge time is defined by

$$\mathrm{CM}q = (q(q-1)t \cdot 0) + (q \cdot t \cdot I) + (q(\mathrm{CR} + \mathrm{CW})).$$

4. JOIN ALGORITHMS

4.1 The Nested Loop Join Algorithm

This algorithm is a parallel version of the most inefficient uniprocessor join algorithm, since it composes the Cartesian product of the relations. This simple algorithm is particularly well suited to parallel execution. The method is to join each page of one relation with the entire other relation. The algorithm has been described and evaluated in [9] for processors having three pages of local memory, where two buffers are used to input pages and one buffer to output pages. The method is now generalized to support execution with more than three pages of local memory. The execution of this algorithm by p processors, each having $(b + 1)$ pages of local memory proceeds as follows. The smaller relation is chosen as the external one and is sequentially distributed among p join processors in blocks of $(b - 1)$ pages. Next, the second (internal) relation is broadcast page by page to the p processors. Then, each processor joins each $(b - 1)$ page block of the external relation with the entire internal relation. For each page of the internal relation, each processor computes an internal join with $(b - 1)$ pages of the external relation, using a result buffer of one page. If the external relation does not fit into the processors' local memories, then the same process must be repeated for the remaining pages of the external relation.

We should point out that, since each tuple of one relation will be compared to each tuple of the other one, this algorithm also supports non-equijoin. Finally, the execution time of the nested loop join algorithm is derived in Appendix A. The final result is

$$\text{TIME(NL)} = m \cdot \text{CR}/p + m \cdot n(\text{CR}/(p(b-1)) + (\text{CJ0} + \text{JS}(t \cdot I + \text{CW}))/p).$$

The formula shows that this time is proportional to $m/p + a \cdot mn/p$. Therefore, the smaller relation must be chosen as external in order to decrease the first term. This equation also shows that the degree of parallelism is constant (i.e., the amount of work each processor performs is constant) during the entire execution of the algorithm. It should be noted, however, that the last pass of the algorithm can use a fewer number of processors if the ratio $m/(p(b - 1))$ does not give an integer result.

4.2 The Sort-Merge-Join Algorithm

This algorithm employs a parallel sort of the operand relations on join attributes, followed by a uniprocessor merge-type operation of the two sorted relations to complete the join. For simplicity, we consider the equijoin. The sorting algorithm is a parallel version of the uniprocessor ascending (or merge sort) algorithm (described in [22], p. 247). In [9] a join algorithm is presented using a parallel binary merge sort, where the processors are organized in the form of binary trees [25]. Performance analysis of this join algorithm shows that it is generally less

efficient than the multiprocessor nested loop join algorithm, if the number of processors is high. This is mainly because, after a certain stage, the degree of parallelism is divided by two at each merge pass, so that, at the last pass, one processor merges the entire relation. We propose a parallel b-way merge sorting, which is more efficient in a multiprocessor environment. Moreover, if the number of processors p is less than or equal to b, the last stage only needs one pass.

We briefly review the b-way merge sort algorithm. Let us suppose that there are n elements to be sorted. A run is defined as an ordered sequence of elements, thus the set to be sorted contains n runs of one element. The method consists of iteratively merging b runs of K elements into a sorted run of $K \cdot b$ elements, starting with $K = 1$. For pass i, each set of b runs of $b^{**}(i-1)$ elements is merged into a sorted run of $b^{**}i$ elements. Starting from $i = 1$, the number of passes necessary to sort n elements is $\log_b n$ [22].

We now describe the application of this method in a multiprocessor database machine. Let us suppose we have to sort a relation of n pages that is too large to fit into the cache memory.

Recall that each of the p processors has a local memory of $b + 1$ pages, where b pages are used as input pages and 1 is used as an output page. At each pass, each processor merges a set of b runs of $b^{**}(i-1)$ pages into a sorted run of $b^{**}i$ pages. The merge of b runs is done by successively reading one necessary page of each run into b input buffers and then moving ordered tuples into the output buffer, writing in the cache memory when full. When read at the first pass, pages are internally sorted. To clarify the analysis, we assume that modulo $(n, p) = 0$ (i.e., that p processors divide the relation into equal parts and share exactly the same amount of work). At each merge pass the number of runs is divided by b, while the size of each run is multiplied by b, and the whole relation is read and written. For the first pass, let N be the number of runs to be merged; if N is greater than $p \cdot b$, one step is necessary for each of the p processors to merge b runs, and this step is repeated $N/(p \cdot b)$ times until all the runs have been read. When N is equal to $p \cdot b$, only one step is necessary, and each processor merges exactly b runs of $n/(p \cdot b)$ pages. This pass is called the optimal stage [9]. The optimal stage then generates p runs of n/p pages and, if p equals b, one processor can merge them in a single pass. But, in certain configurations, p can be very much greater than b, in which case the solution is to arrange the processors as a tree of order b during the last stage, called the postoptimal stage. The number of necessary processors is divided by b at each pass. At the last pass, one processor merges the entire relation. The number of passes for the postoptimal stage is $\log_b p$. This stage degrades the degree of parallelism. However, the result relation is sorted on the join attributes, which can be very useful if such a sort is required in the query.

Joining two sorted relations is done by a uniprocessor merge-type operation, where each relation is read one page at a time. The adaptation of this algorithm to the non-equijoin modifies the merge-type operation into a more complex and time consuming one.

In Appendix B, we evaluate the execution time for sorting a relation for n pages. We then derive the execution time of the sort-merge-join algorithm, which

is

$$\begin{aligned}\text{TIME(SM)} = {} & n/p[\text{Cs} + \text{CM}b\cdot\log_b(n/p)/b + \text{CM}b\cdot(1 - b^{**}\log_b p)/(1 - b)] \\ & + m/p[\text{Cs} + \text{CM}b\cdot\log_b(m/p)/b + \text{CM}b\cdot(1 - b^{**}\log_b p)/(1 - b)] \\ & + (m + n)\text{CR} + \max(m, n)\cdot\text{CJ1} + m\cdot n\cdot\text{JS}\cdot\text{CW}.\end{aligned}$$

This formula is rather complex, and will be interpreted later with the given parameters.

4.3 The Hashing Join Algorithm

The proposed method uses hashing techniques and Boolean arrays. The use of Boolean arrays for implementing the semijoin operation has been described by Babb [2]. The idea is to hash the join attribute and to then use the result as an address into the Boolean array. The presence of a marked bit in the array means that matching tuples exist. The value of the Boolean arrays is to eliminate most of the data not needed in the result. In order to support join as well as semijoin operations, the method is improved and adapted to a multiprocessor context. For simplicity, we describe the equijoin. The method proceeds in two stages, prior to which a Boolean array B is initialized in the cache memory. In the first stage, the smaller relation is read into the cache memory and hashed on the join attribute by the cache processor so that each tuple is written in a bucket of a hashed file. The hashed file is composed of buckets having a variable number of linked pages. For each bucket, a page frame is maintained in cache memory. This avoids the need to manage an overflow area. Simultaneously, for each join attribute value v, $B(h(v))$ is marked (set to 1), where h is a hashing function applied to the join attribute. The first stage is completed when the entire relation has been hashed. In the second stage, the Boolean array is broadcast to p processors, and the larger relation is sequentially distributed among p processors. Each processor uses two buffers as input pages, one buffer as the output page and one buffer to store the Boolean array. Thus, each processor receives one page of the larger relation and performs the following processing. For each tuple of the page such that the join attribute value v' satisfies $B(h(v')) = 1$, one bucket of the hashed file is accessed by specifying the key v' to find the matching tuple(s). Since a bucket of the hashed file may contain more than one page, the bucket is read page by page. The tuples of each page are then compared with v' to complete the join.

This method makes extensive use of hashing. In order for it to be applicable, the number of distinct join attribute values in the hashed relation must be significantly greater than the number of buckets. At the first stage, the choice of hashing the smaller relation is made to minimize the creation and storage times of the hashed file.

The classical problem with hashing is collisions. If $v1$ and $v2$ are different join attribute values, we can have $h(v1) = h(v2)$. Since the Boolean array is accessed by hashing, collisions can lead to useless access to the hashed file during the

second stage. In order to reduce collisions, several hashing functions $h1, h2, \ldots, hq$ can be used, each associated with a Boolean array $B1, B2, \ldots, Bq$. Then, for each value v, all of the corresponding bits in each Bi must be set (i.e., $B1(h1(v)) = 1, B2(h2(v)) = 1, \ldots Bq(hq(v)) = 1$). In [2] it is shown that increasing q causes the probability of collisions to approach 0.

If the hashing function for the hashed file has the property of maintaining order (i.e., given two attribute values $v1$ and $v2$, if $v1 < v2$, then $h(v1) < h(v2)$, and the algorithm can then support non-equijoin).

In Appendix C, we derive the execution time of the algorithm, which is

$$\begin{aligned}\text{TIME(H)} = {} & m[(1 - F)T\text{dc} + t(0 + I)] + (m - C)(1 - F')T\text{dc} \\ & + (n/p)[\text{CR} + t \cdot 0 + t \cdot \text{SS} \cdot (m/c) \cdot \text{CR} + m \cdot \text{JS} \cdot \text{CW}].\end{aligned}$$

It is of the form $a1 \cdot m + a2 \cdot n/p + a3 \cdot mn/p$, where $a3$ is proportional to the join and semijoin selectivities. It already appears therefore that the algorithm will be almost linear for join with a very small selectivity; thus, it is a very efficient algorithm for such a case.

4.4 Comparisons

For purposes of comparison, we fix the values of the parameters discussed in Section 3. The chosen values according to our current implementation are the following:

- 0: the time to compare two attributes is equal to 10 microseconds;
- I: the time to move a tuple in a page is equal to 250 microseconds, based on a page size of 4K bytes and a tuple length of 40 bytes;
- t: the number of tuples in a page is equal to 100 microseconds;
- F, F': cache hit ratios, $F = 0.8$ and $F' = 0.3$;
- *Tdc:* the time of a disk-cache page transfer is equal to 30 milliseconds;
- *Tcp:* the time of a cache-processor page transfer is equal to 4 milliseconds, based on an average bus bandwidth of 1 megabyte per second;
- M: the time to process a message which includes sending, transfer, and receiving times and is assumed to be 10 milliseconds;
- *Ct:* occupation factor of a join attribute is equal to 0.2, so semijoin attribute length is 8 bytes.

Using these fixed values and the previous equations, the three join algorithms are compared. Other comparisons have been done with different values of F and $F,'$ showing that their influence is weak (since the differences between the execution times remain approximately constant). However, decreasing F' increases somewhat the execution time of the sorting join algorithm relative to the other algorithms, since the final merges are done by a unique processor. For simplicity, we denote the nested loop join algorithm as the NL algorithm, the sort-merge-join algorithm as the SM algorithm, and the hashing join algorithm as the H algorithm.

4.4.1 *Execution Times versus Relation Sizes.* Figure 2 illustrates the behavior of the algorithms versus relation sizes. The sizes of the two relations are assumed

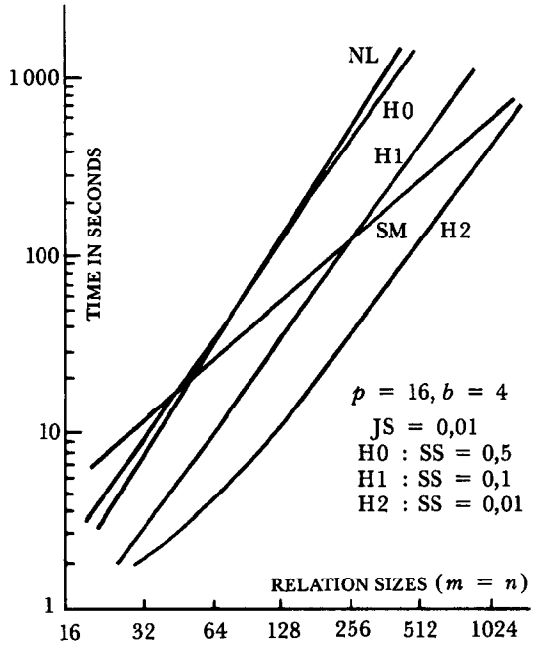

Fig. 2. Execution times of the algorithms versus relation sizes.

to be the same ($m = n$). The configuration comprises 16 processors, each having a local memory of 5 pages with a cache memory size (allocated to the join) of 16 pages. Three curves describe the performances of the H algorithm for three different semijoin selectivity factors (0.5, 0.1, 0.01). The join selectivity factor is assumed to be 0.01. It appears that the SM algorithm is better than both the NL algorithm and the H algorithm (with SS $\geq$ 0.1), when the operand relations are large. The behavior of the H algorithm depends essentially on SS, the semijoin selectivity factor, which yields the number of matching tuples from the distributed relation with the hashed file. When SS is small (SS $<$ 0.01), the number of accesses to the hashed file is low, and the H algorithm is superior. For the given configuration, when SS is less than 0.5, the H algorithm always performs better than the NL algorithm. Beyond SS $\simeq$ 0.5, the H algorithm does not perform well (curve H0). The size of the cache memory allocated to the join (denoted by C) significantly influences the execution time of the H algorithm, because, if C is high enough, the hashed file can fit into the cache without swapping out to the disk.

The previous comparisons show that, for the given configuration, each algorithm has a domain of application where it performs better than the others. Figure 3 illustrates the domains of the NL algorithm in comparison with the SM algorithm, for various sizes of relations. Similarly, Figure 4 depicts the domains of the SM algorithm in comparison with the H algorithm for SS = 0.1, which is a realistic coefficient.

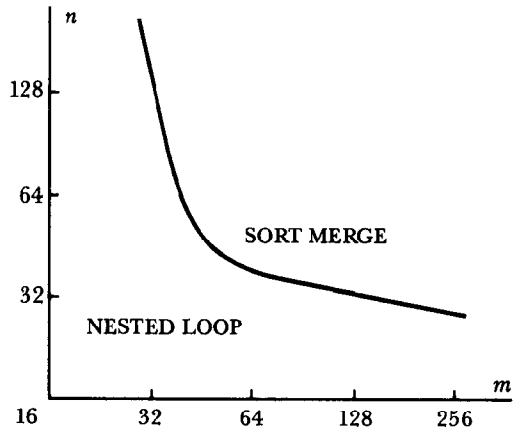

Fig. 3. Limit of the NL algorithm in comparison with the SM algorithm. ($p = 16$, $b = 4$, JS = 0.01)

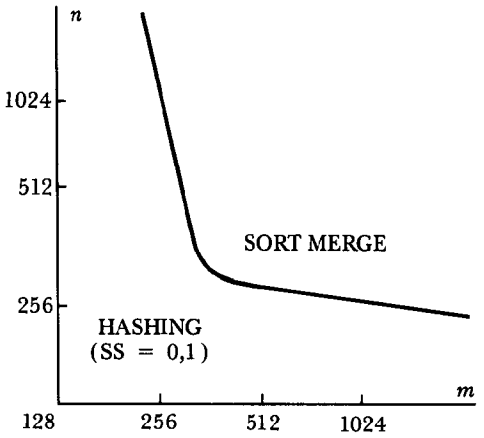

Fig. 4. Limit of the H algorithm in comparison with the SM algorithm.

In conclusion, one of the important results of this first comparison is that no algorithm is predominant. Therefore, a well-designed database machine should implement all the algorithms and select the best one to perform a join, according to the estimated time in utilizing the previous formulas. That is the way the SABRE system performs joins.

4.4.2 *Comparisons with Respect to the Number of Processors.* The configuration is essentially characterized by the number of join processors and the size of the cache memory. Figures 5 and 6 show the behavior of the algorithms for various numbers of processors for small relations (Figure 5) and larger relations (Figure 6). The complexity of the NL algorithm is $1/p$, while that of the SM algorithm is $1/\log_b p$. Thus, when the number of processors increases, the NL

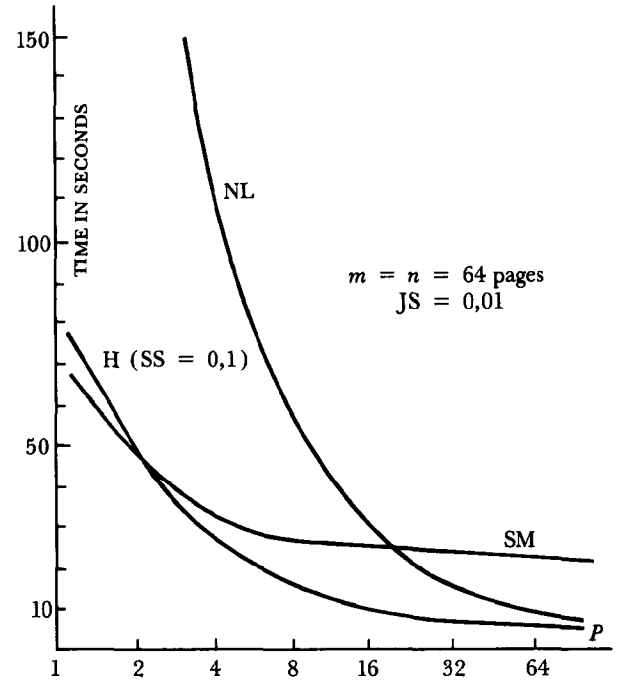

Fig. 5. Execution times of the algorithms versus the number of processors.

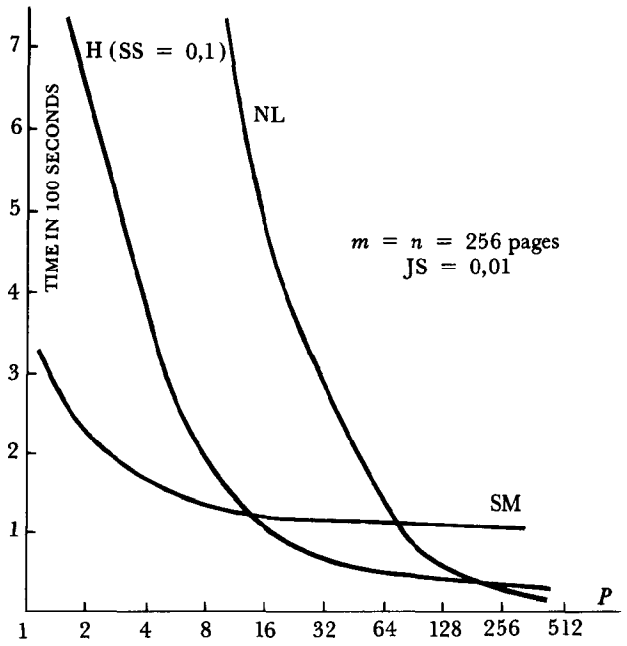

Fig. 6. Execution times of the algorithms versus the number of processors (p).

algorithm performs better than the SM algorithm. The curve of the H algorithm for SS = 0.5 would be approximately the same as that of the NL algorithm. With a favorable semijoin selectivity factor (0.1), the H algorithm gives good results when the number of processors is high. This is because the time of the second stage of the H algorithm is of complexity $1/p$. Generally, the semijoin selectivity factor will be low, and, in that case, the H algorithm seems very attractive. It is notable that, for the SM algorithm, after a certain number of processors, duplicating the number of processors causes very little decrease in the execution time. This is owing to the increasing number of passes of the postoptimal stage, proportional to the number of processors. In fact, the SM algorithm is better for configurations in which $p = b$, when only one postoptimal pass is needed. When p becomes greater than b, the postoptimal stage degrades the degree of parallelism.

5. SEMIJOIN ALGORITHMS

If the comparison operator θ, in the join definition of Section 1, is equal, and if the result attributes belong to a single relation, the operation is called an equi-semijoin. In this section we will mainly discuss and evaluate two equi-semijoin algorithms, one using bit arrays and the other using the selection operation performed by the filtering processors. Finally, we will consider non-equi-semijoins, where the comparison operator θ is not equal.

5.1 Equi-Semijoin with Bit Arrays

The proposed method is an extension of the uniprocessor method for performing semijoins, as described in [2]. The basic version uses a bit array to aid the operation. In our parallel version this array is replicated in the join processors' local memories. The operation in a multiprocessor environment proceeds in two stages. First, semijoin source attributes are retrieved in parallel by d filtering processors, moved to the cache memory, and finally compacted in pages. These pages are then distributed among p join processors. Thus each processor reads na/p pages of the source relation S after projection—called relation S'— and places source attribute values in a local memory resident data structure W, in ascending order, if the value is not already in W. Simultaneously, one bit in the bit array M is marked for each source attribute value v. The bit address is calculated by a hashing function h, where $M(h(v))$ is set to 1. At the end of this first stage, the bit array in each processor represents a part of the attribute values of the source relation, while the union of all the bit arrays represents all the distinct attribute values. The logical union of the p arrays is then computed by one processor, which broadcasts the result to the $p - 1$ other processors. The intermediate lists (W) of each processor are merged to form a single list containing all the distinct source attribute values. This step is performed by one processor in one b-way merge pass if $p \leq b$, or in $\log_b p$ passes if $p > b$, where the number of processors is divided by b at each pass. The resultant merged list does not contain any duplicates and is broadcast to the remaining $p - 1$ processors. The main assumption of this method is that a processor's local memory can contain both the bit array and the list W. This assumption holds in most cases. If,

however, the list does not fit into the local memory, the problem is treated in the same way as the join problem analyzed in Section 4.

The second stage of the algorithm is to distribute the target relation R (after selection by filters) among p processors. Each processor gets a page of R, and each tuple, whose join attribute value v' satisfies $M(h(v')) = 1$, is projected into the result relation if v' is in W. The latter check is necessary because of possible collisions during hashing. (Note that more than one bit array can be used to decrease the probability of collisions.)

The total time of the semijoin algorithm using bit arrays is derived in Appendix D. It includes the I/O time, which is

$$\text{I/O(SJH)} = (n/d)\cdot T\text{dc} + (na/p)\cdot\text{CR} + [p + 1 + (\log_b p)\cdot(b/2)\cdot T\text{ cp} + (m/p)\cdot(\text{CR} + \text{SR}\cdot\text{CW}).$$

and the CPU time, which is

$$\text{CPU(SJH)} = (na/p)\cdot(t/Ct)\cdot(0 + Iw) + Cm + p\cdot 0 + (m/p)\cdot[0\cdot(t + t\cdot\text{SR}\cdot\log_2 nj) + \text{SR}\cdot I].$$

5.2 Equi-Semijoin Using Selection

This method depends entirely on the selection devices offered by the filtering processors [16] to compute the equi-semijoin. The operation proceeds in two stages. The first stage is similar to that of the previous algorithm; bit arrays are, however, not used. The second stage realizes the semijoin by restricting the target relation to the tuples whose semijoin attribute value is in the list W. Only one processor is needed to initiate the operation. From the merged list, W, this processor constructs selection conditions in the target relation by constructing a restriction predicate. This predicate is a disjunction of clauses of the form $(A = Bi)$, where A is the semijoin attribute of R, and Bi are the distinct attribute values in W. If the number of Bis is greater than Nc, which is the number of predicates accepted by a filter, several selection operations are involved. The number of operations is then $\text{Sup}(nj/Nc)$. Since every selection operation requires reading R in its entirety, the performance of this algorithm depends on the filter processing capabilities. One should keep in mind that a simple filter could accept a high number of predicates.

The total time of the semijoin algorithm using selection is derived in Appendix E. It includes the I/O time, which is

$$\text{I/O(SJS)} = (n/d)\cdot T\text{dc} + (na/p)\cdot\text{CR} + \log_b p\cdot(b/2)\cdot T\text{cp} + (nj/Nc)\cdot(m/d)\cdot T\text{dc}.$$

and the CPU time, which is

$$\text{CPU(SJS)} = (na/p)\cdot(t/Ct)\cdot Iw + Cm + nj\cdot 0.$$

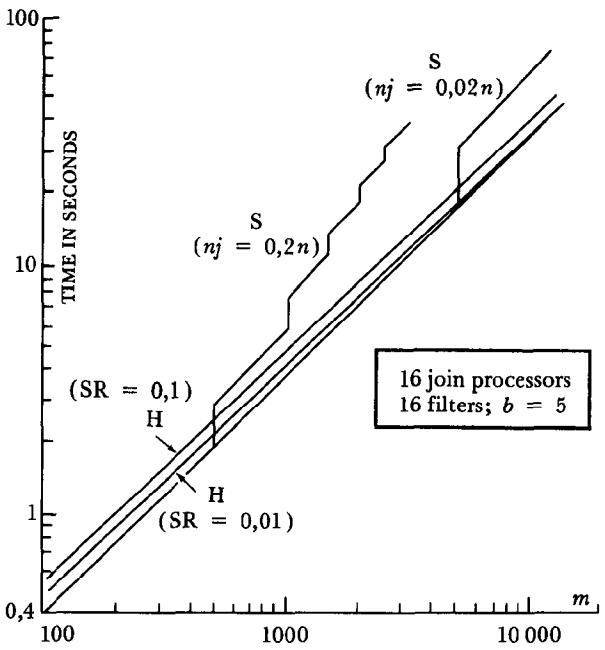

Fig. 7. Execution times versus relation sizes.

5.3 Comparisons

This section presents performance comparisons of the two equi-semijoin algorithms, using the time formulas developed in the previous sections. Results are shown in Figures 7 and 8. Assumptions concerning the processor capabilities, as parametrized by the times defined in Section 3, are the same as those of Section 4.

Figure 7 illustrates the behavior of the algorithms with varying relation sizes. The sizes of both the source and target relations are assumed to be the same ($m = n$). The configuration chosen to represent a multiprocessor architecture is composed of 16 filters and 16 join processors, each owning five pages of local memory. A filter is assumed to accept up to 100 predicates. The stage which consists of reading the projected relation S' and merging the lists W is the same for both algorithms, therefore parameters Ct, Cm, and Iw do not influence the performance. Figure 7 consists of four curves. The two curves representing the time for the algorithm using bit arrays (SJH) vary linearly according to the size of relations (n), and depend on two different semijoin selectivity factors. The selectivity factor slightly affects the write time of the result. The two time curves of the algorithm using selection (SJS) depend on two distinct nj values, and show that this algorithm is heavily influenced by the ratio Sup (nj/Nc), which is the number of needed selection operations. This can be seen by the stair shape of the curves. This algorithm is better when there are a few distinct attribute values. More generally, the algorithm by selection is better than the algorithm using bit

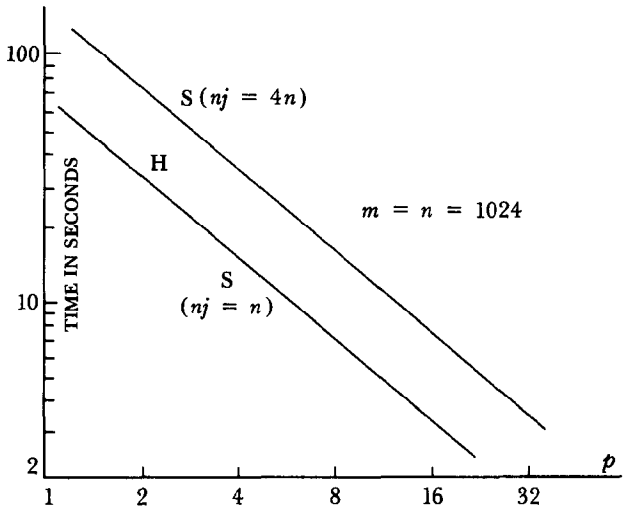

Fig. 8. Execution times versus the number of processors.

arrays when the approximate expression

$$(m/p)\cdot(\mathrm{CW} + \mathrm{SR}\cdot\mathrm{CW}) - \mathrm{Sup}(nj/Nc)\cdot(m/d)\cdot T\mathrm{dc}$$

is strictly positive.

The presence of p join processors cuts the time of the bit arrays algorithm by a factor of p, just as the presence of d filters cuts time off the selection algorithm.

Figure 8 represents execution times as a function of the number of processors, assuming that $p = d$. Performances of these algorithms are proportional to the number of processors, and the difference between the two curves is also due to the ratio (nj/Nc). Two curves depict the performances of the algorithm by selection $(nj = n, nj = 4n)$. The curve for the selection algorithm for $nj = n$ is the same as the curve for the algorithm using bit arrays. For greater values of nj, the algorithm by selection is worse. The performance of the algorithm by selection depends on the number of filters and their capabilities.

5.4 Non-Equi-Semijoin Algorithms

The non-equi-semijoin represents the case when the comparison operator represents the forms $<$, $\leq$, $\geq$, $>$. In this section we propose a simple algorithm to perform non-equi-semijoin in a multiprocessor database machine. The method is based upon the following observation: the non-equi-semijoin consists of selecting those tuples r of the target relation whose semijoin attributes satisfy "$r\cdot A$ op X," where X is either the minimum attribute value in the source relation, if op is $>$ or $\geq$, or the maximum value, if op is $<$ or $\leq$. The search for a maximum or minimum value in a data set is an easily performed function supported by the filtering processors. Each read attribute value is compared by the filter to a maximum or minimum current value in its local memory, and replaces the

current value if the comparison holds. Then, d filters get partial results which will be integrated into one result by a processor, which can be the filter controller. The non-equi-semijoin is finally performed by restricting the tuples in the target through d filters which require tuples to satisfy A op X. Furthermore, this algorithm has the advantage of not using the cache.

The CPU time of this algorithm is negligible, since it only consists of two function calls. The I/O time is the reading time of the source relation (of size n) necessary to calculate the minimum or maximum in parallel by d filters, plus the reading time of the target relation (of size m) to apply the selection. The result relation is written directly in cache, thus the write time is included in the selection time. The execution time of this algorithm is

$$\text{TIME(ISJ)} = ((n/d) + (m/d)) \cdot T\text{dc}.$$

It is assumed that the cache manager always keeps track of available pages during the operation, in order not to slow down the filters by waiting.

6. JOIN USING SEMIJOIN

The previously proposed semijoin algorithms are computed in a time linearly proportional to the size of the operands. The main advantage of the semijoin operator is to reduce the operand relation to just those tuples that participate in the join. Therefore, it may sometimes be advantageous to replace the join of two relations by the join of two semijoins. In order to compare the two alternatives (join or semijoin), we have chosen the multiprocessor nested loop join algorithm presented in Section 4.1, since it is simple and generally satisfactory, with a high degree of real parallelism.

The nested loop join of the initial relations is denoted by NL and is then compared to the nested loop of the semijoins of each relation by the other, denoted NLSJ. The semijoin algorithm using bit arrays is chosen for comparisons, since it is more general than the semijoin algorithm using selection, and more frequently used than the non-equi-semijoin algorithm. The condition that has to be satisfied in order to apply this method is that the entire list of distinct join attribute values fit into local memory. When this condition cannot be satisfied, the semijoin is solved like a join. Therefore, it is assumed that the condition is true in the remainder of this section.

Comparisons of the two methods, NL and NLSJ, are illustrated in Figures 9, 10, 11, and 12. The basic configuration is the same as in Section 4.4. Figure 9 presents the performance of the two methods according to the ratio JS′ = size (R join S)/($m' \cdot n'$), where m' and n' are the sizes of the semijoins of R by S and of S by R, respectively. For each method, three join selectivity factors are considered (JS = 0.16, 0.016, 0.5), while the relation sizes are kept large ($m = n$ = 1024 pages). The parameter JS′ significantly influences the algorithm using semijoins. As JS′ approaches 0, the size of the join result approaches the size of the biggest semijoin result; as JS′ approaches 1, the size of the join result approaches the size of the Cartesian product of the semijoin results. In the latter case, the semijoin reduces the size of the initial relations significantly. The method using semijoin is generally better than the nested loop method, and

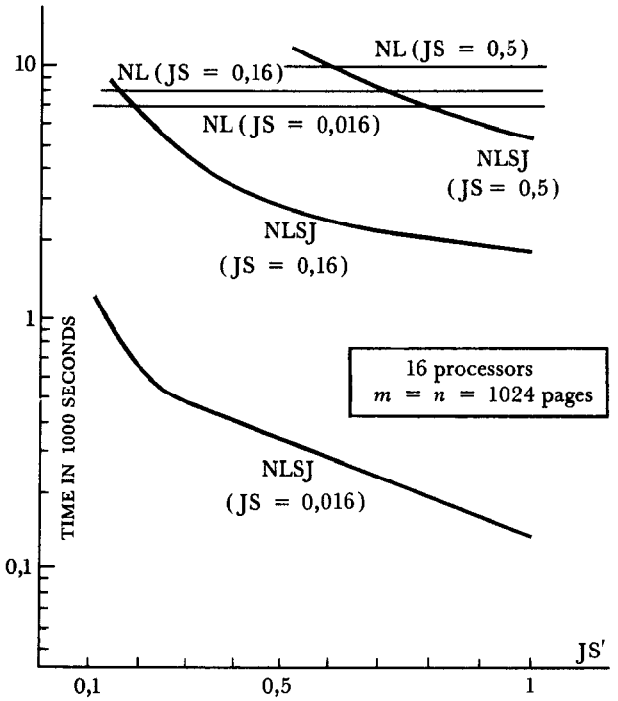

Fig. 9. Comparisons of the time of the nested loop join method (NL) and the join method using semijoin (NLSJ) versus JS'. JS' = size (R join S)/[size (R semijoin S) · size (S semijoin R)].

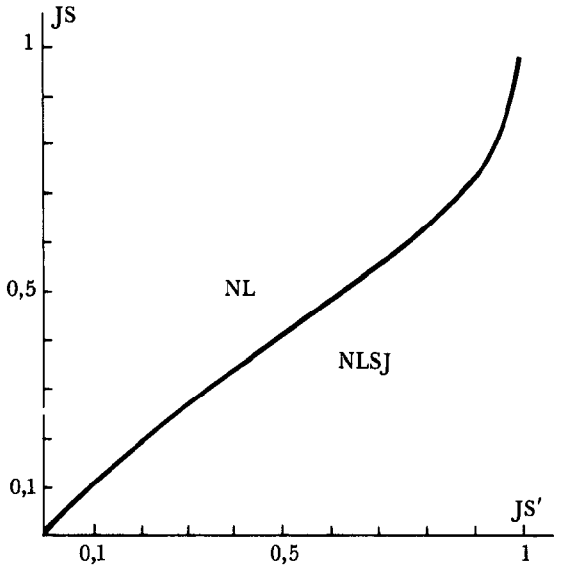

Fig. 10. Limit of the methods for varying JS and JS'. ($p = 16$, $b = 4$, $m = n = 1024$)

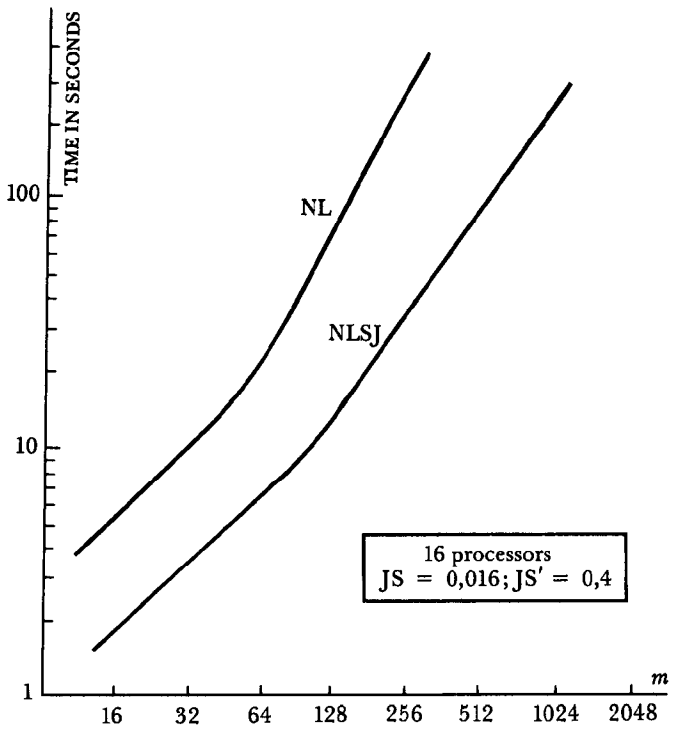

Fig. 11. Comparisons of the algorithms' times versus relation sizes.

becomes quite superior when the join selectivity factor decreases. Figure 10 illustrates the domains of the two methods for various JS and JS′ and shows that the NL method is only competitive when the ratio (JS/JS′) approaches 1. In that case, performing semijoin does not reduce the size of the operand relations.

Figure 11 illustrates the performance of the two methods according to the size of the relations (assumed to be equal) for a join selectivity factor of 0.016 and the ratio JS′ equal to 0.4. The curves show the obvious superiority of the method using semijoins. This is because semijoin time is negligible compared to join time, and the operand relations to be joined are smaller with the semijoin method than with the join method.

The performance of the algorithms with an increasing number of processors, as depicted in Figure 12, still indicates that the performance of the semijoin method is preferable for a typical join selectivity factor of 0.016. The difference between the two methods may be reduced somewhat by increasing that factor.

The comparisons show that if a semijoin of the operand relations precedes the application of the nested loop join algorithm, the total time of the join is generally decreased, which confirms the value of the semijoin operator. Similar comparisons were made, based on the sort-merge-join algorithm and the hashing join algorithm, and similar results were found.

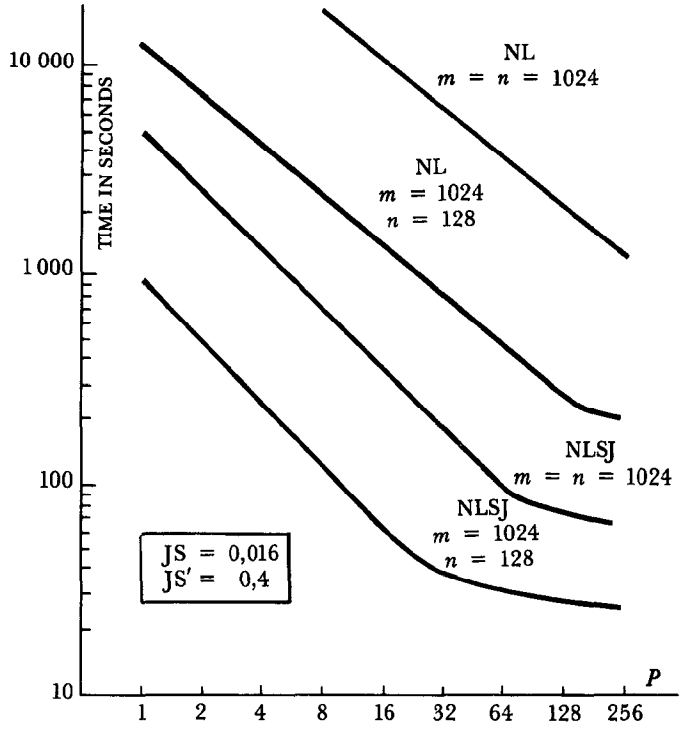

Fig. 12. Comparisons of the algorithms' times versus the number (processors.

7. CONCLUSION

In this paper we have presented and analyzed several algorithms for computing joins and semijoins in a multiprocessor database machine. First, we described the architectural components useful for computing joins in the SABRE database machine, and the time parameters, including I/O, CPU, and communication times, were also described. We then presented and compared three join algorithms according to their execution times. The nested loop join algorithm turns out to be the simplest, and its execution time was found to be inversely proportional to the number of processors. It works best when the number of processors is very high. Furthermore, since each tuple is compared to every other tuple, the nested loop join algorithm easily supports non-equijoin. The sort-merge-join algorithm is more complex and performs better as the operand relations become large. For a certain configuration, the addition of new processors causes very little improvement in performance. But the sort-merge-join algorithm has the merit of utilizing another useful operator (sorting), and yields a result relation which is sorted on the join attributes. The hashing join algorithm is better when the number of matching tuples in the larger relation is small. In that case, using bit arrays results in avoiding much useless access to the hashed file. It is shown that, for a

given configuration, each of the algorithms can excel, according to the characteristics of the operand and result relations. Since the join operation is very important in relational systems, a well-designed database machine would implement various algorithms and, for each join, choose the best one by computing their execution times.

Focusing on the interprocessor transfers in a multiprocessor database machine and considering that semijoin reduces the transmission times, we have presented parallel algorithms for semijoin operations.

Two multiprocessor equi-semijoin algorithms were analyzed. The algorithm using bit arrays gives good results; its execution time is proportional to $na/p + m/p + r/p$, where m is the number of pages of the target relation, na the number of pages containing the projected source semijoin attributes, r the number of pages of the result relation, and p the number of processors. But the main assumption in the application of the algorithm using bit arrays is that the list of distinct semijoin attribute values should fit into the processor's local memory. A second equi-semijoin algorithm using selection was proposed. This algorithm depens upon the parallel filtering power offered by the selection processors assigned to the disks. It is superior to the first algorithm when the semijoin can be computed in a single pass; that is, when the number of distinct semijoin attribute values is less than the number of admitted predicates in a filter. Otherwise, the algorithm using bit arrays is better.

A non-equi-semijoin algorithm was also proposed. It uses the maximum and minimum functions provided by the filter processors and does not need any room in the cache during execution, except for writing the final result. This algorithm has an execution time proportional to $m/d + n/d$, where m and n are the size of the operand relations and d the number of filters associated with the operation.

Comparisons were then performed between the two join strategies. The method of joining two operand relations and the method of joining the result relations of semijoins were comparerd using the nested loop join algorithm. Finally, it was shown that the semijoin method is generally better, which indicates the value of implementing this operator in a database machine.

The results can be extended in several directions. First, the algorithms can be optimized. For example, the postoptimal stage of the multiprocessor b-way merge, which degrades the degree of parallelism, can be improved by using another, more complex, method. Second, other methods, based on index or pre-evaluated joins, can be found. These indices would be organized in such a fashion as to facilitate parallel execution. The performances of the methods would essentially depend on the placement strategies of relations on disk units. An original multiattribute clustering technique [21] will be used in SABRE, and will improve join executions.

APPENDIX A

Analysis of the Nested Loop Algorithm

The algorithm can be divided into several steps, such that each step corresponds to one pass of reading the internal relation. First, let us determine the execution time of one step.

During one step, each processor starts reading $(b - 1)$ pages of R. Thus, $p(b - 1)$ pages of R are read in parallel. The time necessary for reading these pages is

$$t1 = (b - 1)\mathrm{CR}.$$

Then, each page of S is broadcast and joined with $(b - 1)$ pages of R by each processor. The time to read a page of S is

$$t2 = \mathrm{CR}.$$

The time to join $(b - 1)$ unsorted pages of R with one unsorted page of S is

$$t3 = (b - 1)\mathrm{CJ0}.$$

This join generates a result of size $(b - 1)$JS pages. The resulting tuples must be moved to an output buffer; that cost in time is

$$t4 = (b - 1)\mathrm{JS} \cdot t \cdot I.$$

The output buffer, when filled, must be written in cache memory (or on disks). This write requires an average time of

$$t5 = (b - 1)\mathrm{JS} \cdot \mathrm{CW}.$$

Finally, the total time for one step of the algorithm is

$$T1 = t1 + n(t2 + t3 + t4 + t5),$$

which turns out to be

$$T1 = (b - 1)\mathrm{CR} + n(\mathrm{CR} + (b - 1)\mathrm{CJ0} + (b - 1)\mathrm{JS} \cdot t \cdot I + (b - 1)\mathrm{JS} \cdot \mathrm{CW})$$

or, equivalently,

$$T1 = (b - 1)\mathrm{CR} + n(\mathrm{CR} + (b - 1)(\mathrm{CJ0} + \mathrm{JS}(t \cdot I + \mathrm{CW}))).$$

At each step, $p(b - 1)$ pages of R are read. Therefore, the number of steps is

$$k = m/(p(b - 1)).$$

Consequently, the total execution time of the algorithm is

$$\mathrm{TIME(NL)} = k \cdot T1,$$

which turns out to be

$$\boxed{\mathrm{TIME(NL)} = m \cdot \mathrm{CR}/p + m \cdot n(\mathrm{CR}/(p(b - 1)) + (\mathrm{CJ0} + \mathrm{JS}(t \cdot I + \mathrm{CW}))/p).}$$

APPENDIX B

Analysis of the Sort-Merge-Join Algorithm

First, let us evaluate the execution time to sort a relation of n pages. As they are read during the first pass, pages are internally sorted. Each processor internally sorts n/p pages. Thus, the time to internally sort n pages is approximately.

$$t1 = n \cdot Cs/p.$$

During all the passes preceding the optimal stage, each processor is fully utilized. At each pass of the relation a processor performs $n/(p \cdot b)$ merge operations of b pages. The cost of merging b sorted pages is CMb. Thus, the additional cost of each pass before the optimal stage is

$$t2 = n \cdot \mathrm{CM}b/(p \cdot b).$$

At the optimal stage corresponding to the pass K, each processor produces a sorted run of $b^{**}K$ pages. Since each of the p processors has merged exactly n/p pages, the following equation is satisfied:

$$b^{**}K = n/p,$$

and so we have

$$K = \log_b(n/p).$$

Finally, the time necessary to reach the optimal stage is

$$T1 = t1 + K \cdot t2,$$

which turns out to be

$$T1 = n \cdot Cs/p + (N \cdot \mathrm{CM}b \cdot \log_b(n/p))/(b \cdot p).$$

Let us now derive the time necessary after the optimal stage. The optimal stage generates p runs of n/p pages. The number of passes to merge p runs in $\log_b p$. At each pass of the postoptimal stage, the number of processors needed is divided by b, while the work of each is multiplied by b. At the first postoptimal pass, p/b processors perform n/p merge operations of b pages at a cost in time of

$$m1 = \mathrm{CM}b \cdot n/p.$$

More generally, at the ith postoptimal pass, $p/b^{**}i$ processors perform $b^{**}(i-1) \cdot n/p$ merge operations at a cost in time of

$$mi = \mathrm{CM}b \cdot b^{**}(i-1) \cdot n/p.$$

Finally, the total cost of the postoptimal stage is

$$T2 = m1 + m2 + \cdots + mi + \cdots + m \log_b p,$$

which turns out to be

$$T2 = \mathrm{CM}b \cdot (n/p) \cdot (1 + b + b^{**}2 + \cdots + b^{**}((\log_b p) - 1)).$$

Then, applying the well-known formula:

$$1 + b + b^{**}2 + \cdots + b^{**}(k-1) = (1 - b^{**}k)/(1 - b),$$

we obtain

$$T2 = \mathrm{CM}b \cdot (n/p) \cdot (1 - b^{**}\log_b p)/(1 - b).$$

The execution time required to sort a relation of n pages is then:

$$\mathrm{SORT}(n) = T1 + T2,$$

that is,

$$\text{SORT}(n) = n/p[Cs + \text{CM}b\cdot\log_b(n/p) + \text{CM}b(1 - b^{**}\log_b p)/(b - 1)].$$

The execution cost of the algorithm is the cost of sorting the relation R, plus the cost of sorting the relation S, plus the cost of joining the sorted relations by a merge. The uniprocessor merge-type operation consists in reading the sorted relations, joining them, and writing the result relation. The cost of merging m and n pages is then

$$\text{MERGE}(m, n) = (m + n)\text{CR} + \max(m, n)\cdot\text{CJ1} + m\cdot n\cdot\text{JS}\cdot\text{CW}.$$

Finally, the execution time of the sort-merge-join algorithm is

$$\text{TIME(SM)} = \text{SORT}(n) + \text{SORT}(m) + \text{MERGE}(m, n),$$

which turns out to be

$$\begin{aligned}\text{TIME(SM)} = {} & n/p[Cs + \text{CM}b\cdot\log_b(n/p)/b + \text{CM}b\cdot(1 - b^{**}\log_b p)/(1 - b)] \\ & + m/p[Cs + \text{CM}b\cdot\log_b(m/p)/b + \text{CM}b \\ & \qquad \cdot(1 - b^{**}\log_b p)/(1 - b)] \\ & + (m + n)\text{CR} + \max(m, n)\cdot\text{CJ1} + m\cdot n\cdot\text{JS}\cdot\text{CW}.\end{aligned}$$

APPENDIX C

Analysis of the Hashing Join Algorithm

The execution time of the algorithm comprises time $T1$ for hashing the smaller relation by the cache processor, time $T2$ for distributing the larger relation among p processors, time $T3$ for accessing the hashed file, and, finally, time $T4$ for writing the result. The time for broadcasting the Boolean arrays is negligible and, thus, is ignored. We remind the reader that c page frames are available in the cache for the join operation. Thus the creation of the hashed file consists of creating m buckets if $c > m$, or c buckets, otherwise. In the first case, the hashed file could be maintained in cache memory during the entire execution of the join operation. In the latter case, the pages of the same bucket would be linked and written on disk, and retrieved using a table of physical addresses. The time for reading a page for R, taking into account the ratio F, is

$$t1 = (1 - F)T\text{dc}.$$

The time for hashing a page of t tuples is

$$t2 = t(0 + I).$$

The time for writing the hashed file is the time for writing $(m - c)$ pages, since c pages are reserved in cache memory during the join execution. Furthermore, page frames may be available in the cache with the probability F'. Thus the time for writing $(m - c)$ pages from cache to disk is

$$t3 = (m - c)(1 - F')T\text{dc}.$$

Then, the execution time for hashing a relation for m pages is

$$T1 = m(t1 + t2) + t3,$$

which turns out to be

$$T1 = m[(1 - F)T\text{dc} + t(0 + I)] + (m - c)(1 - F')T\text{dc}.$$

The execution time for the second stage is the time for reading the relation S by p processors in parallel, the time for accessing the hashed file, and the time for writing the result relation. Each processor reads n/p pages of the relation S, and, for each of the t tuples of a page, accesses the Boolean array. This leads to a time of

$$T2 = (\text{CR} + t \cdot 0)n/p.$$

An access to the hashed file is needed for each matching tuple of S. The number of matching tuples is defined by the semijoin selectivity factor SS, and each bucket of the hashed file contains (m/c) pages. Thus, for each page of S, the number of pages read from the hashed file is $t \cdot \text{SS} \cdot (m/c) \cdot \text{CR}$, and this occurs (n/p) times for the entire relation S. Therefore, we obtain

$$T3 = (n/p)(m/c)\text{CR} \cdot t \cdot \text{SS}.$$

The time for writing the result relation of size $m \cdot n \cdot \text{JS}$ in parallel by p processors is

$$T4 = m \cdot n \cdot \text{JS} \cdot \text{CW}/p.$$

Finally, the execution time of the multiprocessor hashing join algorithm (H) is

$$\text{TIME(H)} = T1 + T2 + T3 + T4,$$

which turns out to be

$$\text{TIME(H)} = [(1 - F)T\text{dc} + t(0 + I)] + (m - C)(1 - F')T\text{dc} + (\text{n/p})[\text{CR} + t \cdot 0 + t \cdot \text{SS} \cdot (m/c) \cdot \text{CR} + m \cdot \text{JS} \cdot \text{CW}.$$

APPENDIX D

Analysis of the Semijoin Algorithm Using Bit Arrays

The total time of the semijoin algorithm using bit arrays is the sum of the I/O time and the CPU time, which can be represented as

$$\text{TIME (SJH)} = \text{I/O (SJH)} + \text{CPU (SJH)}.$$

(1) *I/O time.* The I/O time consists of the interprocessor page move time, incurred by reading the operand relation, performing the union and merge operations, and finally writing the result relation. First, relation S is projected over the semijoin attributes to relation S'. This operation requires reading S entirely from disk to cache by d filters. The time required for this operation is $t1 = (n/d) \cdot T\text{dc}$. The relation S' has na pages and is read in parallel by p processors for a time $t2 = (na/p) \cdot \text{CR}$.

We suppose that the q bit arrays in a processor's memory are contained in a page, thus the union operation needs p interprocessor moves plus one more to broadcast the result. This operation takes time $t3 = (p + 1) \cdot Tcp$. The merging of p attribute lists is done in $\log_b p$ passes by a b-way merge and, assuming an average list size of $b/2$ pages, the merge I/O time is $t4 = \log_b p \cdot (b/2) \cdot Tcp$.

The time for the second stage of the algorithm consists of reading relation R, of size m, which is $t5 = (m/p) \cdot CR$, and writing the result relation T, whose size depends on the semijoin selectivity factor SR, for a time $t6 = (m/p) \cdot SR \cdot CW$.

Finally, the I/O time is

$$I/O\ (SJH) = t1 + t2 + t3 + t4 + t5 + t6,$$

which may be written as

$$I/O(SJH) = (n/d) \cdot Tdc + (na/p) \cdot CR + [p + 1 + (\log_b p) \cdot (b/2)] \cdot Tcp + (m/p) \cdot (CR + SR \cdot CW).$$

(2) *CPU time.* The CPU time of the semijoin algorithm using bit arrays is now calculated. During the first stage, for each read page of the relation S' (containing t/Ct attribute values), a hashing function is applied to each value, which is then inserted into list W. We define the time of inserting an element in W by Iw. Then, the CPU time for this stage is

$$c1 = (na/p) \cdot (t/Ct) \cdot (0 + Iw).$$

The merge time of the p lists is denoted by Cm. The union of the p arrays needs p comparisons (i.e., $p \cdot 0$ time units). This requires a time

$$c2 = Cm + p \cdot 0.$$

During the second stage, for each read page of R, a hashing function is applied to the t attribute values. Furthermore, for the $t \cdot SR$ tuples held in a page, W is accessed in order to confirm the absence of collision. The list W contains nj attribute values in sorted order. Thus, binary search [22, p. 406] can be applied to W, with a time of $(\log_2 nj) \cdot 0$ per tuple. The move time of the tuples in the result page is $SR \cdot I$ for each read page of R. The time of the second page is therefore,

$$c3 = (m/p) \cdot [0 \cdot (t + t \cdot SR \cdot \log_2 nj) + SR \cdot I].$$

Thus, the total CPU of this algorithm is $c1 + c2 + c3$, that is,

$$CPU(SJH) = (na/p) \cdot (t/Ct) \cdot (0 + Iw) + Cm + p \cdot 0 + (m/p) \cdot [0 \cdot (t + t \cdot SR + \log_2 nj) + SR \cdot I].$$

APPENDIX E

Analysis of the Semijoin Algorithm by Selection

The time of this algorithm also consists of the I/O and the CPU times.

(1) *I/O time.* The algorithm proceeds in two stages. The first stage, similar to

the previous algorithm, projects relation S over the semijoin attributes into relation S', which demands a time

$$t1 = (n/d) \cdot Tdc,$$

and reads the relation S' in parallel by p processors, requiring a time

$$t2 = (na/p) \cdot CR.$$

The merge operation requires the same time as the merge in the previous algorithm, that is,

$$t3 = Tcp \cdot \log_b p \cdot (b/2).$$

The task of the second stage is to perform selection in parallel by d filters on relation R (of size m). One selection on the entire relation requires a time $(m/d) \cdot Tdc$, and may have to be repeated nj/Nc times if nj is greater than Nc. It should be noted that the selection time includes the write time of the result relation in cache. Thus, the time for this second stage is

$$t4 = (nj/Nc) \cdot (m/d) \cdot Tdc.$$

Finally, the I/O time of the algorithm is

$$I/O(SJS) = t1 + t2 + t3 + t4,$$

which turns out to be

$$I/O(SJS) = (n/d) \cdot Tdc + (na/p) \cdot CR + \log_b p \cdot (b/2) \cdot Tcp + (nj/Nc) \cdot (m/d) \cdot Tdc.$$

(2) *CPU time.* The CPU time is incurred by the insertions in the list W containing the semijoin source attribute values. This time is the same as in the previous algorithm, namely $(na/p) \cdot (t/Ct) \cdot Iw$. The merge time of p lists is denoted Cm.

The CPU time for the second stage is due to the initialization of the selection operations. The time of constructing selection predicates with nj attributes in list W is $nj \cdot 0$. The CPU time of the algorithm is then,

$$CPU(SJS) = (na/p) \cdot (t/Ct) \cdot Iw + Cm + nj \cdot 0.$$

ACKNOWLEDGMENTS

The authors wish to thank P. Bernadat, P. Faudemay, K. Karlsson, Y. Viemont, and the other members of the SABRE project, for their helpful discussions. The authors also wish to acknowledge the many valuable comments of T. Ozsu.

REFERENCES

1. AUER, H. ET AL. R.D.B.M.: a relational database machine. Internal Rep. nr8005, Technisch Univ. Braunchweig, June 1980.
2. BABB, E. Implementing a relational database by means of specialized hardware. *ACM Trans. Database Syst. 4*, 1 (March 1979), 1–29.
3. BANCILHON, F.; AND SCHOLL, M. Design of a back-end processor for a database machine. In *Proc. ACM-SIGMOD* (Santa Monica, Calif., May 1980), ACM, New York, 1980.

4. BANERJEE, J.; HSIAO, D.K.; AND BAUM, R.I. Concepts and capabilities of a database computer. *ACM Trans. Database Syst. 3*, 4 (Dec. 1978), 347–384.
5. BERNSTEIN, P.A.; AND CHIU, D.M. Using semijoins to solve relational queries.
6. BERNSTEIN, P.A.; AND GOODMAN, N. Full reducers for relational queries using multiattribute semijoins. In *Proc. 1979 NBS Symposium on Computer Networks* (Dec. 1979), NBS, Washington, D.C.
7. BERNSTEIN, P.A.; AND GOODMAN, N. Inequality semijoins. Tech. Rep. CCA-79-28, Computer Corp. of America, Cambridge, Mass., Dec. 1979.
8. BLASGEN, M.W.; AND ESWARAN, K.P. Storage and access in relational databases. *IBM Syst. J. 16*, 4 (1977), 363–378.
9. BORAL, H.; DEWITT, D.J.; FRIEDLAND, D.; AND WILKINSON, W.K. Parallel algorithms for the execution of relational operations. Tech. Rep. 402, Computer Science Dept., Univ. of Wisconsin, Madison, Jan. 1980.
10. BORAL, H.; DEWITT, D.J.; AND WILKINSON, W.K. Performances evaluation of associative disk designs. In *6th Workshop on Computer Architecture for Nonnumeric Processing* (Hyeres, France, June 1981), ACM, New York, 1981.
11. CHAMBERLIN, D.D.; GILBERT, A.M.; AND YOST, R.A. A history of System-R and SQL/data system. In *7th International Conference on Very Large Data Bases* (Cannes, France, Sept. 1981).
12. CHIU, D.M.; AND HO, Y.C. A methodology for interpreting tree queries into optimal semijoin expressions. *ACM Trans. Database Syst. 5*, 4 (Dec. 1980), 169–178.
13. CODD, E.F. A relational model of data for large shared data banks. *Commun. ACM 13*, 6 (June 1970), 377–387.
14. DEWITT, D.J. Query execution in DIRECT. In *Proc. of the ACM-SIGMOD 1979 International Conference on Management of Data* (May 1979), ACM, New York, 1979, pp 13–22.
15. BORAL, H.; AND DEWITT, D.J. Design considerations for data-flow database machines. In *Proc. of the ACM-SIGMOD Conference on Management of Data* (Los Angeles, Calif., 1980), ACM, New York, 1980, pp 94–104.
16. FAUDEMAY, P. Sur une nouvelle classe de filtres multiexpressions. *J. Machines Bases de Donnees* (Sophia Antipolis, Sept. 1980), INRIA.
17. FINGER, U.; AND MEDIGUE, G. Architectures multimicroprocesseurs et disponibilité: la SM90. *L'echo des Recherches 105* (July 1981).
18. GARDARIN, G. An introduction to SABRE: a multimicroprocessor database machine. In *6th Workshop on Computer Architecture for Nonnumeric Processing* (Hyeres, France, June 1981).
19. GOTLIEB, L.R. Computing joins of relations. In *Proc. of the ACM-SIGMOD Conference on Management of Data* (San Jose, Calif., May 1975), ACM, New York, 1975, pp 55–63.
20. INTEL. Les concepts systemes d'intel: le multibus et ses signaux. E.A.I.263, A. Sabatier, Intel-France, Feb. 1979.
21. KARLSSON, K. Reduced cover-trees and their application in the SABRE access path model. In *7th International Conference on Very Large Data Bases* (Cannes, France, Sept. 1981).
22. KNUTH, D.E. *The Art of Computers Programming; vol. 3: Sorting and Searching*, Addison-Wesley, Reading, Mass., 1973.
23. LEBIHAN, J., ET AL. SIRIUS: a French nationwide project on distributed databases. In *6th International Conference on Very Large Databases* (Montreal, 1980).
24. OZKARAHAN, E.A.; SCHUSTER, S.A.; AND SEVCIK, K.C. Performance evaluation of a relational associative processor. *ACM Trans. Database Syst. 2*, 2 (June 1977), 175–196.
25. PREPARATA, F.P. New parallel-sorting schemes. *IEEE Trans. Comput. C-27* (July 1978).
26. ROTHNIE, J.B., ET AL. SDD1: a system for distributed databases. *ACM Trans. Database Syst. 5*, 1 (March 1980), 1–17.
27. ROHMER, J. Machines et langages pour traiter les ensembles de donnees (textes, tableaux, fichiers). These d'Etat, Grenoble, Dec. 1980.
28. STONEBRAKER, M.; WONG, E.; AND KREPS, P. The design and implementation of INGRES. *ACM Trans. Database Syst. 1*, 3 (Sept. 1976).
29. VALDURIEZ, P. Algorithmes de jointures de relations. *Colloque Les Bases de Donnees* (Tunis, April 1981), AFCET.
30. WAH, B.W.; AND YAO, S.B. DIALOGUE: a distributed-processor organization for a database machine. In *1980 National Computer Conference*, pp 243–253.

Received March 1982; revised February 1983; Accepted May 1983.